

Andrzej Klusiewicz

Python

na luzie



Spis treści

"Hello world" i pisanie na konsoli.....	9
Zmienne i typy danych.....	11
Zmienne.....	11
Typy danych.....	11
Typ tekstowy.....	12
Typy liczbowe.....	12
Instrukcje warunkowe.....	13
Jeden warunek.....	13
Else.....	14
Wiele warunków.....	15
Operatory logiczne w warunkach.....	16
Pętle.....	16
Pętla while.....	16
Pętla for.....	17
Zagnieżdżanie pętli.....	18
Instrukcja BREAK.....	19
Instrukcja CONTINUE.....	19
Łańcuchy znaków.....	20
Funkcje wbudowane.....	20
Upper.....	20
Lower.....	20
Title.....	21
Replace.....	21
len w kontekście ciągów tekstowych.....	21
Count.....	22
Strip.....	22
split i join - zamiana tekstu na listę i listy na tekst.....	23
Łańcuchy funkcji.....	24
Iterowanie po łańcuchach tekstowych.....	24
Mnożenie tekstu. Ale jak?.....	25
Wygodne sprawdzanie czy tekst zawiera frazę.....	25
Czy Python>Java?.....	26
Cięcia, cięcia - o cięciu łańcuchów tekstowych słów kilka.....	27
Listy.....	29
Tworzenie list.....	29
Pobieranie wartości z list.....	30

Iterowanie po listach.....	32
Sprawdzanie czy element znajduje się na liście.....	32
Modyfikowanie zawartości listy.....	33
Dodawanie nowych wartości i wstawianie w miejsce istniejących.....	33
Kasowanie elementów z listy.....	34
Funkcje wbudowane w listy.....	35
Sortowanie i odwracanie list.....	35
Inne ciekawe funkcje i możliwości.....	37
Zaawansowane elementy przetwarzania list i zbiorów.....	39
Listy składane.....	39
Map i filter.....	42
Funkcja map.....	42
Funkcja filter.....	44
Krotki.....	45
Deklaracja i uzupełnianie krotek danymi.....	45
Pobieranie wartości z krotek.....	46
Słowniki.....	47
Tworzenie słowników.....	47
Pobieranie wartości ze słowników.....	48
Modyfikacja zawartości słowników.....	50
Zestawy.....	51
Tworzenie zestawów i konwersje z innych typów złożonych.....	51
Modyfikowanie zawartości zestawów.....	53
Wyjątki.....	54
Obsługa wyjątków.....	54
Funkcje.....	57
Deklarowanie funkcji.....	57
Parametry funkcji.....	57
Zwracanie wyników z funkcji.....	59
Wyrażenia Lambda.....	60
Funkcja jako argument.....	60
Funkcja w funkcji.....	62
Zwracanie funkcji z funkcji.....	63
Rekurencja.....	64
*args.....	66
**kwargs.....	68
Optymalizacja funkcji z użyciem cache funkcji.....	70

Generatory.....	72
Dekoratory.....	77
Funkcja jako argument.....	77
Funkcja w funkcji.....	78
Zwracanie funkcji z funkcji.....	78
Tworzenie dekoratorów.....	79
Dekorowanie funkcji z parametrami.....	80
Dokumentowanie funkcji.....	82
Moduły.....	83
Definiowanie modułów.....	83
Dokumentowanie modułów i sprawdzanie dostępnych funkcji.....	85
Pakiety.....	86
Korzystanie z plików tekstowych.....	87
Czytanie z plików tekstowych.....	87
read().....	88
readlines().....	90
readline().....	91
Funkcja seek().....	91
Sprawdzanie ilości linii w pliku.....	92
Zapis w plikach tekstowych.....	93
Tryby otwarcia pliku.....	93
Wprowadzanie danych do pliku.....	93
Przetwarzanie JSON.....	95
Ładowanie danych JSON z pliku.....	95
Tworzenie i zapisywanie danych JSON do pliku.....	98
Przetwarzanie XML.....	99
Odczyt danych z pliku XML i sięganie do elementu po nazwie.....	99
Sięganie po podelementy.....	102
Sięganie do elementu po pozycji.....	103
Listy wartości w XML i odwoływanie się do "n-tego" wystąpienia tagu.....	104
Atrybuty.....	105
Użyteczne "sztuczki".....	107
Odczytywanie XML jako zwykły tekst.....	107
Sprawdzanie nazwy elementu.....	108
Modyfikowanie drzewa XML.....	109
Modyfikowanie zawartości element.....	109
Dodawanie i modyfikowanie atrybutów element.....	109

Tworzenie nowych elementów.....	110
Usuwanie elementów.....	111
Zapis drzewa XML do pliku.....	112
Dane zdalne - wykorzystanie usług sieciowych.....	113
Pobieranie danych za pomocą GET.....	113
Przesyłanie danych za pomocą POST.....	114
Wykorzystanie baz danych.....	115
Łączenie z serwerem bazy danych.....	115
Łączenie z serwerem PostgreSQL.....	115
Łączenie z serwerem Oracle.....	115
Pobieranie danych z użyciem SELECT.....	116
Wstawianie, zmiana i kasowanie danych, oraz operacje DDL.....	118
Moduł os i poruszanie się po systemie plików.....	119
Funkcja os.walk.....	119
Poruszanie się po systemie plików.....	121
Zmiana i sprawdzenie aktualnego katalogu.....	121
Listowanie zawartości katalogu.....	121
Sprawdzenie czy katalog istnieje.....	121
Sprawdzanie czy mamy do czynienia z plikiem czy z katalogiem.....	122
Sprawdzanie wielkości pliku.....	122
Tworzenie i kasowanie katalogu.....	122
Kasowanie pliku.....	122
Moduł subprocess i wywoływanie komend systemu operacyjnego.....	123
Funkcja call.....	123
Funkcja Popen.....	124
Wyrażenia regularne.....	125
Podstawowe wyszukiwanie.....	125
Typy znaków.....	126
Kwantyfikatory ilościowe.....	126
Wykorzystanie symboli i kwantyfikatorów do wyszukiwania elementów według wzorca.....	127
Testy jednostkowe - framework py.test.....	129
Podstawowe testy.....	129
Uruchamianie wybranych testów.....	133
Parametryzacja testów.....	137
Fikstury.....	141
Problematyka.....	141
Funkcje setup_module i teardown_module.....	143

Dekorator @pytest.fixture.....	145
Makiety (Mocks).....	149
Dane testowe.....	150
Sprawdzanie pokrycia kodu testami.....	151
Wizualizacja danych.....	153
Pierwszy wykres liniowy.....	153
Nanoszenie dodatkowych serii.....	154
Dodawanie legendy do wykresu.....	155
Etykiety osi X i Y.....	156
Zmiana rodzaju i koloru linii.....	157
Siatka na wykresie.....	159
.....	159
Zapisywanie wykresu do pliku.....	160
Wykresy słupkowe.....	161
Zmiana koloru słupków.....	162
Nakładanie serii słupkowych i liniowych na siebie.....	164
Obiektowość w Pythonie.....	165
Deklaracja klasy i pola.....	165
Funkcje w klasie.....	169
Porównywanie obiektów tej samej klasy.....	171
Przeciążanie funkcji.....	172
Metody specjalne.....	173
__init__.....	173
__str__.....	176
__add__ i przeciążanie operatorów.....	177
__getitem__ i __setitem__.....	179
Metody statyczne.....	180
Dziedziczenie.....	182
Dziedziczenie po wielu klasach.....	183
Polimorfizm.....	185
Funkcje prywatne.....	186
Abstrakcja.....	188
Tworzenie własnych wyjątków.....	190
Iteratory.....	193
Wątki.....	195
Pakiet threading.....	196
Wątek jako demon.....	197

Odbieranie wartości z wątku.....	198
Flask – aplikacje internetowe.....	199
Tworzenie projektu i mapowanie pierwszego adresu.....	199
Konfiguracja portu nasłuchu serwera i automatyczna implementacja zmian.....	201
Kod i szablony kodu HTML.....	202
Przekazywanie danych do widoku i jinja2.....	205
Odczyt parametrów z paska.....	210
Pobieranie i umieszczanie danych w sesji.....	216
Obsługa formularzy.....	217
Usługi sieciowe we Flask.....	219
Usługi sieciowe zwracające dane.....	219
Usługi sieciowe przyjmujące dane.....	221
ORM – SQLAlchemy.....	224
Łączenie z bazą i pierwsza encja.....	224
Dodawanie danych do tabeli.....	227
Pobieranie i filtrowanie danych.....	227
Sortowanie wyniku.....	228
Filtracja i sortowanie w jednym.....	228
Zmiana istniejących w bazie danych.....	229
Kasowanie danych.....	230
Podglądanie generowanego SQLa.....	230
Osadzanie aplikacji Flask na Dockerze.....	231
Parsowanie stron internetowych z użyciem BeautifulSoup.....	235
Instalacja pakietu.....	235
Obiekt klasy BeautifulSoup.....	235
Wyszukiwanie elementów i funkcja find.....	237
Wyszukiwanie pierwszego wystąpienia.....	237
Wyszukiwanie po id elementu.....	238
Wyszukiwanie po klasie css.....	239
Wyszukiwanie po atrybutach elementu.....	240
Zagnieżdżanie.....	241
Sięganie do sekcji strony.....	242
Atrybuty elementów.....	242
.....	242
name i string.....	244
Operowanie na listach elementów i funkcja find_all.....	245
Filtrowanie elementów z funkcją find_all.....	246

contents.....	248
Moduł click – obsługa parametrów z linii poleceń.....	252
Stosowanie parametrów.....	252
Stosowanie wielu parametrów.....	254
Automatyczne generowanie pomocy.....	255
Wprowadzanie haseł.....	256
Moduł ipaddress.....	257
Wirtualne środowisko - VENV.....	259
Wzorce projektowe.....	263
Wzorce kreacyjne.....	263
Wzorzec "Singleton"	263
Wzorzec "Fabryka Abstrakcyjna"	266
Wzorzec "Budowniczy"	269

"Hello world" i pisanie na konsoli

Zwykle naukę dowolnego języka programowania rozpoczyna się od wyświetlenia "Hello world" na konsoli. Do pisania na konsoli służy funkcja "print". Przyjmuje ona przez parametr dane do wyświetlenia. Dane te mogą być tekstem, liczbą, datą lub typem złożony. Najprostszy wariant wyglądałby tak:

```
print('Hello world!')
```

Nie ma znaczenia czy użyjesz znaków ", czy ' do objęcia tekstu. Równie dobrze ta instrukcja mogłaby wyglądać tak:

```
print("Hello world!")
```

Jeśli programowałeś w innym języku, zwróć uwagę że na końcu linii w Pythonie nie dajemy średnika, co często czynimy w innych językach programowania.

Wydrukować możesz również liczbę, lub wynik jakiegoś działania matematycznego:

```
print(30)
print(30/10)
```

W drugim przykładzie zadziała to w ten sposób, że najpierw zostanie wykonane działanie matematyczne, a następnie jego wynik zostanie wyświetlony. Print można wykorzystać również do wyświetlania danych typów złożonych:

```
t=[1,2,3,4,5,'nietoperz']
print(t)
```

Powyżej przykład wyświetlania listy. Szczegółami działania list będziemy zajmować się nieco później, na razie nie przejmuj się jeśli nie rozumiesz linii powyżej instrukcji print.

Bywają sytuacje w których musimy wyświetlić jednocześnie tekst i wynik jakiejś operacji matematycznej lub liczbę:

```
print('1/3='+ (1/3))
```

Taka próba zakończy się błędem jak poniżej:

```
print('1/3='+ (1/3))
TypeError: can only concatenate str (not "float") to str
```

Problem tkwi w łączeniu 2 różnych typów danych, tekstu i liczby zmiennoprzecinkowej. Aby ten problem rozwiązać, należy zastosować rzutowanie liczby na tekst:

```
print('1/3='+str(1/3))
```

Tym razem instrukcja działa poprawnie. To za sprawą funkcji str, która przyjmuje dowolny typ danych a zwraca tekst. Pierwotny typ danych jest rzutowany na typ STRING - czyli typ tekstowy.

Taka konkatencja i rzutowanie typów może być bardzo niewygodne, zwłaszcza przy dłuższych ciągach tekstowych. Istnieje wygodniejsza forma:

```
x=123
y=67
print('x={}, y={}'.format(x,y))
```

Przyjrzyj się trzeciej linii. Drukujemy na konsoli ciąg tekstowy i tutaj nie ma nic nowego. Zagadkowa może się wydać końcówka ".format(x,y)". Otóż wszystko w Pythonie jest obiektem. Obiekty mogą mieć wbudowane funkcje. Ciąg tekstowy jest obiektem klasy "string", a ta ma między innymi wbudowaną funkcję format. Posiada też wbudowane funkcje do powiększania i pomniejszania tekstu, zamiany jego fragmentów i wiele innych, ale zajmiemy się nimi w rozdziale "łańcuchy znaków". Z pomocą funkcji format możemy podstawić zmienne w miejsce znaczników {} w tekście. Ponieważ użyliśmy takich znaczników dwa razy, parser będzie oczekiwał dwóch wartości które zostaną podstawione w ich miejsce. Właśnie to robi funkcja format. X trafia w miejsce pierwszego, a Y w miejsce drugiego wystąpienia znaczników. Gdybyśmy nie użyli funkcji format, parser wyświetliłby na konsoli po prostu:

```
x={}, y={}
```

Ponieważ użyliśmy funkcji format, znaczniki {} zostały potraktowane jako elementy dynamiczne. Jeśli zdecydowaliśmy się na użycie tej funkcji, to teraz musimy podać tyle wartości ile razy te znaczniki występują. Gdybyśmy podali ich mniej, dostalibysmy komunikat:

```
print('x={}, y={}'.format(x))
IndexError: tuple index out of range
```

Jeśli damy ich więcej, zostaną użyte tylko pierwsze dwie wartości z podanej listy wartości.

Zmienne i typy danych

Zmienne

Zmienne to takie pojemniki na wartości. Wartości mogą być dowolnego typu. Zmienna służy do chwilowego przetrzymywania jakichś informacji. Przydaje się gdy np. zechcemy użyć tej samej wartości w wielu miejscach. Tworzymy zmienną np. X i przypisujemy do niej wartość. Następnie wykorzystujemy zmienną wszędzie tam gdzie ma pojawić się wartość znajdująca się w zmiennej, a w razie potrzeby zmieniamy tę wartość w jednym miejscu:

```
x=10
print(x*x)
print(x/2)
```

Pierwsza linia to deklaracja zmiennej i przypisanie drugiej wartości. W dwóch kolejnych liniach wykorzystujemy zmienną do wyświetlania wyników operacji matematycznych. Wszędzie tam gdzie pojawia się "x", parser wstawia wartość zawartą w zmiennej "x" - czyli 10, a dopiero później wykonuje obliczenia i wyświetlenie wyniku.

Możemy też przypisywać wartości jednocześnie do kilku zmiennych:

```
x,y=10,20
print(x)
print(y)
```

Wartość 10 zostaje przypisana do x, a wartość 20 do y.

Typy danych

W języku Python deklarując zmienną nie musimy podawać jej typu. Podajemy tylko nazwę zmiennej, oraz wartość jaką do niej przypisujemy. Parser sam rozpozna rodzaj przypisywanych danych. Możemy go z resztą sprawdzić korzystając z funkcji "type", jak w poniższym przykładzie:

```
x=10
print(type(x))
```

Na konsoli zostaje wyświetlone:

```
<class 'int'>
```

co oznacza, że zmienna x jest typu "int" - a więc jest liczbą całkowitą.

Typ tekstowy

Typ tekstowy (string) to typ przeznaczony do przechowywania ciągów tekstowych. Wartość do zmiennej możemy przypisać za pomocą pojedynczych, lub podwójnych cudzysłówów:

```
x='Hiszpańska'  
y="Inkwizycja"
```

oba zapisy są równoznaczne. Możemy użyć obu zmiennych do wyświetlenia komunikatu. Poniżej wykorzystuję do tego celu konkatencję. Pomiędzy zmiennymi dokleiłem jeszcze spację, aby te dwa wyrazy były rozdzielone.

```
print(x+" "+y)
```

Typy liczbowe

W Pythonie mamy dwa rodzaje zmiennych liczbowych (są jeszcze liczby zespolone, ale to dużo bardziej złożony temat). "int" jest typem służącym do przechowywania liczb całkowitych, a "float" zmiennoprzecinkowych. Poniżej przykład użycia rzutowania na oba te typy z typu tekstowego:

```
print(int("1"))  
print(float("1"))
```

Wynik na konsoli:

```
1  
1.0
```

Już nawet po formacie wyświetlanych danych widać jakiego są rodzaju. Gdyby jednak to nam nie wystarczyło, możemy posłużyć się omawianą wcześniej funkcją "type" to weryfikacji typów.

Instrukcje warunkowe

Czasem chcemy coś wykonać w zależności od wystąpienia lub nie jakiegoś warunku. Do takich operacji służą instrukcje warunkowe. W Pythonie mają one taką postać:

if (*jakiś warunek*):

co się ma stać gdy warunek zaistnieje

elif(*jakiś warunek*):

co się ma stać gdy warunek zaistnieje

else:

co się ma stać gdy żaden z powyższych warunków nie zaistnieje

Jeden warunek

Mając w pamięci powyższą strukturę, przeanalizujmy działanie poniższego przykładu. Zaczniemy od najprostszej postaci takiej instrukcji warunkowej:

```
wzrost=190
if(wzrost>180):
    print('kobiety lubią to ;)')
print('no i koniec...')
```

Na początek deklaruje zmienną wzrost o wartości 190. If sprawdza czy wartość tej zmiennej jest większa od 180, a jeśli tak to na konsoli wyświetlany jest stosowny komunikat. Jeśli jednak warunek nie jest prawdziwy, komunikat 'kobiety lubią to' nie jest wyświetlany. Wynika to z faktu że instrukcja odpowiedzialna za wyświetlenie tego komunikatu jest powiązana z warunkiem if.

Kolejna instrukcja print już nie jest związana z wystąpieniem albo nie warunku. Dlaczego? Można to rozpoznać po odstępach instrukcji od lewej krawędzi. Elementy związane z instrukcjami warunkowymi, pętlami, należące do deklarowanej funkcji, to te które są odsunięte w prawo. W chwili gdy parser trafia na instrukcję która nie jest odsunięta w ten sposób, wie że dana instrukcja nie jest już związana z warunkiem, pętlą czy funkcją. Tak więc komunikat 'kobiety lubią to' jest związany ze spełnieniem warunku (wzrost>180), ale komunikat 'no i koniec...' już nie. Pierwszy z komunikatów zostanie wyświetlony tylko jeśli warunek zostanie spełniony, drugi bezwarunkowo. Odstępy robimy (też w zależności od środowiska IDE) spacjami lub tabulatorami. Taka składnia może się wydać nieco abstrakcyjna dla osób które przemigrowały na Pythona z innych języków

programowania m.in Javy (czyli na przykład ja), ale na dłuższą metę okazuje się naprawdę bardzo wygodna.

Else

Else pozwala na określenie co ma się stać gdy warunek okaże się nieprawdziwy:

```
wzrost=176
if(wzrost>180):
    print('kobiety lubią to ;)')
else:
    print('smutna żaba :( ')
print('no i koniec...')
```

Zmieniłem wartość zmiennej "wzrost" w taki sposób by warunek w "if" nie był prawdziwy. Pierwszy komunikat jako związany ze spełnieniem warunku, nie zostaje wyświetlony. Wobec nie spełnienia warunku, wykonywane jest to co związane jest z "else". Na konsoli zostaje wyświetlony komunikat ze smutną żabą.

Wiele warunków

Możemy zechcieć sprawdzić kilka warunków. W zależności od tego czy chcemy sprawdzić je niezależnie od siebie, czy w zależności od zaistnienia lub nie innych warunków, stosujemy albo osobne bloki "if", albo w ramach jednego bloku "if" stosujemy wiele klauzul "elif". Klauzula elif pozwala sprawdzać kolejne warunki, jeśli poprzednie warunki nie były spełnione. Czyli jeśli warunek w "if" nie jest spełniony, to tylko wtedy sprawdzany jest pierwszy elif. Jeśli warunek w pierwszym elif nie jest spełniony, sprawdzany jest następny. Dzieje się tak do czasu natrafienia na spełniony warunek, lub na klauzulę "else". W przypadku spełnienia warunku, kolejne klauzule "elif" w ogóle nie są sprawdzane. Kolejne warunki są więc weryfikowane tylko jeśli wszystkie poprzednie nie były spełnione. Stąd mogłem w poniższym przykładzie posłużyć się swoistymi zakresami.

```
wzrost=166
if(wzrost>180):
    print('kobiety lubią to ;)')
elif(wzrost>170):
    print('przeciętny wzrost')
elif(wzrost>160):
    print('do komandosów nie przyjmą')
elif(wzrost>150):
    print('wzrost nikczemny')
else:
    print('smutna żaba :( ')
print('no i koniec...')
```

Za każdym razem sprawdzam tylko czy wzrost jest większy niż określona wartość, nie sprawdzam natomiast czy jest mniejszy od dolnej wartości granicznej. Komunikat "do komandosów nie przyjmą" powinien zostać wyświetlony przy wzroście 161-170 cm, a ja sprawdzam tylko czy wzrost jest większy od 160, nie sprawdzając czy aby nie przekroczył 170. Dlaczego? Wynika to bezpośrednio z tej kaskadowości warunków. Skoro parser dotarł do warunku sprawdzającego czy wzrost jest większy niż 160, to znaczy że wszystkie poprzednie warunki nie były spełnione. Piętro wyżej sprawdziłem już czy wzrost nie jest większy niż 170, a więc dotarłszy do kolejnego warunku już wiem że wzrost na pewno jest mniejszy bądź równy 170. Pozostaje mi jedynie sprawdzić dolną granicę.

Operatory logiczne w warunkach

W warunkach możemy stosować również operatory logiczne "and" i "or". Ich działanie jest zgodne z logiką matematyczną. W przypadku and obie strony muszą być spełnione by cały warunek był spełniony. W przypadku or wystarczy tylko jedna ze stron by cały warunek był prawdziwy. Oczywiście jeśli obie strony będą spełnione to warunek także jest spełniony.

```
wzrost=185
zarobki=1000000
if(wzrost>180 and zarobki>10000):
    print('gwiazda Tindera ;')
```

Pętle

Pętle służą do wielokrotnego wykonywania jakiejś czynności. Jeśli wiemy ile razy dana czynność ma być wykonana, stosujemy pętlę "for". Jeśli zamierzamy wykonywać jakąś czynność aż do skutku (np. czytanie pliku aż się skończą dane), stosujemy pętlę "while".

Pętla while

Pętla while będzie wykonywana tak długo, jak długo określony warunek jest prawdziwy. Poniższa pętla będzie się wykonywała tak długo, jak długo warunek $x \leq 10$ jest prawdziwy. W chwili gdy warunek przestaje być prawdziwy, pętla jest przerywana. Zmienną użytą w warunku musiałem osobno zadeklarować. Tworząc tego typu pętle należy pamiętać by warunek miał szansę przestać być spełniony, bo jeśli o to nie zadbamy pętla będzie się wykonywała w nieskończoność. Z tego powodu po wypisaniu aktualnej wartości zmiennej x, zwiększam ją o jeden przy każdym "obrocie" pętli. Zapis $x+=1$ znaczy tyle co $x=x+1$, ale jest po prostu krótszy.

```
x=1
while(x<=10):
    print(x)
    x+=1
```


Pętla for

Pętla for będzie wykonywana tyle razy ile określimy poprzez zakres zmiennej. Myślę że najlepiej będzie to zobrazować przykładem.

```
for x in range(1,11):  
    print(x)
```

W powyższej pętli for niejawnie deklarowana jest zmienna x, która jest widoczna tylko w ramach pętli. Służy ona do iteracji. Jej wartość rośnie przy każdym obrocie pętli w zakresie od 1 do 10. Do 10 a nie 11? Tak właśnie, ponieważ jest to zakres niedomknięty z prawej strony. Na konsoli zostają wypisane wartości od 1 do 10. Funkcja range będąca podstawą tej pętli może też przyjąć trzeci argument, określający o ile ma się zwiększać nasz iterator przy każdym obrocie pętli.

```
for x in range(1,11,2):  
    print(x)
```

Po uruchomieniu powyższej pętli, na konsoli zostanie wypisana co druga wartość w zakresie 1-10, a więc 1,3,5,7,9.

Ten trzeci argument może być również liczbą ujemną, gdy chcemy by wartość naszego iteratora malała. Tym razem jednak muszę odwrócić pierwsze dwa argumenty. O ile przy rosnącym iteratorze pierwsza wartość musi być mniejsza od drugiej, o tyle przy malejącym dokładnie odwrotnie. Pamiętaj o zakresie niedomkniętym z prawej strony. Poniższa pętla wyświetla wartości od 10 do 1 na konsoli:

```
for x in range(10,0,-1):  
    print(x)
```

Zagnieżdżanie pętli

Pętle można zagnieżdżać jedna w drugiej. Przyjrzyjmy się poniższemu przykładowi:

```
for y in range(1,11):  
    for x in range(1,11):  
        print('y={}, x={}'.format(y,x))
```

Na każdy obrót zewnętrznej pętli przypada pełny cykl obrotów wewnętrznej pętli. Czyli przy pierwszym obrocie pętli zewnętrznej y jest równy 1, następuje 10 obrotów wewnętrznej pętli. Y wzrasta o 1, następuje pełen cykl obrotów pętli wewnętrznej. Ten proces będzie się powtarzać aż y osiągnie wartość 10. Początek wyniku działania powyższego kodu dla zobrazowania:

```
y=1, x=1  
y=1, x=2  
y=1, x=3  
y=1, x=4  
y=1, x=5  
y=1, x=6  
y=1, x=7  
y=1, x=8  
y=1, x=9  
y=1, x=10  
y=2, x=1  
y=2, x=2  
y=2, x=3  
y=2, x=4  
y=2, x=5  
y=2, x=6  
y=2, x=7  
y=2, x=8
```

.....

Zagnieżdżać możemy również pętle while. Nic nie stoi też na przeszkodzie by pętlę for zagnieżdżyć w pętli while, ale również odwrotnie. Poniżej dający ten sam efekt co ostatni przykład kod z użyciem dwóch pętli while.

```
y,x=1,1  
while(y<=10):  
    x=1  
    while(x<=10):  
        print('y={}, x={}'.format(y, x))  
        x+=1  
    y+=1
```

Instrukcja BREAK

Instrukcja "break" służy do przerywania działania pętli. Jak zawsze obraz wart więcej niż 1000 słów:

```
for x in range(1,101):
    if (x%17.5==0):
        print("szukana liczba to {}".format(x))
        break
```

Powyższy kod szuka pierwszej liczby z zakresu 1-100 podzielnej przez 17.5. Przeszukujemy zakres liczba po liczbie i sprawdzamy czy reszta z dzielenia tej liczby przez 17.5 wynosi 0. Jeśli tak, to jest to poszukiwana przez nas liczba i można przerwać dalsze poszukiwania z użyciem instrukcji break.

Instrukcja CONTINUE

Instrukcja "continue" powoduje przeskok do następnego obrotu pętli. Przykład obrazujący:

```
for x in range(-10,11):
    if (x==0):
        continue
    print(1/x)
```

Powyższy kawałek kodu drukuje na konsoli wynik dzielenia 1 przez kolejne wartości z zakresu -10 do 10. W przypadku trafienia na zero, pojawi się wyjątek dzielenia przez zero. Moglibyśmy oczywiście taki wyjątek obsłużyć (tym jak się to robi zajmiemy się nieco później), albo oprzeć wyświetlanie wyniku dzielenia o warunek sprawdzający czy aby x jest na pewno różny od 0. Skorzystałem z instrukcji "continue", która powoduje przeskoczenie dalszych instrukcji w ramach danego obrotu pętli i przejście do kolejnego jej obrotu. W tym przypadku została pominięta instrukcja wydruku zawierająca problematyczne dzielenie.

Łańcuchy znaków

Zmienne typu "string" będąc tak naprawdę obiektami klasy string posiadają wbudowane funkcje, pozwalające nam na wykonywanie na tych zmiennych różnorodnych operacji.

Funkcje wbudowane

W poniższym zestawieniu prezentuję tylko wybrane funkcje wbudowane, te które uznałem za użyteczne. Pełniejsze zestawienie można znaleźć na przykład tu:

https://www.w3schools.com/python/python_ref_string.asp

Upper

Funkcja upper powiększa ciąg znaków:

```
napis="ministerstwo do spraw niezbyt istotnych spraw"  
print(napis.upper())
```

powyższy kod wydrukuje na konsoli tekst:

MINISTERSTWO DO SPRAW NIEZBYT ISTOTNYCH SPRAW

Zwróć uwagę, że funkcja upper zwraca powiększony ciąg znaków, a nie zmienia zawartości zmiennej napis!

Lower

Funkcja lower pomniejsza ciąg znaków:

```
napis="MINISTERSTWO DO SPRAW NIEZBYT ISTOTNYCH SPRAW"  
print(napis.lower())
```

Wynikiem działania tej funkcji będzie poniższy ciąg tekstowy na konsoli:

ministerstwo do spraw niezbyt istotnych spraw

Zwróć uwagę, że funkcja lower zwraca pomniejszony ciąg znaków, a nie zmienia zawartości zmiennej napis!

Title

Funkcja title powiększa pierwsze litery wszystkich wyrazów w ciągu, a pomniejsza pozostałe litery.

```
napis="MINISTERSTWO DO SPRAW NIEZBYT ISTOTNYCH SPRAW"  
print(napis.title())
```

Powyższy kod wydrukuje na konsoli komunikat o takiej treści:

Ministerstwo Do Spraw Niezbyt Istotnych Spraw

Replace

Ta funkcja pozwala zamieniać wybrane znaki lub ciągi na inne:

```
napis="MINISTERSTWO DO SPRAW NIEZBYT ISTOTNYCH SPRAW"  
print(napis.replace("I", "X"))
```

W ten sposób wydrukujemy na konsoli to:

MXNXSTERSTWO DO SPRAW NXEZBYT XSTOTNYCH SPRAW

len w kontekście ciągów tekstowych

Funkcja len nie jest związana bezpośrednio z klasą string, nie jest funkcją wbudowaną. Jest jednak często wykorzystywana w odniesieniu do łańcuchów tekstu, dlatego tu się pojawia. Jeśli do funkcji len podamy ciąg tekstowy, zwróci nam ona ilość znaków zawartych w tym ciągu.

```
x=len("MINISTERSTWO DO SPRAW NIEZBYT ISTOTNYCH SPRAW")  
print(x)
```

wyświetli nam na konsoli 45.

Count

Count zlicza ilość wystąpień danego ciągu lub znaku w łańcuchu tekstowym:

```
napis="MINISTERSTWO DO SPRAW NIEZBYT ISTOTNYCH SPRAW"  
print(napis.count('IS'))
```

Powyższy kod drukuje na konsoli liczbę 2, ponieważ tyle razy występuje w ciągu "IS". Uwaga - ta funkcja zwraca uwagę na wielkość liter!

Strip

Funkcja strip służy do usuwania białych (domyślnie bez parametru) lub wskazanych znaków lub ciągów.

```
napis="                MINISTERSTWO DO SPRAW NIEZBYT ISTOTNYCH SPRAW    "  
print(napis.strip())
```

Przykładowy kod wyświetli na konsoli napis pozbawiony na początku i na końcu spacji. Funkcja strip może też przyjąć argument:

```
napis="MINISTERSTWO DO SPRAW NIEZBYT ISTOTNYCH SPRAW"  
print(napis.strip("MINI"))
```

W takim wypadku na konsoli zostanie wypisane:

STERSTWO DO SPRAW NIEZBYT ISTOTNYCH SPRAW

Ponieważ z początku (i ewentualnie z końca jeśli taki ciąg tam też występuje) napisu został usunięty ciąg podany jako parametr funkcji strip. Uwaga - ta funkcja zwraca uwagę na wielkość liter!

split i join - zamiana tekstu na listę i listy na tekst

Funkcja split służy do dzielenia łańcuchów tekstowych na części i zwraca pod postacią listy wyrazów. Jeśli do funkcji split nie podamy argumentu, funkcja przyjmie że poszczególne wyrazy rozdzielane są spacją. Taki kod:

```
napis="MINISTERSTWO DO SPRAW NIEZBYT ISTOTNYCH SPRAW"  
print(napis.split())
```

zwraca

['MINISTERSTWO', 'DO', 'SPRAW', 'NIEZBYT', 'ISTOTNYCH', 'SPRAW']

Być może widzisz taki format danych po raz pierwszy - to jest lista, rodzaj zbioru zmiennych w Pythonie. Ten typ będziemy jeszcze bardzo szczegółowo omawiać.

Funkcja split może także przyjąć argument, który będzie wskazywał czym poszczególne wyrazy są rozdzielane.

```
napis="MINISTERSTWO;DO;SPRAW;NIEZBYT;ISTOTNYCH;SPRAW"  
print(napis.split(";"))
```

Powyższy przykład również wyświetli na konsoli ten sam zestaw co wcześniej, przy czym teraz wyrazy były rozdzielane średnikami.

['MINISTERSTWO', 'DO', 'SPRAW', 'NIEZBYT', 'ISTOTNYCH', 'SPRAW']

Łańcuchy funkcji

Ponieważ każda z funkcji łańcuchów tekstowych zwraca zmodyfikowany obiekt tej samej klasy (string), można wywoływać funkcje kaskadowo:

```
napis="MINISTERSTWO DO SPRAW NIEZBYT ISTOTNYCH SPRAW"  
print(napis.strip("MINI").title().replace('i','X'))
```

Każda z tych funkcji będzie wywoływana na ciągu zwróconym przez poprzednią. Z napisu najpierw zostanie usunięte początkowe "MINI", następnie w zwróconym ciągu zostają powiększone pierwsze litery każdego wyrazu, by na końcu podmienić wszystkie małe i na duże X. Powyższy kod wyświetli na konsoli ciąg:

Sterstwo Do Spraw NXezbyt Istotnych Spraw

Iterowanie po łańcuchach tekstowych

Po literach w łańcuchach tekstowych można iterować. Jest nawet specjalny rodzaj pętli który przechodzi po literach:

```
nazwa="Llanfairpwllgwyngyllgogerychwyrndrobwl11lantysiliogogoch"  
for n in nazwa:  
    print(n)
```

W wyniku działania tego kodu na konsoli zostaje wyświetlona każda z liter z ciągu w kolejnych liniach. "n" jest niejawnie deklarowaną w ramach pętli (i widoczną tylko w jej ramach) zmienną reprezentującą przy każdym obrocie pętli kolejną literę z ciągu. Ciekawostka - ten ciąg który przypisałem do zmiennej "nazwa", nie jest jak może się wydawać przypadkowym ciągiem znaków. To najdłuższa na świecie nazwa miejscowości. Mieści się ona w Walii.

Mnożenie tekstu. Ale jak?

```
kriszna= "rama "*5+" "+5*"hare "  
print(kriszna)
```

Wygląda to tak, jakbyśmy mnożyli tekst razy liczbę, a to przecież nie możliwe. Chodzi jednak o co innego. Wydruk zmiennej kriszna wyrzuca na konsolę:

```
rama rama rama rama rama  hare hare hare hare hare
```

....i teraz wszystko powinno być jasne. Mnożenie oznacza po prostu powtórzenie danej frazy n razy.

Wygodne sprawdzanie czy tekst zawiera frazę

Czasem musimy sprawdzić czy gdzieś w tekście znajduje się jakiś znak lub fraza. Jest do tego bardzo wygodna konstrukcja:

```
if ("X" in "SpaceX") :  
    print("ten ciąg zawiera X!")
```

W miejsce X możesz podstawić też dowolną frazę. Ten sam efekt można osiągnąć na przykład wbudowaną funkcją count.

Czy Python>Java?

Tytuł tego podrozdziału to trochę trolling, ale w Pythonie faktycznie można to sprawdzić (z przewidywalnym skutkiem ;)). Sprawdźmy zatem:

```
if("Python">"Java") :  
    print("to chyba jasne :) Tu jest miejsce na  
hejt: .....")  
else:  
    print("coś się spsuło...")
```

Z jakiegoś powodu ten kod wyświetlił na konsoli komunikat:

to chyba jasne :) Tu jest miejsce na hejt:

I to wcale nie jest jakiś żart twórców Pythona. To jest po prostu sprawdzenie czy fraza "Python" znalazłaby się po frazie "Java" gdyby ułożyć je alfabetycznie.

Cięcia, cięcia - o cięciu łańcuchów tekstowych słów kilka

Z łańcuchów stosunkowo łatwo jest wycinać fragmenty na podstawie pozycji znaków. Zaczniemy od wycięcia pojedynczego znaku:

```
lancuch="123456789"  
print(lancuch[2])
```

Zadeklarowałem łańcuch "łańcuch". Dla wygody umieściłem w nim kolejne cyfry, by można było szybko weryfikować działanie wycinania. Wszystkie kolejne przykłady w tym podrozdziale będę opierał na tej samej zmiennej. Korzystając z nawiasów kwadratowych, podając między nimi liczbę mogę pobrać pojedynczy znak z wskazanej pozycji. Pisząc [2] nie wyciągam jednak znaku z drugiej, a z trzeciej pozycji. Pamiętać należy o tym że w Pythonie liczymy od zera. W ten sam sposób możemy podawać pozycję od końca.

```
print(lancuch[-2])
```

Taki zapis będzie oznaczał pobranie drugiego znaku od końca. W naszym przypadku będzie to 8.

```
print(lancuch[2:5])
```

Powyższy przykład wycina znaki "345". To jest wycinanie na zasadzie - od pozycji do pozycji. Tutaj obowiązuje ta sama zasada nieomkniętego z prawej strony zbioru jak w przypadku funkcji range.

W taki sposób:

```
print(lancuch[:5])
```

wytniemy początek łańcucha - do piątej (czyli szóstej) pozycji bez niej samej, czyli w tym przypadku będzie to "12345" Możemy też podawać pozycję od końca. Poniższy kod wytnie ciąg "123456", czyli bez ostatnich trzech pozycji.

```
print(lancuch[:-3])
```

Możliwe jest też też przeskakiwanie co któryś znak. W poniższym przykładzie z ciągu wyciętego od pierwszej do siódmej pozycji wycinam co drugi znak. Uruchomienie tego kodu spowoduje wyświetlenie ciągu "135":

```
print(lancuch[0:6:2])
```

Gdybyśmy chcieli wyciąć z całego tekstu co drugi znak, moglibyśmy to zrobić w sposób mało czytelny:

```
print(lancuch[0:len(lancuch):2])
```

Albo nieco bardziej prześny:

```
print(lancuch[::2])
```

Listy

Listy to struktury danych pozwalające przechowywać zestawy wartości. Posiadają wiele wbudowanych możliwości, jak np. możliwość odwracania, sortowania, szybkiego i łatwego przeszukiwania. Najłatwiej zrozumieć działanie list gdy przyjrzymy się jak działają i co mają do zaoferowania.

Tworzenie list

Puste listy możemy zadeklarować na jeden z dwóch sposobów:

```
lista=[]  
lista2=list()
```

Własności obu list będą takie same. Obie pozostają puste po stworzeniu, posiadają identyczne możliwości. Listę stworzyć możemy również od razu zapelniając ją danymi:

```
lista3=[1,2,3,"Baba Jaga patrzy!"]
```

Ciekawostka - w Pythonie listy mogą zawierać elementy dowolnych rodzajów, mogą to być elementy zupełnie różnych typów. Nie ma potrzeby zabawy w polimorfizm etc. W tym miejscu chciałbym pozdrowić programistów niektórych innych popularnych języków programowania ;) Elementem listy może także być inna lista:

```
lista4=["nie","toperz",123,lista3]
```

Taką listę możesz wydrukować jak każdy inny element. Wydruk powyższej listy wyświetla na konsoli:

```
['nie', 'toperz', 123, [1, 2, 3, 'Baba Jaga patrzy!']]
```

Pobieranie wartości z list

Z listami możemy pracować podobnie jak z łańcuchami tekstowymi. Zamiast litery ze wskazanej pozycji otrzymamy jednak element na danej pozycji zawarty. Pamiętaj jednak że w Pythonie liczymy od zera (nie tylko w Pythonie)! Dla listy "lista4" z poprzedniego podrozdziału możemy pobrać i wyświetlić element z drugiej pozycji w ten sposób:

```
print(lista4[1])
```

Podobnie jak w łańcuchach możemy podawać numer elementu od końca. Pobranie drugiego od końca elementu:

```
print(lista4[-2])
```

Możemy wycinać również zakresy list korzystając z pozycji. Podobnie jak w przypadku łańcuchów tekstowych, pamiętamy o niedomknięciu zbioru z prawej strony. Jeśli więc chcemy z listy:

```
lista4=["nie","toperz",123,lista3]
```

pobrać element drugi i trzeci, to robimy to tak:

```
print(lista4[1:3])
```

A co jeśli zechcemy pobrać elementy od drugiego do przedostatniego? Możemy to zrobić na co najmniej dwa sposoby. Pierwszy:

```
print(lista4[1:-1])
```

i drugi:

```
print(lista4[1:len(lista4)-1])
```

W powyższym przykładzie posłużyłem się funkcją len, którą znamy ze sprawdzania długości ciągu tekstowego. W przypadku list sprawdza ona ilość elementów na liście.

Sposób korzystania zakresów jest identyczny jak przy łańcuchach. Jeśli więc chcemy pobrać wszystko do pozycji 3 włącznie (czyli do indeksu 2), zrobimy to tak:

```
print(lista4[:3])
```

Analogicznie pobranie wszystkich elementów od indeksu numer 2 (czyli od trzeciej włącznie pozycji), zrobimy to tak:

```
print(lista4[2:])
```

Wyświetlenie całej zawartości listy w sposób nieco bardziej finezyjny niż "print(lista4)":

```
print(lista4[:])
```

Iterowanie po listach

Jeśli zechcemy przejść po wszystkich elementach listy i coś z nimi zrobić, np. wydrukować na ekranie (czy zapisać do pliku) istnieje bardzo wygodna metoda:

```
li = ["siała", "baba", "mak", "ale nałożyli", 23, "%", "VAT"]
for l in li:
    print(l)
```

Tą pętlę możesz już kojarzyć z łańcuchów tekstowych. W tym przypadku "l" nie jest kolejnym znakiem z łańcucha, a kolejnym elementem listy. Tak więc pętla obróci się tyle razy ile będzie elementów w liście, a "l" za każdym jej obrotem będzie reprezentowało kolejny jej element. Powyższy kod przejdzie po wszystkich elementach listy i wydrukuje na konsoli wszystkie jej elementy po kolei. Jest też sposób dla osób które preferują mniej czytelny kod i łamanie palców na klawiaturze:

```
for i in range(len(li)):
    print(li[i])
```

W tym przypadku "i" będzie po prostu liczbą która rośnie przy każdym obrocie pętli o jeden, aż osiągnie wartość równą długości listy. W funkcji print podaję po prostu kolejne elementy listy wybierając je po indeksie.

Sprawdzanie czy element znajduje się na liście

Ponieważ znaleźliśmy już nieco analogii do łańcuchów tekstowych, pewnie nie będzie zaskoczeniem że istnieje specjalna konstrukcja bardzo podobna do tej z łańcuchów (w zasadzie tak naprawdę ta sama) która służy do wymienionego w tytule celu:

```
poszukiwani=["Michael Scofield", "Lincoln Burrows", "Theodore Bagwell", "Uczciwy polityk"]
if("Andrzej Klusiewicz" in poszukiwani):
    print("pszypau")
else:
    print("nie pszypau")
```


Modyfikowanie zawartości listy

Dodawanie nowych wartości i wstawianie w miejsce istniejących

Poniżej prezentuję kilka przykładów dodawania i podmiany elementów na liście. Przyjrzyj się im i za chwilę je omówimy:

```
lista=[1,2,3,4,5,6,7]
lista.append(8)
print(lista)
lista.insert(0,"X")
print(lista)
lista[1]="Y"
print(lista)
```

Uruchomienie tego kodu wydrukiuje na ekran trzy linie:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

```
['X', 1, 2, 3, 4, 5, 6, 7, 8]
```

```
['X', 'Y', 2, 3, 4, 5, 6, 7, 8]
```

Funkcja "append" powoduje dodanie elementu na końcu listy. Widzimy to w pierwszej wydrukowanej linii - przybyła jedna cyfra na końcu. Funkcja "insert" podstawia wartość w miejsce wskazanego indeksu (tutaj 0) przesuując resztę elementów w prawo. Widzimy to w drugiej wydrukowanej linii. Zapis typu "lista[indeks]=wartość" nie rozsuwa listy jak insert, a podmienia element pod wskazanym indeksem.

Kasowanie elementów z listy

I tu mamy kilka opcji. Ponownie przyjrzymy się dostępnym wariantom na zasadzie kod + wynik. Najpierw kod:

```
lista=[1,0,2,0,3,0,4,0,5,0,6,0,7,0,3]
del lista[2]
print(lista)
lista.remove(3)
print(lista)
del lista[0:4]
print(lista)
lista.clear()
print(lista)
```

Teraz wynik:

```
[1, 0, 0, 3, 0, 4, 0, 5, 0, 6, 0, 7, 0, 3]
```

```
[1, 0, 0, 0, 4, 0, 5, 0, 6, 0, 7, 0, 3]
```

```
[4, 0, 5, 0, 6, 0, 7, 0, 3]
```

```
[]
```

Instrukcja "del lista[2]" powoduje skasowanie elementu o podanym indeksie i zsunięcie listy. Tym sposobem usunęliśmy element zawierający cyfrę "2", co widzimy w pierwszej linii wyniku.

Instrukcja "lista.remove(3)" usunie TYLKO PIERWSZE wystąpienie wartości 3 z listy. Widzimy to w linii drugiej wyniku. Trójka z końca nie została usunięta.

"del lista[0:4]" kasuje elementy o indeksach 0-3, co widzimy w linii trzeciej wyniku. W tym przypadku posługujemy się taką samą składnią zakresów jak w przypadku pobierania elementów z listy.

"lista.clear()" kasuje całą zawartość listy, widzimy efekt jej działania w linii czwartej wyniku.

Funkcje wbudowane w listy

Funkcje wbudowane przedstawiam wybiórczo, prezentuję te które uważam za najbardziej użyteczne moim zdaniem. Zainteresowanych pozostałymi funkcjami odsyłam do dokumentacji.

Sortowanie i odwracanie list

W przypadku list składających się z elementów prostych - ciągów tekstowych albo liczb (ale nie jednego i drugiego w jednej liście) sortowanie jest bardzo łatwe. Przyjrzyjmy się:

```
pdk=["Ogórek", "Pomidor", "Ziemniak", "Marchew", "Steven Hawking"]
pdk.sort()
print(pdk)
```

Wynik:

```
['Marchew', 'Ogórek', 'Pomidor', 'Steven Hawking', 'Ziemniak']
```

Funkcja "sort" posortowała naszą listę alfabetycznie. Uwaga - działa ona bezpośrednio na liście, nie zwraca osobnej zmodyfikowanej listy. Gdyby na liście znajdowały się liczby, zostałyby one posortowane rosnąco. Gdyby lista składała się z mieszanych typów, dostalibyśmy wyjątek:

TypeError: '<' not supported between instances of 'int' and 'str'

Na potrzeby przykładu dodałem do powyższej listy cyfrę.
Co jednak jeśli zechcemy posortować malejąco? W takim przypadku przyda nam się funkcja wbudowana "reverse":

```
pdk=["Ogórek", "Pomidor", "Ziemniak", "Marchew"]
pdk.sort()
pdk.reverse()
print(pdk)
```

I oczywiście wynik:

```
['Ziemniak', 'Pomidor', 'Ogórek', 'Marchew']
```

Ten sam efekt możemy uzyskać stosując dodatkowy przełącznik funkcji "sort":

```
pdk=["Ogórek","Pomidor","Ziemniak","Marchew","Steven Hawking"]
pdk.sort(reverse=True)
print(pdk)
```

Wynik taki sam jak ostatnio:

```
['Ziemniak', 'Pomidor', 'Ogórek', 'Marchew']
```

Wiemy już że listy mogą składać się z elementów różnych typów, w tym typów złożonych. Każdy z elementów listy sam również może być listą. Rozważmy taki przykład w kontekście sortowania:

```
furki=[ [3,"Renault"], [2,"Citroen"], [1,"Audi"], [4,"Zaporożec"] ]
furki.sort()
print(furki)
```

Sortowanie w takim wypadku zadziała, pewnie nawet można się domyślić że posortuje po pierwszej wartości z każdej z list:

```
[[1, 'Audi'], [2, 'Citroen'], [3, 'Renault'], [4, 'Zaporożec']]
```

A co jeśli zechcielibyśmy posortować alfabetycznie po markach? Także istnieje taka możliwość. Musimy jednak skorzystać z dodatkowej funkcji "itemgetter" znajdującej się w paczce "operator". Musimy tę funkcję zaimportować, zanim zaczniemy z niej korzystać.

```
from operator import itemgetter
furki=[ [3,"Renault"], [2,"Citroen"], [1,"Audi"], [4,"Zaporożec"] ]
furki.sort(key=itemgetter(1))
print(furki)
```

Wynik:

```
[[1, 'Audi'], [2, 'Citroen'], [3, 'Renault'], [4, 'Zaporożec']]
```

Zwróć uwagę na trzecią linię kodu. Zastosowałem dodatkowy parametr funkcji sort - "key" łącząc go z wywołaniem funkcji "itemgetter". W funkcji "itemgetter" jako parametr podałem indeks elementu podlisty, po którym ma być posortowana zewnętrzna lista.

Inne ciekawe funkcje i możliwości

Zacniemy od sprawdzania długości listy, oraz ilości wystąpień określonego elementu w liście. Jeśli chcemy sprawdzić ilość elementów w liście, wystarczy posłużyć się znaną nam już i używaną dotąd kilkakrotnie uniwersalną funkcją "len". Gdy zechcemy sprawdzić ile razy występuje jakiś element w liście, posługujemy się funkcją "count":

```
lista=[1,3,5,8,13,21,34,55,1,1,1,"Batman"]
print(len(lista))
print(lista.count(1))
```

Z listy możemy też kopiować dane. Poniżej pokazuję dwa sposoby. Albo posługując się zakresami (linia druga), albo stosując funkcję "copy" kopiującą całą listę (czwarta linia). Możnaby zapytać czemu nie zrobić po prostu "kopia=lista"? Takie przypisanie odbywa się przez wskaźnik a nie przez kopię i spowodowałoby, że kopia byłaby w rzeczywistości tą samą listą. Modyfikacje na kopii powodowałyby również modyfikacje na oryginalnej liście.

```
lista=[1,3,5,8,13,21,34,55,1,1,1,"Batman"]
podlista=lista[2:7]
print(podlista)
kopia=lista.copy()
print(kopia)
```

Wynik:

```
[5, 8, 13, 21, 34]
[1, 3, 5, 8, 13, 21, 34, 55, 1, 1, 1, 'Batman']
```

W odniesieniu do list możemy też stosować funkcje agregujące:

```
fib=[1,3,5,8,13,21,34,55]
print( sum(fib) , max(fib) , min(fib) )
```

Warunkiem jest jednak to by wszystkie elementy listy były liczbami. Wynik:

```
140 55 1
```

Nieco enigmatyczny może wydawać się zapis:

```
print( sum(fib) , max(fib) , min(fib) )
```

Dotychczas funkcja "print" przyjmowała tylko jeden argument, a tu nagle trzy (?)... Nadal przyjmuje jeden argument, z tym że argumentem tym jest krotka :) Jest to specjalny złożony typ danych, którym zajmiemy się w kolejnym podrozdziale.

Z innych ciekawych elementów przedstawię jeszcze funkcję "index". Jest to metoda sprawdzająca pod jakim indeksem na liście znajduje się wskazany element:

```
fib=[1,3,5,8,13,21,34,55]  
print(      fib.index(13)      )
```

Gdyśmy jako argument podali element którego na liście nie ma, otrzymalibysmy wyjątek:

ValueError: 14 is not in list

Zaawansowane elementy przetwarzania list i zbiorów

Listy składane

W praktyce programisty często zdarza się sytuacja w której na podstawie zawartości jednej listy tworzysz inną listę, często filtrując na podstawie zawartości. Przyjrzyjmy się jak to można zrobić przy pomocy dotychczas znanych metod i porównajmy to z zastosowaniem list składanych.

Naszym celem jest na podstawie 20 liczb w zakresie 1-20 wygenerować listę elementów parzystych z tego zbioru. Przede wszystkim potrzebujemy tej źródłowej listy:

```
liczby=[]  
for x in range(1,21): liczby.append(x)  
print(liczby)
```

Nasz zestaw danych wejściowych wygląda tak:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

Sprawdźmy jak wybralibyśmy z tej listy liczby parzyste i dodali je do innej listy dotychczas znanymi metodami:

```
parzyste=[]  
for i in liczby:  
    if(i%2==0):  
        parzyste.append(i)  
print(parzyste)
```

Jest to jakiś sposób, ale też jest sposób krótszy z użyciem list składanych właśnie:

```
parzyste=[i for i in liczby if i%2==0]  
print(parzyste)
```

Skutek działania powyższego kodu jest taki sam jak tego poprzedniego. Co tu się stało? Zmienna "i" reprezentuje każdy element listy podanej w klauzuli "in liczby". Czyli i to po prostu kolejne liczby z naszej listy liczb. Tak więc zapis:

```
parzyste=[i for i in liczby]
```

oznaczałoby przypisanie do nowej listy o nazwie "parzyste" wszystkich liczb ze zbioru "liczby". Konstrukcja "for i in liczby" powinna Ci już być znana. Oznacza ona "dla każdego elementu i ze zbioru liczby". Co oznacza to "i" przed "for"? Tutaj określamy co ma trafić do listy docelowej - czy źródłowy element, czy też może w jakiś sposób przetworzony. Rozważmy taki przykład:

```
parzyste=[i for i in liczby]
print(parzyste)
```

Sprawdzenie zawartości listy "parzyste" pokazuje nam tę samą zawartość co w liście "liczby":

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Zmodyfikujemy teraz nieco nasz kod:

```
parzyste=[i*10 for i in liczby]
print(parzyste)
```

Sprawdźmy co teraz znajduje się na liście "parzyste":

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200]
```

Co tu się stało? Na początku klauzuli zamiast "i for i in" jest "i*10 for i in". Pojawiło się dodatkowe mnożenie. Jak widzimy, do listy docelowej trafiły elementy z listy pierwotnej pomnożone przez 10. Taką właśnie funkcję spełnia element przed "for". Pozwala Ci na jakieś modyfikacje elementów które trafiają do docelowej listy. To może być cokolwiek - mnożenie jak tu, jakaś unifikacja tekstów czy zastosowanie dowolnej innej funkcji która ma zadziałać na elemencie zanim ten trafi do docelowej listy.

Połączmy to teraz i wyjaśnijmy krok po kroku:

```
parzyste=[i for i in liczby if i%2==0]
print(parzyste)
```

1. "parzyste=" - powstanie nowa lista o nazwie parzyste do której przypiszemy....
2. "[i for" - każdy element z listy pierwotnej trafia do docelowej bez żadnej modyfikacji.
3. "i in liczby" - przetwarzamy elementy z listy "liczby" i do każdego z nich odnosimy się per "i"
4. "if i%2==0]" - do docelowej listy trafiają tylko liczby parzyste-czyli podzielne przez 2 bez reszty

A teraz całość razem: "Stwórz listę 'parzyste' do której trafiają elementy i bez żadnej modyfikacji z listy 'liczby', które spełniają warunek podzielności przez 2 bez reszty" :)

Skoro już tak dobrze się bawimy, to przeróbmy cały nasz kod dalej:

```
liczby=[]
for x in range(1,21): liczby.append(x)
print(liczby)
parzyste=[i*10 for i in liczby if i%2==0]
print(parzyste)
```

Na ten moment wygląda to tak... czyli paskudnie. Skoro już wiemy jak działają listy składane, to postarajmy się to maksymalnie skrócić. Zaczniemy od stworzenia listy liczb w określonym zakresie za pomocą list składanych. Zamiast zapisu:

```
liczby=[]
for x in range(1,21): liczby.append(x)
```

Zapiszmy to tak:

```
liczby=[i for i in range(1,21)]
```

Co się stało się? Stworzyłem listę 'liczby' do której przypisałem bez żadnej modyfikacji elementy z zakresu 1-20. Idźmy krok dalej. Skoro zależy mi na tym by po prostu wyświetlić listę liczb parzystych w określonym zakresie, to wcale nie potrzebuję 2 list, nazywania ich i ich przepisywania. Równie dobrze mogę zrobić to tak:

```
print( [i for i in range(1,21) if i%2==0] )
```

Czyż Python nie jest piękny? :)

Map i filter

Funkcja map

Funkcja "map" służy do stosowania funkcji na każdym z elementów listy. Poniższy kod powoduje przetworzenie listy "liczby" w taki sposób, by do listy "parzyste" trafiły wartości z listy "liczby" pomnożone przez 2.

```
liczby=[i for i in range(1,11)]  
parzyste=list(map(lambda x:x*2,liczby))  
print(parzyste)
```

Wynik działania funkcji:

[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

W zasadzie interesować nas będzie tylko jedna linia z kodu:

```
parzyste=list(map(lambda x:x*2,liczby))
```

Element list(.....) powoduje oczywiście powstanie listy. Samo wywołanie funkcji map to:

```
map(lambda x:x*2,liczby)
```

Do tej funkcji trafiają dwa parametry - funkcja do wykonania (zaraz do tego wróć) i lista źródłowa. Co oznacza ten tajemniczy zapis?:

```
lambda x:x*2
```

Jest to tak zwane wyrażenie lambda. Polega to w skrócie na tym, że możesz podać dynamicznie treść funkcji która ma zostać wykonana. W tym przypadku musimy podać treść funkcji która zostanie wykonana na każdym z elementów listy źródłowej. Zapis "x:" oznacza że ta dynamiczna funkcja przyjmuje jeden parametr, a wewnątrz tej funkcji będę go nazywał "x". Fragment "x*2" określa sposób przetworzenia tego elementu podanego przez parametr, zanim zostanie on zwrócony.

Zasadniczo zapis:

```
lambda x:x*2
```

Byłby odpowiednikiem

```
def funkcja(x):  
    return x*2
```

Czyli na każdym elemencie z listy źródłowej zadziała taka funkcja, a element po przetworzeniu trafi do listy docelowej. W zasadzie ten zapis:

```
parzyste=list(map(lambda x:x*2,liczby))
```

Jest równoznaczny ze znaną nam już konstrukcją list składanych :

```
parzyste=[i*2 for i in liczby]
```

Jeśli przyszło Ci to na myśl, to bardzo dobrze - oznacza to że rozumiesz obie techniki. Owszem - wynik będzie ten sam, natomiast różne są wewnętrzne metody jego osiągnięcia. W zależności od stosowanego przeliczenia i zawartości listy mogą charakteryzować się różną wydajnością.

Co do samych wyrażeń lambda, na razie nie będziemy ich bardziej zgłębiać bo ten poziom zaznajomienia w kontekście list nam w zupełności wystarczy. Przyjrzymy się wyrażeniom lambda w rozdziale w którym omawiam funkcje.

Funkcja filter

Funkcja filter jak sama nazwa wskazuje, służy do filtrowania zawartości list. Tradycyjnie zaczniemy od przykładu kodu:

```
liczby=[i for i in range(1,21)]
parzyste=list(filter(lambda x: x%2==0 ,liczby))
print(parzyste)
```

I równie tradycyjnie sprawdzimy co nam ten kod wydrukował:

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Istotny jest w zasadzie tylko element:

```
parzyste=list(filter(lambda x: x%2==0 ,liczby))
```

Ponieważ po ostatnim niespodziewanym porównaniu funkcji "map" do list składanych Twoja podejrzliwość zapewne nieco wzrosła, to tym razem od razu wyłożę karty na stół:

```
parzyste=[x for x in liczby if x%2==0]
```

Tym razem w wyrażeniu lambda oczekuje się od nas warunku logicznego stwierdzającego "prawda" lub "fałsz" dla każdego z filtrowanych elementów. Robi to fragment "x%2==0" dla którego jesteśmy w stanie stwierdzić dla każdego elementu czy podane wyrażenie będzie prawdziwe czy nie. Będzie prawdziwe oczywiście tylko dla liczb parzystych, więc tylko takie trafią do listy docelowej.

Krotki

Krotki w zasadzie podobne są bardzo do list. Właściwie to krotka jest taką listą której zawartości nie można zmieniać, w związku z czym nie posiada np. funkcji sortowania czy odwracania które działają przecież bezpośrednio na zbiorze. Nie posiadają też np. funkcji insert czy append. Zawartość krotki deklarujemy przy jej definicji, potem nie możemy już jej zmienić.

Skoro krotki są takimi ograniczonymi listami to właściwie po co nam one? Typ danych który po zainicjalizowaniu zmiennej jest "tylko do odczytu" może być czasami przydatny. Ponadto krotki działają szybciej niż listy, z tego też powodu wiele bardzo użytecznych funkcji z różnorodnych bibliotek Pythona zwraca nam właśnie taki typ danych.

Deklaracja i uzupełnianie krotek danymi

Krotki podobnie jak listy mają dwa możliwe warianty deklaracji:

```
krotka=("Tytanowy Janusz","Molibdenowy Mateusz",777,"Przypadkowy  
tekst",[12,55,"koza"])  
krotka2=tuple()
```

Przyjrzyjmy się pierwszej linii. Deklaracja krotki bardzo przypomina deklarację listy, gdyby nie to że używamy nawiasów okrągłych w miejsce kwadratowych. Jak widać w przykładzie, krotka podobnie jak lista może przechowywać elementy różnorodnych typów, w tym np. listy. Piętro niżej pokazuję "krotkowy" odpowiednik "lista=list()".

Pobieranie wartości z krotek

Tu zasady są identyczne jak w przypadku list. Stosujemy te same konstrukcje do pobierania i iterowania po elementach. Nie będę powtarzał tych samych przykładów z wyjaśnieniami zmieniając tylko nawiasy kwadratowe na okrągłe, a słowo lista na krotka ponieważ zostało to już dosyć szczegółowo opisane w rozdziale dotyczącym list. Na takiej samej zasadzie działa sprawdzanie długości krotki (funkcją "len"), również sprawdzenie czy element znajduje się w krotce. Aby jednak nie kończyć tego tematu tym lakonicznym opisem, poniżej przedstawiam kilka przykładów:

```
madness=("Longer","jeśli","to","czytasz i nie  
jesteś","lamus","to","podrzucić","jakiegoś","dobrego","mema")  
print(madness[2:7])  
print(madness[0:5:2])  
if "Longer" in madness:  
    print("Wygrałem! :D")  
for m in madness:  
    print(m)
```

Słowniki

Słowniki są zbiorem składającym się z klucza i wartości powiązanej z tym kluczem. Mogą spełniać taką samą rolę jak słowniki w bazach danych. Przypuśćmy że w hurtownii produktów AGD-RTV mamy wiele produktów wielu marek i wielu rodzajów. Na każdym z produktów znajduje się naklejka z kodem typu "AMICA561", które wykorzystujemy do odnalezienia szerszych informacji o produkcie. Oczywiście pod danym kluczem/kodem mogą znajdować się informacje tylko o jednym produkcie. Dwa produkty nie mogą posiadać tego samego kodu. W podobny sposób działają słowniki w Pythonie. Zwykle korzystając z klucza będziemy wyszukiwać wartość. Słowniki podobnie jak listy (i przeciwnie do krotek) są modyfikowalne.

Tworzenie słowników

Podobnie jak w przypadku list czy krotek, mamy kilka wariantów tworzenia słowników. Poniżej przykład tworzenia pustych słowników:

```
sloownik={}
sloownik2=dict()
```

Możemy też tworząc słownik, od razu zapełnić go danymi:

```
info={
    "LG123": "Telewizor 60' z HD Ready, wejściem na internety i  
filtrem reklam",
    "SONY666": "Piekielnie dobry telewizor",
    "SZAJ Sung999": "Telewizor świetnie nadający się do zakrycia  
dziury w ścianie (i niczego więcej)"
}
```

Wartości "LG123", "SONY666", "SZAJ Sung999" są w powyższym słowniku kluczami i to właśnie z nich korzystając będziemy wyszukiwali wartości. Wartości znajdują się po prawej stronie każdej wartości klucza po znaku ":". Kluczem mogą być nie tylko ciągi tekstowe, ale również liczby czy typy złożone. Podobnie rzecz ma się z wartościami

Pobieranie wartości ze słowników

Wartości ze słowników pobieramy na podstawie klucza. Poniższa konstrukcja może się wydać znajoma. Być może skojarzysz ją z pobieraniem wartości z krotek czy list, choć tam pobieraliśmy wartości za pomocą indeksu który był liczbowy. W tym przypadku nie chodzi jednak o indeks elementu, a o wartość klucza. Nadałem kluczom wartości tekstowe by uniknąć nieporozumień. Gdybyś wywołał np. "info[2]" to nie oznaczałoby to pobrania elementu z trzeciej pozycji (jak w przypadku list czy krotek) a element o kluczu "2" (który wcale nie musiałby się na tej pozycji znaleźć). Same klucze mogą być mieszanych typów. W obrębie jednego słownika możesz mieć zarówno klucze tekstowe jak i liczbowe.

```
print( info["SONY666"] )
```

Po słownikach możemy iterować w podobny sposób jak po listach czy krotkach, z tą różnicą że tutaj iterator jest kluczem a nie wartością:

```
for i in info:
    print(info[i])
```

Taki zapis będzie równoznaczny z:

```
for k in info.keys():
    print(info[k])
```

Wynik działania powyższego kodu:

Telewizor 60' z HD Ready, wejściem na internety i filtrem reklam

Piekielnie dobry telewizor

Telewizor świetnie nadający się do zakrycia dziury w ścianie (i niczego więcej)

"i" w tym przypadku reprezentuje kolejne klucze w ramach słownika. Przechodzimy po całej długości słownika i korzystając z kolejnych wartości klucza wywołujemy "info[i]", czyli pobieramy kolejne wartości na podstawie kolejnego klucza.

W drugim przypadku skorzystałem z wbudowanej funkcji "keys" z której skorzystanie w takim kontekście da nam taki sam wynik jak poprzednie rozwiązanie. Słowniki posiadają też funkcję

"values" która działa na podobnej zasadzie jak "keys", z tą różnicą że zamiast listy kluczy zwraca listę wartości (po której również możemy iterować). Lets czek dis ałt:

```
print(info.keys())
```

zwraca:

```
dict_keys(['LG123', 'SONY666', 'SAJSUNG999'])
```

a:

```
print(info.values())
```

zwraca:

```
dict_values(['Telewizor 60' z HD Ready, wejściem na internety i filtrem reklam', 'Piekielnie  
dobry telewizor', 'Telewizor świetnie nadający się do zakrycia dziury w ścianie (i niczego  
więcej)'])
```

Podobnie jak w listach czy krotkach, możemy analogiczną konstrukcją sprawdzić czy obiekt znajduje się na liście:

```
if "LG123" in info:  
    print("Mamy taki produkt")  
else:  
    print("niet :(")
```

W podobny sposób możemy przeszukać wartości zamiast kluczy, i tu właśnie skorzystamy z funkcji "values":

```
if "Telewizor 60' z HD Ready, wejściem na internety i filtrem  
reklam" in info.values():  
    print("mamy produkt o takim opisie")  
else:  
    print("taki opis nie pasuje do żadnego produktu")
```

Sprawdzana w "if" wartość musi jednak dokładnie odpowiadać wartości ze słownika. Zwracam na to uwagę ponieważ w przypadku omawianych wcześniej łańcuchów tekstowych konstrukcja typu "if 'wartość' in 'coś coś coś wartość' zwróciłaby True i mógłbyś się tym zasugerować.

Modyfikacja zawartości słowników

Dodajemy i nadpisujemy elementy słowników w taki sam sposób:

```
info["KLUCZ"]="WARTOŚĆ"
```

Jeśli w słowniku nie będzie elementu o podanym kluczu to zostanie od dodany. Jeśli będzie, zostanie nadpisany.

Kasowanie elementu ze słownika odbywa się w podobny sposób jak w krotkach czy listach, z tą różnicą że zamiast indeksu elementu podajemy jego klucz:

```
del info["LG123"]
```

Zestawy

Zestawy to takie specyficzne zbiory elementów bez powtórzeń. W zestawach żadna wartość nie może się powtórzyć dwa razy. Są użyteczne np. do usuwania duplikatów. Dodatkowo zestawy automatycznie się sortują.

Tworzenie zestawów i konwersje z innych typów złożonych

Podobnie jak w listach, krotkach i słownikach, także i tu mamy różne sposoby deklaracji zestawów. Możemy zadeklarować pusty zestaw takimi sposobami:

```
z={ }  
z2=set()
```

Możemy też stworzyć zestaw od razu zappełniając go danymi:

```
z3={1, 3, 2, 1, 5, 1}  
print(z3)
```

Pisałem wcześniej że zestawy nie mogą zawierać duplikatów, a tymczasem dodałem kilkakrotnie tę samą wartość "1". Przy takiej deklaracji nie dostaniemy żadnego wyjątku. Po prostu "1" na liście wystąpi tylko raz. W dodatku dane będą posortowane. Sprawdźmy co zobaczymy na konsoli po uruchomieniu powyższego kodu:

```
{1, 2, 3, 5}
```

Zwróć uwagę że w przypadku zestawów używamy nawiasów klamrowych. Jak widać 1 wystąpiła tylko raz, w dodatku zestaw jest posortowany rosnąco.

A co w przypadku zestawów elementów złożonych? Sprawdźmy to na przykładzie zestawu krotek:

```
z4={ (1, "A") , (2, "B") , (1, "C") , (1, "A") }  
print(z4)
```

Co zostało wydrukowane na konsoli?

```
{(1, 'C'), (2, 'B'), (1, 'A')}
```

Czyli za duplikat zostaną uznane tylko te krotki, które mają identyczną zawartość wszystkich podelementów.

Jeśli na przykład mamy zbiór elementów pod postacią listy lub krotki, a chcielibyśmy pozbyć się z nich duplikatów to jak to zrobić? Możemy do tego celu wykorzystać właśnie zestawy:

```
lista=[1,2,3,4,3,2,1]  
zestaw=set(lista)  
lista2=list(zestaw)  
print(lista2)
```

Powyższy kod drukuje na konsoli taki wynik:

```
[1, 2, 3, 4]
```

Modyfikowanie zawartości zestawów

Do dodawania i kasowania elementów w zbiorach służą odpowiednio funkcje wbudowane "add" i "remove":

```
s={1,2,3,4}
s.add(5)
s.remove(1)
print(s)
```

Powyższy kod na konsoli drukuje:

```
{2, 3, 4, 5}
```

Funkcja remove przyjmuje przez parametr wartość która ma zostać usunięta ze zbioru. Wystąpić może tylko raz w zbiorze, więc nie musimy się zastanawiać czy usunięte zostaną wszystkie czy tylko pierwsze wystąpienie wartości :)

Podobnie jak np. w listach istnieje też funkcja clear, która kasuje całą zawartość zestawu.

Wyjątki

Dotychczas w wielu sytuacjach spotykaliśmy się z wyjątkami. Przy próbie dzielenia przez zero, przy próbie dostępu do nieistniejącego elementu słownika czy listy. W przypadku pojawienia się wyjątku program przestaje być wykonywany, dlatego warto reagować na pojawiające się wyjątki.

Obsługa wyjątków

Najpierw spróbujmy wywołać jakiś wyjątek:

```
print(1/0)
print("dalej")
```

Gdy spojrzymy na konsolę po próbie uruchomienia powyższego kodu, zobaczymy komunikat o pojawieniu wyjątku:

```
print(1/0)
```

```
ZeroDivisionError: division by zero
```

Słowo "dalej" nie zostało wypisane na konsoli, czyli program został przerwany w momencie pojawienia się wyjątku. Co jednak gdy chcielibyśmy aby w przypadku pojawienia się wyjątku został on obsługowany przez zaplanowany przez nas sposób, a program kontynuował swoje działanie? Tu z pomocą przychodzi nam konstrukcja try-except-finally-else dostępna w Pythonie.

Skorzystajmy z jej podstawowej formy w celu poprawienia powyższego programu:

```
try:
    print(1/0)
except:
    print("nie zabanglało")
print("dalej")
```

Tym razem na konsoli nie pojawił się żaden wyjątek, za to zostało wydrukowane słowo "dalej". Powiedzieliśmy Pythonowi "try" - czyli spróbuj zrobić to, a jak się nie uda "except" to zrób to. Wyjątek pojawiający się w instrukcji "print(1/0)" został przechwycony w bloku except i została wykonana zaplanowana przez nas reakcja sprowadzająca się do wyświetlenia stosownego komunikatu. Ponieważ wyjątek został obsługowany program pracował dalej i na konsoli wyświetliło

się również "dalej". Gdyby po instrukcji "print(1/0)" znalazły się jeszcze kolejne, nie zostałyby one wykonane.

Prezentowany kod będzie działał dla każdego wyjątku. Możemy także obsługiwać wskazane rodzaje wyjątków:

```
try:
    print(1/0)
except ZeroDivisionError:
    print("nie można dzielić przez zero!")
```

Tym razem obsłużyliśmy jedynie wyjątek dzielenia przez zero (ZeroDivisionError). Gdyby kod spowodował pojawienie się innego wyjątku, nie zostałby on obsłużony. Na taką ewentualność też mamy sposób:

```
try:
    print(1/0)
except ZeroDivisionError:
    print("nie można dzielić przez zero!")
except:
    print("jakiś inny błąd")
```

Inny błąd zostanie obsłużony instrukcjami pojawiającymi się po drugim wystąpieniu klauzuli "except". To byłaby taka obsługa ogólna dla wszystkich pozostałych wyjątków, którą musimy umieścić zawsze na końcu. Obsługiwać wyjątki możemy też zbiorczo dla kilku wybranych rodzajów wyjątków:

```
try:
    print(1/0)
except (ZeroDivisionError, IOError):
    print("albo to, albo to")
```

W powyższym przykładzie dla obu wyjątków obsługa sprowadzałaby się do tych samych czynności.

Możemy też zechcieć odebrać komunikat błędu (np. żeby zarejestrować go w logach), a w takim przypadku nadajemy alias:

```
try:
    print(1/0)
except ZeroDivisionError as e:
    print(e)
```

Na konsoli zostaje wyświetlone:

division by zero

Może się też tak zdarzyć że będziemy chcieli wykonać jakąś czynność niezależnie od wystąpienia lub nie wyjątku. Wtedy wykorzystujemy klauzulę finally:

```
try:
    print(1/0)
except:
    print("wyjątek")
finally:
    print("co by się nie działo to ja się uruchamiam")
```

Istnieje też możliwość zareagowania w sytuacji gdy wyjątek nie wystąpił:

```
try:
    print("info")
except ValueError:
    print("wyjątek")
else:
    print("nie było wyjątku")
```

Python umożliwia też wywoływanie wyjątków, nawet jeśli taki wyjątek nie wystąpi "naturalnie":

```
try:
    raise TypeError()
except TypeError:
    print("to było do przewidzenia...")
```


Funkcje

Funkcje przydadzą nam się wszędzie tam gdzie będziemy potrzebowali kodu wielokrotnego użytku. Funkcje mogą przyjmować parametry różnych typów, mogą też zwracać jakieś dane.

Deklarowanie funkcji

Najprostsza postać deklaracji funkcji:

```
def sayHello():  
    print("Hello my friend!")
```

Funkcja nosi nazwę "sayHello". Jej wywołanie sprowadza się do:

```
sayHello()
```

Należy tylko pamiętać że deklaracja funkcji musi znajdować się nad jej wywołaniem. Z tego powodu funkcje deklarujemy zwykle na początku pliku, lub w osobnym module (jeden z kolejnych tematów) który następnie importujemy na początku pliku.

Parametry funkcji

Parametry funkcji służą do przekazywania do niej danych.

```
def sayHello(imie):  
    print("Hello my friend {}".format(imie))
```

Powyżej przeróbka poprzedniego przykładu. Nie możemy posiadać w tym samym pliku dwóch funkcji o tej samej nazwie a różniącej się tylko liczbą parametrów. Przeciążanie tu nie funkcjonuje. Każda kolejna funkcja o takiej samej nazwie przesłania poprzednią.

Funkcje mogą przyjmować wiele parametrów, rozdzielamy je przecinkami:

```
def dodaj(x, y):  
    print(x+y)
```

Funkcje z parametrami wywołujemy tak samo jak te bez parametrów, z tym że musimy podać wartości które do parametrów mają trafić:

```
dodaj(3,5)
```

Wartości parametrów można podmieniać wewnątrz funkcji - nie są tylko do odczytu jak w niektórych językach programowania. Przykładowo poniższa funkcja zawsze będzie witała Czesława, niezależnie od tego co podamy przy wywołaniu:

```
def sayHello(imie):  
    imie="Czesław"  
    print("Hello my friend {}!".format(imie))
```

Ponieważ zadeklarowane przez nas funkcje mogą być użytkowane przez inne osoby, a te niekoniecznie będą wiedziały jaki rodzaj danych nasza funkcja obsługuje, warto znać sposób na sprawdzenie typu danych które zostają podane przez parametry:

```
def sprawdz_typ(x):  
    if( isinstance(x,int)): # sposób na sprawdzenie czy parametr  
    jest spodziewanego typu  
        print('otrzymałem liczbę całkowitą')  
    else:  
        print('otrzymałem coś innego niż liczba całkowita')
```

W Pythonie możemy również zadeklarować wartość domyślną dla parametru funkcji:

```
def domyslnne_wartosci(a="brak",b="brak"):  
    print('a='+a)  
    print('b='+b)
```

Przy wywołaniu możemy, ale nie musimy wtedy podawać wartości parametrów tej funkcji:

```
domyslne_wartosci("X", "Y")
domyslne_wartosci()
domyslne_wartosci("coś")
domyslne_wartosci(b="coś innego")
```

Wynik na konsoli:

```
a=X
b=Y
a=brak
b=brak
a=coś
b=brak
a=brak
b=coś innego
```

W pierwszym przypadku wywołanie jak dotychczas. Chciałem jedynie zaznaczyć, że fakt posiadania przez funkcję wartości domyślnych parametrów nie powoduje że nie możemy jej wywoływać tak jak we wcześniejszych przykładach.

W drugim nie podaję wartości dla parametrów, a funkcja przyjmuje wartości domyślne.

Trzeci wariant jest bardzo ciekawy - co jeśli podamy mniej wartości niż parametrów? Python przypisze je do parametrów wg. kolejności, a pozostałe przyjmą wartości domyślne - ale tylko jeśli takie wartości zostaną zadeklarowane. Bez tego dostalibyśmy wyjątek.

Czwarta opcja to przekazywanie wartości do parametrów po nazwie.

Zwracanie wyników z funkcji

Funkcje mogą zwracać wartości. Mogą to być pojedyncze liczby czy ciągi tekstowe, ale również złożone struktury jak np. tablice. Najprostsza funkcja zwracająca "0":

```
def oddaj0():
    return 0
```

Wynik z takiej funkcji możemy odebrać i przekazać do innej funkcji (np. print), albo przypisać do jakiejś zmiennej:

```
print(oddaj0())
x=oddaj0()
print(x)
```

Przykład funkcji zwracającej złożony typ danych - listę cyfr od 0 do 9:

```
def dajcyferki():  
    l=list(range(10))  
    return l
```

```
dajcyferki()
```

Wyrażenia Lambda

Funkcje możemy deklarować w locie, najczęściej do jednorazowego użytku. Wyrażenia Lambda są dużym i złożonym tematem - zwłaszcza ich zastosowanie, na ten moment w zupełności wystarczy nam się orientować co to takiego:

```
fun=lambda a,b: a+b  
print (fun(10,20)) #wykorzystanie odebranego w ten sposob ciala  
funkcji.
```

Wyrażenie Lambda zwraca ciało funkcji. To musi być konkretnie wyrażenie, nie może tu być np. print. Poniżej równoważna deklaracja zwykłej funkcji:

```
def nielambda(a,b):  
    return a+b
```

Funkcja jako argument

Funkcja może być przekazana jako argument innej funkcji:

```
def razy2(a): # funkcja która będzie użyta jako argument  
    return a*2
```

```
def funkcja_jako_argument(f,x):  
    print(f(x))
```

```
funkcja_jako_argument(razy2,33)
```

Zadeklarowałem dwie funkcje. Pierwsza przyjmuje liczbę którą zwraca podwojoną. Druga funkcja przyjmuje dowolną funkcję "f" (ze względu na wywołanie w print musi ona posiadać jeden argument) oraz liczbę która zostanie podana do funkcji "f" w ciele funkcji "funkcja_jako_argument". Na końcu mamy wywołanie funkcji "funkcja_jako_argument", której efektem działania jest wypisanie na konsoli wartości "66".

Przekazujemy funkcję bez podania nawiasów - nie chcemy wywoływać tej funkcji przy przekazaniu a przekazać funkcję jako obiekt.

Poniżej przykład połączenia przekazywania funkcji jako argument i argumentów *args. Kod poniżej spowoduje wyświetlenie każdej z wartości *args po obróbce funkcją przekazaną jako argument. Funkcją "obrabiającą" jest tu funkcja powieksz która zwraca powiększony otrzymany przez argument tekst. Warunkiem działania poniższej kodu (ze względu na wywołanie x.upper()) jest podanie samych ciągów tekstowych do args:

```
def powieksz(x):  
    return x.upper()
```

```
def zastosuj_dla_wszystkich(fun,*args):  
    for a in args:  
        print(fun(a))
```

```
zastosuj_dla_wszystkich(powieksz, 'siała', 'baba', 'mak')
```

Wynik działania na konsoli:

```
SIAŁA  
BABA  
MAK
```

Funkcje mogą być przekazywane również jako listy. Poniższy przykład prezentuje zastosowanie rzędu funkcji na jednej zmiennej.

```
def pomnoz_razy_dwa(x):  
    return x*2
```

```
def podziel_przez_trzy(x):  
    return x/3
```

```
def dodaj_piec(x):  
    return x+5
```

```
funkcje=[pomnoz_razy_dwa,podziel_przez_trzy,dodaj_piec]
```

```
def obrob(wartosc,*funkcs):  
    for f in funkcs:  
        wartosc=f(wartosc)  
    return wartosc
```

```
print (obrob(1,pomnoz_razy_dwa,podziel_przez_trzy,dodaj_piec) )
```

Zdeklarowałem trzy proste funkcje. Ich nazwy wskazują co robią z otrzymaną wartością. Stworzyłem listę referencji do obiektów funkcji - zwróć uwagę że przekazuję funkcje bez podawania nawiasów a tym bardziej argumentów - ostatecznie chodzi o przekazanie referencji do funkcji a nie jej wywołanie. Wartość przekazana jako pierwszy argument zostaje obrobiona przez wszystkie funkcje podane jako *args (tutaj *funkcs) a następnie zwrócona z funkcji "obrob" i wyświetlona na konsoli. W efekcie działania dostajemy wartość "5.666666666666667".

Funkcja w funkcji

Możliwa jest deklaracja funkcji we wnętrzu innej funkcji. Taka funkcja wewnątrz innej funkcji będzie widziana tylko w niej:

```
def zewnetrzna(x) :  
    def wewnetrzna(x) :  
        return x*2  
    print(wewnetrzna(x) )
```

```
zewnetrzna(50)
```

Zwracanie funkcji z funkcji

Tworzone przez nas funkcje mogą zwracać zawarte w nich funkcje. Ponieważ funkcje są obiektami, możemy nie tylko przyjmować je przez argumenty, ale też zwracać. Poniżej obrazujący to przykład. Wewnątrz funkcji `wybierz` zadeklarowałem dwie funkcje. Jedna powiększa otrzymany tekst, druga go pomniejsza. W zależności od liczby którą podamy jako argument dla wywołania funkcji "wybierz" zostaje zwrócona jedna albo druga funkcja. Przy wywołaniu odbieram zwracaną funkcję i stosuję ją wobec wypisywanego tekstu:

```
def wybierz(tryb):
    def powieksz(x):
        return x.upper()
    def pomniejsz(x):
        return x.lower()
    if tryb==1:
        return powieksz
    elif tryb==2:
        return pomniejsz

funkcja=wybierz(1)
print(funkcja('Witaj Świecie!'))
funkcja=wybierz(2)
print(funkcja('Witaj Świecie!'))
```

Wynik na konsoli:

WITAJ ŚWIECIE!

witaj świecie!

Jeszcze jeden przykład:

Przeanalizujemy poniższy przykład:

```
def daj_funkcje(x):  
    def podwoj(a):  
        return a*2  
    def polowa(a):  
        return a/2  
    if x==1:  
        return podwoj  
    elif x==2:  
        return polowa
```

```
fun=daj_funkcje(1)  
print ( fun(6) )
```

W wyniku uruchomienia powyższego kodu otrzymujemy na konsoli liczbę 12. Funkcja "daj_funkcje" zwraca jedną ze swoich wewnętrznych funkcji w zależności od otrzymanej przez argument wartości. Zwróć uwagę że mamy zapis "return podwoj" a nie "return podwoj()" - różnica jak zawsze tkwi w szczegółach. W pierwszym przypadku zwracamy obiekt funkcji, w drugim zwracamy wynik jej działania po wywołaniu.

Rekurencja

Rekurencja w dużym uproszczeniu sprowadza się do wywoływania funkcji przez samą siebie, aż do uzyskania określonego wyniku. Przykład z obliczaniem silni:

```
def silniaRek(n):  
    if n==0:  
        return 1  
    else:  
        return n*silniaRek(n-1)
```


Inny przykład rekurencji. Choć można wypisywanie wartości w wybranym zakresie zorganizować znacznie prościej, to tutaj użyłem do tego celu rekurencji:

```
def rekurencyjny_wypisywacz(n):  
    print(n)  
    if n>0:  
        rekurencyjny_wypisywacz(n-1)  
    else:  
        return n  
    return n
```

```
rekurencyjny_wypisywacz(10)
```

Po wywołaniu na konsoli wyświetlane są liczby w zakresie 10-0 malejąco.

*args

Wyrażenia z jedną gwiazdką (*) używamy gdy do funkcji chcemy przekazać dowolną liczbę argumentów pozycyjnych. Przyjrzyjmy się простemu przykładowi

```
def args(*args):  
    for a in args:  
        print(f'{a}')
```

Dla każdego elementu w "args" wypisujemy jego zawartość. Args jest więc iterowalne. Zauważ że deklarując argument piszę "*" przed args, ale już iterując po nim tego nie robię. Wywołanie funkcji:

```
args(1,2,3,4,5)
```

Powoduje wyświetlenie każdej z wartości w osobnej linii.

```
1  
2  
3  
4  
5
```

Zwróć uwagę, że elementy nie zostały przekazane jako lista, ale jako kolejne argumenty!
Przyjrzyjmy się jeszcze jednemu простemu przykładowi:

```
def powiekszacz(*słowa):  
    for s in słowa:  
        print(s.upper())
```

```
powiekszacz('siema', 'tu', 'mapet')
```

Po wywołaniu dostajemy na konsoli:

```
SIEMA  
TU  
MAPET
```

Nazwa *args nie jest wymagana, jest to tylko taka przyjęta konwencja. Może się nazywać choćby *słowa jak w powyższym przykładzie.

Do funkcji poza parametrami `*args` możemy przekazać jeszcze jakieś dodatkowe parametry, ale wtedy będą musiały być one wymienione jako pierwsze.

```
def dlakazdego(param1,*args):  
    print(f'param1={param1}')  
    for a in args:  
        print(a)  
  
dlakazdego('pierwsza wartość','pierwszy arg','drugi arg', 3)
```

Należy być ostrożnym przy przekazywaniu list do takiej funkcji. Weźmy pod uwagę taki przypadek:

```
def argi(*args):  
    for e in args:  
        print(e)  
  
argi([1,2,3,4], 'coś')
```

Przekazana lista zostaje potraktowana jako jeden element. Zobaczmy co zostało wyświetlone:

```
[1, 2, 3, 4]  
coś
```

Jeśli chcemy by każdy z elementów tej listy został potraktowany jako osobny argument, musimy tę listę rozpakować za pomocą operatora `"*"`:

```
argi(*[1,2,3,4], 'coś')
```

co skutkuje wyrzuceniem na konsolę:

```
1  
2  
3  
4  
coś
```

****kwargs**

Wyrażenia z dwoma gwiazdkami (**) stosujemy gdy do funkcji chcemy przekazać zestaw argumentów klucz wartość. W przeciwieństwie do *args argumentom przypiszemy nazwy, a nazwy te staną się kluczem dla wartości. Po argumentach przekazanych w ten sposób poruszamy się jak po słowniku:

```
def parametr_kwargs( **kwargs):  
    for k in kwargs:  
        print(k,kwargs[k])  
  
parametr_kwargs(dodatkowy=48, nastepny=111)
```

Wynik działania na konsoli:

dodatkowy 48

nastepny 111

"k" stanowi tutaj klucz i w kolejnych "k" będziemy mieli nazwy parametrów jakie zostały przekazane, a w kwargs[k] ich wartości. Nieco inna wizualizacja:

```
def kwargs(**parametry):  
    print(parametry)  
  
kwargs(param1=1,param2=2,param3=3)
```

powoduje wyświetlenie na konsoli:

{'param1': 1, 'param2': 2, 'param3': 3}

Jeśli zechcielibyśmy do funkcji przekazać jakieś dodatkowe poza `**kwargs` parametry to musimy wymienić je jako pierwsze:

```
def kwargs_argument(argument,**kwargs):  
    print(argument)  
    print(kwargs)
```

```
kwargs_argument(argument=10,param1='Andrzej',param2='Klusiewicz')
```

Wynik działania na konsoli:

```
10  
{'param1': 'Andrzej', 'param2': 'Klusiewicz'}
```

Przykład zastosowania `**kwargs`, funkcja która umieszcza w pliku o przekazanej nazwie dowolne parametry w formacie pliku csv:

```
def zapisz_parametry_do_pliku(nazwa_pliku,**parametry):  
    plik=open(nazwa_pliku,mode='w', encoding='utf-8')  
    for p in parametry:  
        plik.write(f'{p};{parametry[p]}\n')  
    plik.close()
```

```
zapisz_parametry_do_pliku('mojplik.csv',parametr1='wartość 1',parametr2=2)
```

Zawartość pliku 'mojplik.csv':

```
parametr1;wartość 1  
parametr2;2
```

Optymalizacja funkcji z użyciem cache funkcji

Przeanalizujemy poniższy przykład:

```
from datetime import datetime
import time

def czekacz():
    time.sleep(1)
    return 1

początek=datetime.now()
for x in range(10):
    czekacz()
koniec=datetime.now()
print(koniec-początek)
```

Mam funkcję "czekacz" której jedynym zadaniem jest poczekanie 1 sekundy i zwrócenie liczby 1. Funkcja ta jest wykonywana dziesięciokrotnie, a na koniec wypisywany jest czas realizacji całości. Kod w powyższym stanie wykonuje się niedługo ponad 10 sekund. Przyjmijmy teraz że zamiast oczekiwania 1 sekundy mamy do wykonania jakieś złożone obliczenie które trwa. Jeśli funkcja dla danego outputu daje nam zawsze ten sam wynik (jest deterministyczna) to wynik tej funkcji można umieścić w cache i ponownie jej reużyć przy ponownym wywołaniu funkcji z tą samą wartością dla niej zamiast obliczać po raz wtóry. Do tego służy moduł "functools" i zawarty w nim dekorator "lru_cache". Przeanalizujemy teraz ten przykład po pewnych zmianach. Nad funkcją czekacz wpisałem dekorator "@functools.lru_cache". Dekorator ten sprawia że wynik działania naszej funkcji czekacz ląduje w cache i jest pobierana przy kolejnych wywołaniach tej funkcji dla tych samych argumentów (w naszym przypadku brak argumentu - funkcja zawsze zwraca 1):

```
from datetime import datetime
import time
import functools
@functools.lru_cache(maxsize=None)
def czekacz():
    time.sleep(1)
    return 1

początek=datetime.now()
for x in range(10):
    czekacz()
koniec=datetime.now()
print(koniec-początek)
```

Tym razem wykonanie całości zajęło niedługo ponad 1 sekundę! Wynika to z tego, że kolejne wywołania tej funkcji odczytywały wartość z cache nie wykonując tej funkcji. Argument tego

dekoratora (`maxsize=None`) określa dla ilu wartości wejściowych funkcja ma przechowywać dane w cache.

Cache można stosować tylko do funkcji deterministycznych - czyli w skrócie takich które dla tych samych parametrów wejściowych zwrócą nam zawsze te same dane wyjściowe.

Generatory

Generator jest funkcją która może zostać wstrzymana i wznowiona od miejsca w którym została wstrzymana. Generatory cechują się leniwą ewaluacją. Tworzą kolejne elementy dopiero w momencie odwołania się do generatora. Pozwala nam to wydajniej wykorzystywać pamięć operacyjną i zwracać z generatora nieskończoną liczbę elementów

Generatory nie tworzą całej zwracanej kolekcji od razu, tylko każda kolejna wartość jest generowana w momencie jej pobrania. To duża oszczędność dla pamięci, ponieważ w dowolnym momencie możemy przerwać pobieranie kolejnych wartości. Nie moglibysmy tego zrobić gdyby funkcja zwracała listę, choć sposób iteracji byłby taki sam. W przypadku generatorów w pamięci przechowywany jest tylko jeden aktualnie pobierany element a nie cała lista, co pozwala nam przetwarzać zbiory w zasadzie nieskończenie długie, czego nie moglibysmy zrobić w przypadku zwracania listy. Lista mogłaby przepełnić dostępną pamięć.

Z zagadnień podstawowych pamiętamy taką konstrukcję:

```
for x in range(10):  
    print(x)
```

Konstrukcja ta pozwalała na iterowanie po kolejnych elementach zwracanych przez range. Istnieje możliwość tworzenia własnych mechanizmów tego typu. Przyjrzyjmy się poniższemu kodowi:

```
def elementy():  
    yield 'element numer 1'  
    yield 'element numer 2'  
    yield 'element numer 3'  
    yield 'element numer 4'
```

```
for e in elementy():  
    print(e)
```

Powyżej widzimy bardzo prosty generator. Wynik jego działania przedstawia się następująco:

```
element numer 1  
element numer 2  
element numer 3  
element numer 4
```

Moja funkcja "elementy" za pomocą słowa kluczowego "yield" podaje nam kolejne elementy. Słowo kluczowe „yield” odpowiada za przerwanie wykonania funkcji, zapisanie jej aktualnego stanu i zwrócenie kolejnej wartości. Jak widzimy w powyższym przykładzie, stan wykonania funkcji elementy

był zapamiętywany i dlatego iterując po wyniku tej funkcji dostajemy kolejne podawane przez „yield” elementy.

W podobny sposób możemy utworzyć własną adaptację range podającą nam wartości co 10:

```
def myrange(n):  
    for x in range(n):  
        yield x*10  
  
for x in myrange(10):  
    print(x)
```

Idąc tym tropem możemy budować bardziej złożone konstrukcje. Poniżej przykład generatora który podaje tyle potęg kolejnych liczb ile otrzyma przez argument:

```
def potegi2(n):  
    for x in range(1, n + 1):  
        yield pow(2, x)  
  
for p in potegi2(5):  
    print(p)
```

Wynik działania na konsoli:

```
2  
4  
8  
16  
32
```

Nie zawsze chcemy przetworzyć cały zbiór jaki generator może nam zwrócić. Weźmy pod uwagę funkcję generującą która będzie nam zwracała kolejne wartości bez końca. W poniższym przypadku będą to kolejne dziesięci:

```
def dziesieci():  
    i=1  
    while True:  
        yield i*10  
        i+=1
```

Mogę teraz pojedynczo pobierać kolejne wartości korzystając z poniższej konstrukcji:

```
dz=dziesieci()  
print( dz.__next__() )  
print( dz.__next__() )  
print( dz.__next__() )
```

Na konsoli dostaję dane:

```
10  
20  
30
```

Ten sam skutek mogę osiągnąć taką konstrukcją:

```
dz=dziesieci()  
print(next(dz))  
print(next(dz))  
print(next(dz))
```

Generator może oddawać wartości różnych typów i generować je na różne sposoby, ważne by oddawać kolejne wartości za pomocą yield:

```
def poryRoku():
    pory = ['styczeń', 'luty', 'marzec', 'kwiecień', 'maj', 'czerwiec', 'lipiec',
            'sierpień', 'wrzesień', 'październik',
            'listopad', 'grudzień']
    for e in pory:
        yield e
```

```
for p in poryRoku():
    print(p)
```

Powyżej generator zwracający nam nazwy kolejnych miesięcy. Kolejny przykład to generator kolejnych liczb parzystych:

```
def parzyste(n):
    for x in range(n + 1):
        yield 2 * x
```

```
for p in parzyste(10):
    print(p)
```

Generator kolejnych liter alfabetu:

```
def literki():
    for x in range(97,123):
        yield(chr(x))
```

```
for l in literki():
    print(l)
```

Teraz nieco bardziej praktyczny przykład. Generator który czyta plik csv, rozbija każdą z linii csv na listę wg podanego przez argument rozdzielacza. Na potrzeby tego przykładu stworzyłem plik o nazwie "plik.csv" i umieściłem w nim następujące dane:

```
1;Artur
2;Krzysztof
3;Zenon
4;Marcin
5;Andrzej
```

Dodajemy generator i wywołujemy go:

```
def rozbijacz_csv(np,r):  
    plik=open(np,encoding='utf-8')  
    while True:  
        linia=plik.readline()  
        if not linia:  
            break  
        yield linia.strip().split(r)  
  
rc=rozbijacz_csv('plik.csv',';')  
print(next(rc))  
print(next(rc))
```

Wynik na konsoli:

```
['1', 'Artur']  
['2', 'Krzysztof']
```

Zwróć uwagę że nie użyłem konstrukcji typu "for linia in plik.readlines():" tylko wczytuję kolejne linie jedna po drugiej. Robię tak dlatego, że "readlines()" wczytuje od razu całą zawartość pliku, co mogłoby skończyć się przepełnieniem pamięci w przypadku bardzo dużego pliku.

Dekoratory

Dekorator to jeden ze strukturalnych wzorców obiektowych. Pozwala on na dynamiczne dodanie nowej funkcjonalności do istniejącej klasy podczas działania programu. Zasada działania dekoratorów polega na opakowaniu oryginalnej klasy nową tak zwaną klasą dekorującą.

Aby rozpocząć definiowanie własnych dekoratorów musimy wiedzieć że:

- Funkcja może być przekazywana do innej funkcji jako parametr
- Funkcja może być zdefiniowana wewnątrz innej funkcji
- Funkcja może zwracać inną funkcję

Poniżej małe przypomnienie.

Funkcja jako argument

```
def obrob(fun,a,b):  
    print ( fun(a,b) )
```

```
def dodaj(a,b):  
    return a+b
```

```
def odejmij(a,b):  
    return a-b
```

```
obrob(dodaj,10,5)  
obrob(odejmij,10,5)
```

Wynik na konsoli:

```
15  
5
```

Zadeklarowałem funkcję "obrob" która przez argumenty przyjmie jedną funkcję oraz dwie wartości liczbowe. Funkcja "obrob" zastosuje przekazaną przez pierwszy argument funkcję na pozostałych 2 argumentach i wyświetli wynik tej operacji. Następnie deklaruję dwie proste funkcje - jedna dodaje do siebie otrzymane argumenty, druga je od siebie odejmuje. Wywołuję funkcję "obrob" podając przez argumenty wartości liczbowe do obrobienia i funkcję która ma zostać na tych wartościach zastosowana.

Funkcja w funkcji

```
def zewnetrzna():  
    def wewnetrzna(a,b):  
        return a*b  
    x=wewnetrzna(4,5)  
    return x  
  
print(zewnetrzna())
```

Wewnątrz funkcji "zewnetrzna" zadeklarowałem funkcję "wewnetrzna". Jest ona widoczna tylko z wnętrza funkcji "zewnetrzna" z linii znajdujących się pod deklaracją funkcji "wewnetrzna".

Zwracanie funkcji z funkcji

```
def zewnetrzna():  
    def wewnetrzna(a,b):  
        return a*b  
    return wewnetrzna  
  
x=zewnetrzna()  
print( x(19,13) )
```

Funkcja "zewnetrzna" zwraca zaimplementowaną w swoim wnętrzu funkcję "wewnetrzna". Obiekt funkcji zostaje przypisany do zmiennej x - która od tej pory będzie reprezentowała przypisaną funkcję. Następnie wypisuję wynik działania odebranej funkcji na wartościach 19 i 13.

Tworzenie dekoratorów

Dekorator to funkcja która przyjmuje przez argument dekorowaną funkcję, tworzy wewnętrzną funkcję która nadpisuje działanie funkcji dekorowanej a następnie zwraca tą funkcję wewnętrzną. Poniżej przykład:

```
def doopakowania():  
    print('do opakowania')
```

```
def dekorator(fun):  
    def opakowujaca():  
        print('opakowująca')  
        fun()  
    return opakowujaca
```

```
dek=dekorator(dopakowania)  
dek()
```

Wynik działania powyższego kodu na konsoli:

```
opakowująca  
do opakowania
```

Zwróć uwagę że funkcja "dekorator" zwraca referencję do obiektu funkcji a nie jej wywołanie - stąd brak nawiasów przy "return opakowujaca". W powyższym przypadku funkcja "opakowujaca" dodaje dodatkowe zadanie wydruku informacji na konsoli. Ten sam efekt możemy osiągnąć również w ten sposób:

```
def dekorator(fun):  
    def wewnetrzna():  
        print('jakieś dodatkowe działania')  
        fun()  
    return wewnetrzna
```

```
@dekorator  
def funkcja():  
    print('jestem funkcją')
```

```
funkcja()
```

Użyłem tutaj zapisu "@dekorator". To co znajduje się po znaku "@" to nazwa funkcji dekorującej. Zwróć uwagę że zmieniłem kolejność deklaracji funkcji. Dekorator musi zostać zdefiniowany przed jego użyciem.

Dekorowanie funkcji z parametrami

Dotychczas dekorowaliśmy funkcje nie przyjmujące żadnych argumentów. Co jednak jeśli takie się pojawiają? Mamy dwie możliwości. Możemy posłużyć się zwykłym parametrem lub *args czy **kwargs. Najpierw pierwszy wariant:

```
def dekorowana(x):  
    print(f'siema {x}')
```

```
def dekorator(fun):  
    def wewn(x):  
        print('przed')  
        fun(x)  
        print('po')  
    return wewn
```

```
d=dekorator(dekorowana)  
d('Andrzej')
```

Na konsoli zostało wyrzucone:

przed

siema Andrzej

po

Alternatywna technika dająca ten sam wynik działania:

```
def dekorator(fun):  
    def wewn(x):  
        print('przed')  
        fun(x)  
        print('po')  
    return wewn
```

@dekorator

```
def dekorowana(x):  
    print(f'siema {x}')
```

```
dekorowana('Andrzej')
```


Drugi wariant przekazywania argumentów. Nasz dekorator poza funkcją może przyjmować argumenty również jako `*args` lub `**kwargs` które następnie przekazuje do dekorowanej funkcji. Musimy pamiętać by `*args` czy `**kwargs` były podane po funkcji:

```
def funkcja(imie):
    print(f'hello {imie}!')
```

```
def dekorator(fun,*args):
    def wewn():
        print('dekoracja')
        fun(*args)
    return wewn
```

```
f=dekorator(funkcja,'Andrzej')
f()
```

Wynik działania na konsoli:

```
dekoracja
hello Andrzej!
```

Alternatywny sposób tworzenia dekoratora:

```
def dekorator(fun,*args):
    def wewn(*args):
        print('dekoracja')
        fun(*args)
    return wewn
```

```
@dekorator
def funkcja(imie):
    print(f'hello {imie}!')
```

Dokumentowanie funkcji

Jeśli zamierzamy funkcję opublikować dla innych programistów, albo jeśli zamierzamy używać naszej funkcji w przyszłych programach warto dodać do niej dokumentację by było wiadomo jak z tej funkcji korzystać. Sprowadza się to do umieszczenia tekstu pomiędzy znacznikami `"""` pod nagłówkiem funkcji:

```
def zdokumentacja():  
    """to jest dokumentacja tej funkcji"""  
    pass
```

Aby uzyskać dostęp do takiej dokumentacji, możemy użyć jednego z poniższych wywołań:

```
help(zdokumentacja)  
  
print(zdokumentacja.__doc__)
```

Moduły

Moduły to po prostu pliki zawierające funkcje. Stosuje się je po to by stworzyć sobie biblioteki narzędzi i odseparować ich kod od kodu właściwego programu. Moduły można też dystrybuować by wykorzystywać je w innych aplikacjach.

Definiowanie modułów

Stwórzmy dwa nowe pliki :

- paczka.py
- paczka2.py

W pierwszym umieszczamy kod:

```
def hello() :  
    '''Ta funkcja wypisuje na ekranie tekst hello world!'''  
    print("hello world!")
```

W drugim :

```
def sayno() :  
    print('NO!')
```

```
def sayyes() :  
    print('YES')
```

```
def add(a,b) :  
    return a+b
```

Tworzymy teraz trzeci plik, w którym będzie nasz "właściwy program" i na początek umieszczamy w nim taką instrukcję:

```
import paczka
```

Taki zapis oznacza "zassanie" zawartości modułu "paczka" do naszego pliku w taki sposób, że funkcje umieszczone w tym importowanym module stają się widoczne w naszym programie. Możemy teraz wywołać dowolną publiczną funkcję z zaimportowanego modułu w ten sposób:

```
paczka.hello()
```

Gdyby moduł paczka miał dłuższą nazwę, konieczność podawania jej przy każdym wywołaniu funkcji w niej zawartej byłoby conajmniej niehumanitarna ;). Jeżeli chcemy skrócić wywołanie, możemy przy okazji importu modułu nadać mu lokalny alias:

```
import paczka as p
```

dzięki czemu funkcje z tego modułu będzie można wywoływać teraz w ten sposób:

```
p.hello()
```

Istnieje też możliwość importu pojedynczych funkcji z modułu (ten może być przecież bardzo duży i zawierać wiele niepotrzebnych nam, a podlegających parsowaniu funkcji). Dzięki temu będziemy mogli przy okazji wywoływać zaimportowane funkcje bez poprzedzania ich nazw nazwą modułu czy jego aliasem.

```
from paczka2 import add, sayno  
sayno()
```

Dokumentowanie modułów i sprawdzanie dostępnych funkcji

Aby sprawdzić dostępne w module funkcje, możemy posłużyć się:

```
print(dir(paczka))
```

Wyświetli nam się lista na konsoli:

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__',  
 '__spec__', 'hello']
```

Jeśli chcemy, możemy teraz sprawdzić dokumentację jakiejś wybranej funkcji:

```
help(moduly_pomocnicze.hello)
```

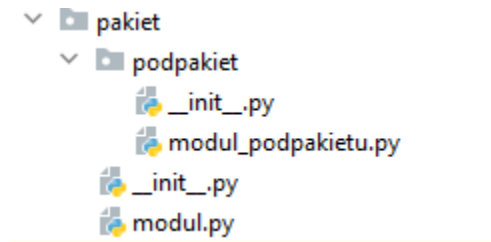
Zobaczymy wtedy to co zadeklarowaliśmy jako dokumentację danej funkcji:

```
hello()  
Ta funkcja wypisuje na ekranie tekst hello world!
```

Pakiety

Pakiet to po prostu folder zawierający moduły bądź inne pakiety. Aby było wiadomo że dany folder jest pakietem Pythona powinien on zawierać plik "__init__.py". Plik ten jest uruchamiany automatycznie gdy importujesz coś z danego pakietu. Jeśli go jednak nie utworzysz, nic złego się nie stanie.

Utworzyłem sobie w katalogu głównym projektu układ o takiej strukturze:



Wewnątrz modułu "modul.py" mam umieszczoną jedną funkcję:

```
def witacz():
    print('hi')
```

Wywołanie tej funkcji wygląda tak:

```
import pakiet.modul as pm
pm.witacz()
```

Identyczną funkcję "witacz" umieściłem w module "modul_podpakietu" pakietu "podpakiet". Tym razem wywołanie będzie wyglądało w ten sposób:

```
import pakiet.podpakiet.modul_podpakietu as mp
mp.witacz()
```

Korzystanie z plików tekstowych

Czytanie z plików tekstowych

Najprostsza forma otwarcia pliku do odczytu:

```
plik = open('dane.txt')
```

W parametrze funkcji "open" podajemy ścieżkę względną lub bezwzględną do pliku. Jeśli podajemy względną- to w odniesieniu do położenia naszego pliku z kodem. Jeśli w pliku pojawiają się jakiegolwiek znaki spoza alfabetu łacińskiego, to bardzo możliwe że zobaczysz zamiast nich "krzaczki", a wynika to z domyślnego kodowania. Możemy sprawdzić jakie mamy domyślne kodowanie w ten sposób:

```
import locale  
print(locale.getpreferredencoding())
```

Problem ten możemy rozwiązać stosunkowo łatwo, ponieważ funkcja open przyjmuje także argument dotyczący kodowania jakiego będziemy używać do odczytu pliku:

```
plik=open('dane.txt', encoding='utf-8')
```

Do katalogu w którym znajduje się nasz plik z kodem wrzucam plik "dane.txt" o zawartości:

```
linia 0  
linia 1  
linia 2  
linia 3  
linia 4  
śczęǫż
```

Ostatnią linijkę dodałem w celu sprawdzania poprawności odczytu polskich znaków diakrytycznych. Spróbujmy teraz odczytać plik. Jest na to kilka sposobów w zależności od tego w jakim formacie chcemy mieć zwrócone dane. Sposób odczytu pliku zależy od tego w jaki sposób chcesz go przetwarzać. Generalnie możesz użyć następujących funkcji:

- `read()`
- `readlines()`
- `readline()`

read()

Odczytuje plik jako jeden ciągły tekst typu "str":

```
pl=open('dane.txt', encoding='utf-8')
linie=pl.read()
plik.close()
print(linie)
print(type(linie))
```

Pojawia nam się tu jeszcze jedna nowa rzecz - zamykanie pliku funkcją "close". Program zadziała nawet jeśli pliku nie zamkniesz po odczycie, ale lepiej to robić by zwalniać zasoby. Wynik na konsoli:

linia 0

linia 1

linia 2

linia 3

linia 4

ściąćóź

<class 'str'>

Na końcu wyświetliłem też typ odczytanych danych - dla porównania, nie jest to zawartość pliku. Jak widzimy jest to typ "str", czyli po prostu jedna ciągła zmienna tekstowa. Sposób wykorzystania - przydaje się np. gdy chcesz przeszukać cały plik wyrażeniami regularnymi (które są przedmiotem jednego z kolejnych rozdziałów). Pamiętaj że z racji odczytu pliku jako całości, może okazać się że dojdzie do wycieku pamięci z racji jego wielkości. Przyda Ci się więc umiejętność sprawdzenia

"Python na luzie" v. 1.6 - Andrzej Klusiewicz - www.jsystems.pl 88/269

wielkości pliku. Będziesz mógł wtedy podjąć decyzję o tym czy chcesz wczytywać plik jako całość, czy robić to linia po linii.

Aby sprawdzić wielkość pliku, możesz zastosować poniższą konstrukcję:

```
import os
ile=os.path.getsize('dane.txt')
print(ile)
```

Do zmiennej "ile" zostanie przypisana wielkość pliku w bajtach. Jest jeszcze jedna alternatywa. Możesz użyć parametru funkcji "read" określającego ile znaków ma zostać wczytane:

```
plik=open('dane.txt',encoding='utf-8')
linie=plik.read(20)
print(linie)
print("-----")
linie=plik.read(20)
print(linie)
```

Wynik na konsoli:

linia 0

linia 1

lini

a 2

linia 3

linia 4

readlines()

Podobnie jak funkcja "read" odczytuje całą zawartość pliku na raz. Różnica tkwi w zwracanym typie, ponieważ tym razem dostaniemy listę - każda linijka w pliku znajdzie się w kolejnym elemencie listy.

```
pl=open('dane.txt', encoding='utf-8')
linie=pl.readlines()
pl.close()
print(linie)
print(type(linie))
```

Wynik z konsoli:

```
['linia 0\n', 'linia 1\n', 'linia 2\n', 'linia 3\n', 'linia 4\n', 'śczęgół']
<class 'list'>
```

Znacznik "\n" to znacznik końca linii. Możesz się go pozbyć używając funkcji "split" na każdym z elementów listy, lub zastosować nieco zmodyfikowany sposób korzystania z funkcji "read":

```
pl=open('dane.txt', encoding='utf-8')
linie=pl.read().splitlines()
pl.close()
print(linie)
print(type(linie))
```

Skutek podobny, ponieważ w obu przypadkach czytamy cały plik a w wyniku dostajemy listę pozbawioną znaczników końca linii:

```
['linia 0', 'linia 1', 'linia 2', 'linia 3', 'linia 4', 'śczęgół']
<class 'list'>
```

readline()

Służy do odczytu pliku linia po linii. Przydatne na przykład gdy chcesz przeszukiwać plik aż do znalezienia jakiegoś fragmentu tekstu. Nie zawsze jest potrzebne czytanie całego (często bardzo dużego pliku). Poniżej przykład konstrukcji pozwalającej czytać plik linia po linii:

```
with open('dane.txt',encoding='utf-8') as f:
    for l in f:
        print(l)
f.close()
```

Funkcja seek()

Funkcja "seek" przesuwa kursor do wskazanej pozycji - numer znaku w pliku. Jeśli nie podasz parametru, funkcja domyślnie przyjmie wartość "0". Przypuśćmy że otworzyliśmy plik i odczytaliśmy z niego dane, po czym ponownie zechcemy odczytać jego zawartość. Poniższy kod

```
plik=open('dane.txt',encoding='utf-8')
linie=plik.readlines()
print(linie)
linie=plik.readlines()
print(linie)
plik.close()
```

drukuje na konsoli taki wynik:

```
['linia 0\n', 'linia 1\n', 'linia 2\n', 'linia 3\n', 'linia 4\n', 'śczęgół']
```

```
[]
```

Jak widzisz ponowny odczyt funkcją "readlines" zwraca nam pustą listę. Wynika to z faktu, że po pierwszym odczycie kursor został przesunięty na koniec pliku, więc kolejne wywołanie "readlines()" nie ma już co czytać.

Wykorzystując funkcję "seek" możemy ponownie przesunąć kursor na początek pliku:

```
plik=open('dane.txt',encoding='utf-8')
linie=plik.readlines()
print(linie)
plik.seek(0)
linie=plik.readlines()
print(linie)
plik.close()
```

Tym razem na konsoli dostajemy taki wynik:

```
['linia 0\n', 'linia 1\n', 'linia 2\n', 'linia 3\n', 'linia 4\n', 'ścżąęóź']
['linia 0\n', 'linia 1\n', 'linia 2\n', 'linia 3\n', 'linia 4\n', 'ścżąęóź']
```

Sprawdzanie ilości linii w pliku

Aby sprawdzić ilość wierszy w pliku musimy go niestety w całości przeczytać. Nie jest to najwydajniejsze, ale niestety nie ma alternatywy. Możemy to zrobić np. w ten sposób:

```
plik=open('dane.txt',encoding='utf-8')
linie=plik.readlines()
print('liczba wierszy w pliku={}'.format(len(linie)))
plik.close()
```

Powyższy sposób sprowadza się do odczytania wszystkich linii z pliku jako listy i sprawdzenia ilości elementów na liście.

Zapis w plikach tekstowych

Tryby otwarcia pliku

Aby zapisywać do pliku musimy w funkcji open podać wartość dodatkowego parametru "mode". W zależności od tego czy chcemy nadpisać ewentualnie istniejącą zawartość, czy dopisywać, czy może jednocześnie móc czytać i pisać do tego samego pliku, używamy jednej z poniższych instrukcji. Jeśli chcemy nadpisać ewentualną zawartość stosujemy przełącznik "w":

```
plik=open('nowy.txt',encoding='utf-8',mode='w')
```

aby dopisywać używamy "a":

```
plik=open('nowy.txt',encoding='utf-8',mode='a')
```

Oba powyższe tryby powodują stworzenie pliku jeśli taki by nie istniał. Jeśli chcemy jednocześnie pisać i czytać używamy "r+":

```
plik=open('nowy.txt',encoding='utf-8',mode='r+')
```

Wprowadzanie danych do pliku

Najprostszy sposób pisania do plików:

```
plik=open('nowy.txt',encoding='utf-8',mode='w')
for x in range(10):
    plik.write(str(x))
plik.close()
```

Tym sposobem do pliku zapiszemy ciąg "0123456789". Każda kolejna wartość znajdzie się w tej samej linii.

Jeśli chcesz zapisać kolejne wartości w kolejnych liniach, wystarczy dodać znacznik "\n":

```
plik=open('nowy.txt',encoding='utf-8',mode='w')
for x in range(10):
    plik.write(str(x)+"\n")
plik.close()
```

Python umożliwia też zapis do pliku od razu całej listy elementów. Obrazuje to poniższy przykład:

```
plik=open('nowy.txt',encoding='utf-8',mode='w')
linie=[]
for x in range(10):
    linie.append("linia numer {} \n".format(x))
plik.writelines(linie) # to nas interesuje
plik.close()
```

Najpierw przygotuję sobie listę wartości które mają zostać zapisane (umieszczając w każdej znacznik "\n" by wartości te znajdowały się w kolejnych liniach. Nie musi to być konkretnie lista, może to być dowolny iterowalny element zawierający ciągi tekstowe.

Przetwarzanie JSON

JSON to obecnie bardzo popularny format danych, wciąż zyskujący na popularności. Zyskuje kosztem formatu XML który jest coraz częściej zastępowany przez JSON. Wykorzystywany jest do przechowywania danych, ale również do komunikacji między aplikacjami jako format uniwersalny. Niezależnie od tego skąd dane będą pochodzić, czy będą pobierane z pliku czy np. z usługi sieciowej, sposób jego przetwarzania w Pythonie jest taki sam. Poniżej przedstawiam dane które będę wykorzystywał w przykładach tego rozdziału. Dane w przykładach zawsze będą znajdowały się w pliku dane.json

```
dane={
    "imie": "Andrzej",
    "nazwisko": "Klusiewicz",
    "adres": {
        "miasto": "Warszawa",
        "kod": "02-019"
    },
    "jezyki": ["polski", "angielski", "Java", "R", "Python", "PL/SQL"]
}
```

Ładowanie danych JSON z pliku

W przykładach będę używał danych umieszczonych w pliku. Do wygodnej zabawy z tym formatem użyję modułu "json". Natywne metody odczytu pliku nie byłyby zbyt użyteczne przy tym formacie danych. Moduł "json" posiada funkcję "load", która przyjmuje zmienną reprezentującą otwarty plik, a zwraca dane w postaci słownika.

```
import json
plik=open('dane.json',encoding='utf-8')
obj=json.load(plik)
print(obj)
print(type(obj))
plik.close()
```

Po uruchomieniu powyższego kodu na konsoli zobaczymy:

```
{'imie': 'Andrzej', 'nazwisko': 'Klusiewicz', 'adres': {'miasto': 'Warszawa', 'kod': '02-019'},
'jezyki': ['polski', 'angielski', 'Java', 'R', 'Python', 'PL/SQL']}
<class 'dict'>
```

Jak widzimy po informacji zwracanej przez funkcję "type", jest to zwyczajny słownik, taki z jakim już się wcześniej spotkaliśmy. Skoro tak, to dane możemy teraz przetwarzać tak samo jak każdy inny słownik. Sięgnijmy zatem do wartości elementu znajdującego się pod kluczem "imie":

```
print(obj['imie'])
```

Na konsoli zobaczymy:

Andrzej

Możemy znanymi nam już metodami przeiterować i wyświetlić np wszystkie klucze:

```
for k in obj.keys():  
    print(k)
```

Wynik na konsoli:

imie

nazwisko

adres

jezyki

Wszystkie poznane w rozdziale o słownikach metody przetwarzania tego formatu danych mają tu zastosowanie. Co jednak w przypadku bardziej złożonych struktur? Przypomnijmy zawartość pliku:

```
{  
    "imie": "Andrzej",  
    "nazwisko": "Klusiewicz",  
    "adres": {  
        "miasto": "Warszawa",  
        "kod": "02-019"  
    },  
    "jezyki": ["polski", "angielski", "Java", "R", "Python", "PL/SQL"]  
}
```


Wartością klucza "adres" jest struktura złożona. Czyli wartość dla klucza "adres" to kolejny słownik. Chciałbym teraz wydłubać nazwę miasta:

```
print( obj['adres']['miasto'] )
```

Czyż Python nie jest piękny? A co jeśli chciałbym przeiterować po wszystkich językach i je wyświetlić na konsoli?

```
for j in obj['jezyki']:
    print(j)
```

Wynik na konsoli:

```
polski
angielski
Java
R
Python
PL/SQL
```

Mogłem tak zrobić, ponieważ zbiór "języki" jest zwracany jako lista. Skoro tak, to również mogę odwoływać się do poszczególnych języków po ich pozycji na liście:

```
print(obj['jezyki'][2])
```

Zwróci nam element o indeksie 2 (czyli pozycji trzeciej) z listy języków tj. "Java".

Tworzenie i zapisywanie danych JSON do pliku

Skoro już wiemy że dane odczytywane jako JSON są zwracane pod postacią pythonowego słownika, to naturalną konsekwencją zapis będzie się odbywał dokładnie odwrotnie tj tworzymy słownik i zrzucamy do pliku. Możemy to zrobić na dwa sposoby. Wybór metody to kwestia wygody użycia, co kto lubi jak komu wygodnie bo efekt jest ten sam. Sposób pierwszy:

```
import json
obj=dict()
obj['ksiazka']='Finansowy Ninja'
obj['film_na_wieczor']='https://www.youtube.com/watch?v=sCNRK-n68CM'
obj['banknoty']=[10,20,50,100,200,500]
plik=open('jsonout.json','encoding='utf-8',mode="w")
json.dump(obj,plik)
plik.close()
```

Tworzę pythonowy słownik klucz po kluczu i przypisuję do każdego klucza wartości. Funkcja "dump" z modułu "json" zapisuje słownik w postaci formatu JSON do pliku podanego przez zmienną do drugiego parametru.

Alternatywnie mogę po prostu ręcznie zadeklarować sobie słownik i zainicjalizować go danymi pisanymi jako całość:

```
import json
dane={
    "ksiazka": "Finansowy Ninja",
    "film_na_wieczor": "https://www.youtube.com/watch?v=sCNRK-n68CM",
    "banknoty": [10,20,50,100,200,500]
}
plik=open('jsonout2.json','encoding='utf-8',mode="w")
json.dump(dane,plik)
plik.close()
```

W obu przypadkach zawartość pliku wyjściowego jest taka sama:

```
{"ksiazka": "Finansowy Ninja", "film_na_wieczor":
"https://www.youtube.com/watch?v=sCNRK-n68CM", "banknoty": [10, 20,
50, 100, 200, 500]}
```

Przetwarzanie XML

Modułów służących do przetwarzania XML w Pythonie jest bardzo wiele, prezentuję tutaj taki z jakiego sam korzystam. Nawet jeśli zamierzasz korzystać z innego, możesz potraktować ten rozdział jako przykład.

W tym samym katalogu co kod który będę uruchamiać w ramach przykładów umieszczam plik o nazwie "dane.xml", a oto jego zawartość:

```
<?xml version="1.0" encoding="UTF-8"?>
<dane atrybut="jakaś wartość">
<imie>Andrzej</imie>
<nazwisko param="wartość przykładowa" param2="kolejna wartość
przykładowa">Klusiewicz</nazwisko>
<wzrost>176cm</wzrost>
<adres>
    <miasto>Warszawa</miasto>
    <kod>02-019</kod>
</adres>
<jezyki>polski</jezyki>
<jezyki>angielski</jezyki>
<jezyki>Java</jezyki>
<jezyki>R</jezyki>
<jezyki>Python</jezyki>
<jezyki>PL/SQL</jezyki>
</dane>
```

Są to te same dane których używałem w rozdziale o przetwarzaniu JSON - tyle że tu w formacie XML.

Odczyt danych z pliku XML i sięganie do elementu po nazwie

Zacniemy od zaimportowania odpowiedniej biblioteki i parsowania pliku XML.

```
import xml.etree.ElementTree as et
drzewo=et.parse('dane.xml')
```

Funkcja `parse` powoduje odczyt wskazanego pliku w całości. Tą samą funkcją można też przeładować dane, gdyby na przykład zostały zmodyfikowane a chcielibyśmy zobaczyć zmiany. Zmienna `drzewo` jest obiektem specjalnej klasy opakowującej, tym razem nie będzie to żadna z omawianych dotychczas struktur danych, zapewne dla tego że w Pythonie nie ma właściwego odpowiednika strukturalnego. Możesz sprawdzić jaki to typ wywołując:

```
print(type(drzewo))
```

Na konsoli zostanie wyrzucony typ:

```
<class 'xml.etree.ElementTree.ElementTree'>
```

Jest to klasa zdefiniowana w tej samej bibliotece co narzędzia których będziemy używać. W związku z tym będziemy musieli używać specjalnych funkcji (również wbudowanych w tę samą bibliotekę) służących do przetwarzania obiektu XML.

Format XML wymaga by struktura danych posiadała korzeń - a więc jeden element w którym zawarte są wszystkie pozostałe. Taki element oczywiście w rzeczonym przykładzie występuje, a jest to element który w pliku jest reprezentowany parą tagów:

```
<dane atrybut="jakaś wartość">
....
</dane>
```

Aby uzyskać do niego dostęp wykorzystamy funkcję `"getroot"` która zwraca nam handler do korzenia:

```
root=drzewo.getroot()
```

W tej chwili możemy już poruszać się po drzewie XML. Dla przykładu jeśli zechcemy odczytać zawartość pola imię które w pliku znajduje się tu:

```
<?xml version="1.0" encoding="UTF-8"?>
<dane atrybut="jakaś wartość">
<imie>Andrzej</imie>
```

zastosujemy konstrukcję jak poniżej:

```
imie=root.find('imie')
```

Do zmiennej "imie" nie zostanie jednak przypisana wartość z elementu, czego pewnie można by się spodziewać, a obiekt klasy Element. Jest tak, ponieważ każdy z elementów może mieć jeszcze podelementy do których także będziemy uzyskiwać dostęp. Możemy to sprawdzić drukując samą zmienną "imie", oraz jej typ:

```
print(imie)
print(type(imie))
```

Na konsoli zobaczymy:

```
<Element 'imie' at 0x000002BBEC6CAA98>
<class 'xml.etree.ElementTree.Element'>
```

W związku z tym, będziemy potrzebowali wykorzystać atrybut "text" tej klasy do pobrania właściwej wartości:

```
print(imie.text)
print(type(imie.text))
```

co dopiero zwróci nam oczekiwane przez nas dane. Zrzut z konsoli jak zwykle:

```
Andrzej
<class 'str'>
```

Ok, mamy rozebrany proces na części pierwsze. Złożmy teraz pacjenta do kupy. Całość dotychczasowego kodu, od otwarcia pliku do wydłubania danych z elementu "imie":

```
import xml.etree.ElementTree as et
drzewo=et.parse('dane.xml')
root=drzewo.getroot()
imie=root.find('imie').text
print(imie)
```

Sięganie po podelementy

Korzystając z funkcji "find" wydobywaliśmy obiekt "imie" klasy "Element" z obiektu "root". Tak się składa, że obiekt "root" także jest klasy "Element", a z tego płynie wniosek że wydobywanie podelementów wyglądać będzie tak samo dla dowolnego obiektu tej klasy. Sprawdźmy:

```
import xml.etree.ElementTree as et
tree=et.parse('dane.xml')
root=tree.getroot()
adres=root.find('adres')
miasto=adres.find('miasto')
print(miasto.text)
```

Zrób uwagę że z obiektu "adres" za pomocą funkcji "find" wydobywam podelement "miasto" w taki sam sposób w jaki wcześniej wydobywałem obiekt "imie" z obiektu "root". Po uruchomieniu tego kodu, na konsoli zobaczymy "Warszawa".

Sięganie do elementu po pozycji

Podobnie jak mogę odnajdywać elementy po nazwie, tak mogę i po pozycji.

```
import xml.etree.ElementTree as et
drzewko=et.parse('dane.xml')
korzonek=drzewko.getroot()
drugi=korzonek[1].text
print(drugi)
```

W ten sposób odwołam się do elementu "nazwisko" będącego drugim elementem w pliku (czyli mającemu indeks 1 - liczenie od zera). Co jednak jeśli zechcielibyśmy odwołać się do elementu "kod" zagnieżdżonego w elemencie "adres"? Adres ma indeks 3 (czwarta pozycja), a "kod" ma indeks 1 (druga pozycja) wewnątrz elementu "adres", przypomina więc to listę zagnieżdżoną w liście... Skoro tak, to możemy do "kodu" dobrać się tak:

```
import xml.etree.ElementTree as et
drzewko=et.parse('dane.xml')
korzonek=drzewko.getroot()
zag=korzonek[3][1].text
print(zag)
```

Listy wartości w XML i odwoływanie się do "n-tego" wystąpienia tagu

W naszym pliku XML mamy jeszcze takie wartości:

```
...
...
</adres>
<jezyki>polski</jezyki>
<jezyki>angielski</jezyki>
<jezyki>Java</jezyki>
<jezyki>R</jezyki>
<jezyki>Python</jezyki>
<jezyki>PL/SQL</jezyki>
</dane>
```

Chcielibyśmy odnaleźć wszystkie elementy znajdujące się pomiędzy tagami <jezyki> i </jezyki>. Nic prostszego:

```
import xml.etree.ElementTree as et
d=et.parse("dane.xml")
root=d.getroot()
for e in root.findall('jezyki'): # tylko po elementach "jezyki"
    print(e.text)
```

Analogicznie do funkcji "find", jest też funkcja "findall" odnajdująca wszystkie elementy o określonym tagu. Funkcja ta zwraca nam zwyczajną listę, po której możemy iterować. Skoro jest to lista, to rozwiązuje nam to również problem typu "a jak się dostać do 2 wystąpienia elementu xyz?"

```
import xml.etree.ElementTree as et
d=et.parse("dane.xml")
root=d.getroot()
print (root.findall('jezyki')[1].text)
```


Atrybuty

Atrybuty występują w naszym pliku z danymi w dwóch miejscach. Podświetliłem je na żółto w przykładzie poniżej.

```
<?xml version="1.0" encoding="UTF-8"?>
<dane atrybut="jakaś wartość">
<imie>Andrzej</imie>
<nazwisko param="wartość przykładowa" param2="kolejna wartość
przykładowa">Klusiewicz</nazwisko>
<wzrost>176cm</wzrost>
<adres>
    <miasto>Warszawa</miasto>
    <kod>02-019</kod>
</adres>
<jezyki>polski</jezyki>
<jezyki>angielski</jezyki>
<jezyki>Java</jezyki>
<jezyki>R</jezyki>
<jezyki>Python</jezyki>
<jezyki>PL/SQL</jezyki>
</dane>
```

Chciałbym się teraz do nich dostać. Podobnie jak mogę na elemencie wywołać "text" by dostać jego zawartość, tak mogę wywołać również "attrib" dostając w zamian wszystkie atrybuty w postaci słownika:

```
import xml.etree.ElementTree as et
tree=et.parse("dane.xml")
root=tree.getroot()
nazwisko=root.find('nazwisko')
print(nazwisko.attrib)
```

Wynik działania:

```
{'param': 'wartość przykładowa', 'param2': 'kolejna wartość przykładowa'}
```

Skoro to słownik, to chcąc wybrać zawartość atrybutu "param", mogę posłużyć się notacją znaną nam już ze słowników i wykorzystać nazwę atrybutu jako klucz słownika (bo tak to jest przechowywane jak widać powyżej):

```
import xml.etree.ElementTree as et
tree=et.parse("dane.xml")
root=tree.getroot()
nazwisko=root.find('nazwisko')
print(nazwisko.attrib['param'])
```

Wynik działania:

wartość przykładowa

Ewentualnie to samo ale w krótszym zapisie:

```
import xml.etree.ElementTree as et
print(et.parse("dane.xml").getroot().find("nazwisko").attrib['param'])
```

Użyteczne "sztuczki"

Odczytywanie XML jako zwykły tekst

Gdybyśmy zechcieli odczytać zawartość pliku jako zwykły tekst, moglibyśmy oczywiście odczytać plik XML jako zwykły plik tekstowy. Nie zawsze jednak dane XML będą pochodzić z pliku, mogą być np. pobrane z jakiejś usługi sieciowej. Wydrukowanie xml na konsolę w ten sposób:

```
import xml.etree.ElementTree as et
drzewko= et.parse("dane.xml")
korzen = drzewko.getroot()
print(korzen)
```

wyświetli nam informacje o obiekcie, a nie zawartość tekstową:

```
<Element 'dane' at 0x000002A21895A9A8>
```

Moduł ElementTree posiada metodę tostring która pozwoli nam na odczyt XML jako tekst:

```
import xml.etree.ElementTree as et
drzewko= et.parse("dane.xml")
korzen = drzewko.getroot()
print(et.tostring(korzen))
```

Sprawdzanie nazwy elementu

Poniżej pokazuję przykład wyświetlenia nazwy, atrybutów i zawartości wszystkich elementów na pierwszym poziomie zagnieżdżenia. Chodzi mi zasadniczo o wywołanie "e.tag" które jest tu nowością, a służy do pobierania nazwy elementu właśnie. Przykład jednak uznałem za użyteczny i dlatego zamieszczam go w takiej formie:

```
import xml.etree.ElementTree as et
r=et.parse("dane.xml").getroot()
for e in r:
    print(e.tag, e.attrib, e.text)
```

Wynik działania:

imie {} Andrzej

nazwisko {'param': 'wartość przykładowa', 'param2': 'kolejna wartość przykładowa'}
Klusiewicz

wzrost {} 176cm

adres {}

jezyki {} polski

jezyki {} angielski

jezyki {} Java

jezyki {} R

jezyki {} Python

jezyki {} PL/SQL

Modyfikowanie drzewa XML

Modyfikowanie zawartości element

Podobnie jak używaliśmy funkcji find do odnalezienia elementu w celu jego odczytania, tak możemy jej użyć w celu zmiany zawartości. Tym razem zamiast wykorzystywać "text" do odczytu zawartości elementu, używamy go do podstawienia nowej wartości:

```
import xml.etree.ElementTree as et

r=et.parse("dane.xml").getroot()
for e in r:
    print(e.tag,e.text)

r.find('nazwisko').text="Klusiewicz po zmianie"

print("-----")
for e in r:
    print(e.tag,e.text)
```

Dodawanie i modyfikowanie atrybutów element

Dodawanie i modyfikowanie atrybutów elementów odbywa się za pomocą "attrib" tak samo jak jego pobieranie. Jeśli atrybut o podanej w nawiasach kwadratowych już istnieje dla podanego elementu, zostanie nadpisany, jeśli nie to zostanie dodany:

```
import xml.etree.ElementTree as et

r=et.parse("dane.xml").getroot()
for e in r:
    print(e.tag,e.text,e.attrib)

r.find('nazwisko').attrib['encoding']="utf-8"

print("-----")
for e in r:
    print(e.tag,e.text,e.attrib)
```

Tworzenie nowych elementów

Do tworzenia podelementów używamy "SubElement" z modułu ElementTree. Jako jego parametry podajemy element który ma się stać rodzicem nowo dodawanego elementu (w tym przypadku korzeń drzewa - czyli nowy element będzie równoległy do elementów nazwisko, imię etc), oraz nazwę element. Później dodajemy wartość elementu, tak samo jak robiliśmy to w istniejących elementach:

```
import xml.etree.ElementTree as et

r=et.parse("dane.xml").getroot()
for e in r:
    print(e.tag,e.text,e.attrib)

nowy=et.SubElement(r,"masa")
nowy.text=78

print("-----")
for e in r:
    print(e.tag,e.text,e.attrib)
```

Taki nowy podelement zostanie dodany na końcu. Gdybyśmy zechcieli dodać go w wybranym przez nas miejscu, użyjemy funkcji "insert":

```
import xml.etree.ElementTree as et

r=et.parse("dane.xml").getroot()
for e in r:
    print(e.tag,e.text,e.attrib)

nowy=et.Element("samochod")
nowy.text="Renault"
r.insert(0,nowy)

print("-----")
for e in r:
    print(e.tag,e.text,e.attrib)
```

Funkcja "insert" jest wywoływana na rzecz tego elementu w którym chcemy umieścić nowy element jako podelement. Przyjmuje ona dwa parametry. Pierwszy to pozycja (w tym przypadku pierwsza), drugi to element który ma zostać dodany.

Usuwanie elementów

Kasować elementy z drzewa XML możemy po pozycji:

```
import xml.etree.ElementTree as et

r=et.parse("dane.xml").getroot()
for e in r:
    print(e.tag,e.text)

del r[0]
print("-----")
for e in r:
    print(e.tag,e.text)
```

lub po nazwie :

```
import xml.etree.ElementTree as et

r=et.parse("dane.xml").getroot()
for e in r:
    print(e.tag,e.text)

naz =r.find('nazwisko')
r.remove(naz)

print("-----")
for e in r:
    print(e.tag,e.text)
```

Zapis drzewa XML do pliku

Zapis do pliku odbywa się w podobny sposób jak zapis do pliku tekstowego. Tym razem skorzystamy z funkcji `write` wbudowanej w element. Zadbamy też o to by zapis wykorzystał właściwe kodowanie. Poniższy przykład odczytuje drzewo XML z pliku `"dane.xml"`, a następnie bez jego modyfikowania zapisuje je do pliku `"dane2.xml"`:

```
import xml.etree.ElementTree as et
d = et.parse("dane.xml")
d.write("dane2.xml", encoding="utf-8")
```


Dane zdalne - wykorzystanie usług sieciowych

Pobieranie danych za pomocą GET

Usługi sieciowe służą do wymiany informacji za pomocą protokołu HTTP. Zwykle ten protokół wykorzystywany jest do przesyłania stron internetowych, ale zamiast kodu HTML mogą także zostać przesłane dane w formacie np. JSON czy XML. Usługi sieciowe mogą nie tylko zwracać jakieś dane, ale również je odbierać. W ten sposób często integrowane są systemy napisane w różnych językach programowania czy różne autonomiczne elementy jednej aplikacji, zwłaszcza na fali ostatniej mody na mikroserwisy.

Do obsługi usług sieciowych w Pythonie mamy moduł "requests", który trzeba zaimportować do naszego skryptu zanim zaczniemy z niego korzystać. Poniżej przykład korzystania z takich danych zdalnych. Wprawdzie użyte tutaj wywołanie http prowadzi do pliku statycznego, ale w zasadzie nie ma to znaczenia, bo pod tym adresem równie dobrze mogłaby znajdować się usługa sieciowa która zwróciłaby dane w takiej samej formie:

```
import requests
odpowiedz=requests.get("http://jsystems.pl/static/blog/python/dane.js
on",auth=('user','pass'))
print(odpowiedz.status_code)
dane = odpowiedz.json()
print(dane)
```

Nawiązanie połączenia z usługą odbywa się za pomocą funkcji "get" która za parametr przyjmuje adres usługi. Zwraca ona obiekt odpowiedzi zawierający same dane, ale też dodatkowe informacje nagłówkowe. Jedną z takich informacji jest na przykład kod statusu, który tu wyświetlam w linii 3. Kod o numerze 200 oznacza prawidłową odpowiedź. To są te same statusy co i przy każdym innym wywołaniu http - czyli np. 404 oznacza brak zasobu do którego się odwołujemy. W funkcji get pojawia się tu jeszcze jeden parametr - auth. Jest on opcjonalny, służy do ewentualnej autoryzacji która w tym przypadku jest całkiem zbędna.

Sam obiekt odpowiedzi jest obiektem opakowującym, dlatego jeśli zechcemy wydłubać z niego dane wywołujemy dodatkowo funkcję "json" która to zwraca nam je w formacie słownika. Wynik działania:

```
{'imie': 'Andrzej', 'nazwisko': 'Klusiewicz', 'adres': {'miasto': 'Warszawa', 'kod': '02-019'},
'jezyki': ['polski', 'angielski', 'Java', 'R', 'Python', 'PL/SQL']}
```

Gdybym się zechciał dobrać do konkretnej wartości w danych, to dalej postępuję tak jak z każdym innym danymi zawartymi w strukturze słownika. Poniżej przykład odczytania miasta z adresu zawartego w zwróconych danych:

```
import requests
odpowiedz=requests.get("http://jsystems.pl/static/blog/python/dane.json")
dane = odpowiedz.json()
print(dane['adres']['miasto'])
```

Przesyłanie danych za pomocą POST

Do przesyłania danych do usług sieciowych wykorzystujemy ten sam moduł. Przygotowałem sobie pusty słownik "dane" który następnie przesyłam do usługi sieciowej za pomocą funkcji "post". Funkcja ta poza adresem i danymi przyjmuje również parametr "headers" który służy do przekazywania informacji m.in o typie zawartości przesłanych informacji. Podobnie jak funkcja "get" przyjmuje również parametr "auth" którego tu jednak nie użyłem:

```
import requests
dane=dict()
odpowiedzWysylka=requests.post("http://jsystems.pl/static/blog/python/dane.json",data=dane,headers={"Content-Type":"application/json"})
print(odpowiedzWysylka.status_code)
```

Wykorzystanie baz danych

Łączenie z serwerem bazy danych

Sposób łączenia z serwerem bazodanowym w Pythonie jest zależny od tego z jakim rodzajem bazy danych się łączymy. Po nawiązaniu połączenia, sposób korzystania z bazy (pobieranie, zmiana, kasowanie i ładowanie danych) odbywa się tak samo dla każdego rodzaju bazy relacyjnej. Bazy obiektowe rządzą się swoimi prawami, ale nie będziemy ich tu omawiać.

Łączenie z serwerem PostgreSQL

Do łączenia z serwerem PostgreSQL proponuję bibliotekę "psycopg2". Łączenie z bazą PostgreSQL odbywa się z jej pomocą w ten sposób:

```
import psycopg2
polaczenie=psycopg2.connect(host="jsystems.pl",database="demo",user="demo",password="demo",port=5432)
```

Nie trzeba tu chyba za wiele tłumaczyć. Trzeba podać host, nazwę bazy danych, użytkownika, hasło i port. W zamian dostajemy utworzony obiekt połączenia, dzięki któremu będziemy mogli na tej bazie wykonywać zapytania.

Łączenie z serwerem Oracle

Do łączenia z serwerem Oracle trzeba będzie sobie doinstalować do środowiska bibliotekę "cx_Oracle". Jest dostępna w zdalnym repo dla PIP, więc jej instalacja nie powinna nastręczać trudności. Po tej operacji możemy przystąpić do nawiązania połączenia:

```
import cx_Oracle
polaczenie = cx_Oracle.connect('uzytkownik/haslo@host/baza_danych')
```

Pobieranie danych z użyciem SELECT

Niezależnie od tego, do jakiej bazy się łączymy (Oracle, PostgreSQL, MySQL, SQL Server etc), po nawiązaniu połączenia sposób obsługi bazy jest taki sam. Na potrzeby przykładów w tym rozdziale wykorzystam stworzone połączenie do bazy PostgreSQL. W bazie do której się łączę znajduje się tabela "owoce". Ma ona dwie kolumny: "numer" i "nazwa". Poniżej kompletny kod łączący się z tą bazą, wykonujący zapytanie i zwracający wynik na konsoli.

```
import psycopg2
polaczenie=psycopg2.connect(host="jsystems.pl",database="demo",user="
demo",password="demo",port=5432)
kursor = polaczenie.cursor()
sql="select * from owoce"
kursor.execute(sql)
for w in kursor:
    print(w)
kursor.close()
```

Ponieważ łączenie z bazą zostało omówione już wcześniej, skupmy się teraz na samym pobieraniu danych i ich przetwarzaniu. Interesuje nas linijka od deklaracji zmiennej "kursor". Kursor to specjalny obiekt pozwalający na wykonywanie operacji na bazie danych. Uzyskamy go wywołując funkcję "cursor()" na rzecz obiektu połączenia. Sam kursor posiada funkcję "execute". Podajemy do niej przez parametr treść zapytania SQL które chcemy wykonać. Nie ma znaczenia czy będzie to zapytanie "SELECT" czy jakiś DDL albo DML. Jeśli zapytanie zwróci jakieś dane, będziemy mogli wykorzystać pętlę i je pobrać z użyciem kursora (co widać w przedostatnich 2 liniach). Kursor dobrze jest na koniec zamknąć by niepotrzebnie nie blokować zasobów. Wynik zapytania zostanie nam zwrócony w postaci krotek. Oto wynik działania poniższego kodu:

(1, 'Pomarańcza')

(2, 'Jabłko')

(3, 'Cytryna')

(4, 'Owoc żywota Twojego je ZUS')

Skoro są to krotki, to możemy postępować z nimi jak z każdym innym zestawem krotek. Na ten przykład odczytajmy tylko 2 kolumnę (o indeksie 1 ;)) :

```
import psycopg2
polaczenie=psycopg2.connect(host="jsystems.pl",database="demo",user="
demo",password="demo",port=5432)
kursor = polaczenie.cursor()
sql="select * from owoce"
kursor.execute(sql)
for w in kursor:
    print(w[1])
kursor.close()
```

Tym razem dane wydrukowane na konsolę wyglądają tak:

Pomarańcza
Jabłko
Cytryna
Owoc żywota Twojego je ZUS

Ważna informacja dla osób równie zeschizowanych na punkcie wydajności co i ja (czyli mam nadzieję również Ciebie szanowny czytelniku ;)): Dane odczytywane są z bazy w momencie wywołania funkcji "execute". Fetch nie następuje w pętli kursorowej (do czego mogli przywyknąć użytkownicy takich języków jak np. PL/SQL), a od razu w momencie wywołania wykonania zapytania.

Wstawianie, zmiana i kasowanie danych, oraz operacje DDL

Poniższy kod wstawia jeden wiersz do tabeli owoce. W zasadzie różnica w poniższym kodzie w stosunku do tego z instrukcją "SELECT" tkwi jedynie w treści zapytania, oraz wykorzystaniu nowej instrukcji "commit" (przedostatnia linia). Służy ona do zatwierdzenia transakcji. Jeśli tego nie zrobisz, zmiana będzie widoczna tylko dla Twojej sesji, a kiedy ją zakończysz bez zatwierdzenia transakcji - dane nigdy nie zostaną utrwalone.

```
import psycopg2
polaczenie=psycopg2.connect(host="jsystems.pl",database="demo",user="demo",password="demo",port=5432)
kursor = polaczenie.cursor()
sql="insert into owoce(nazwa) values ('Granat')"
kursor.execute(sql)
polaczenie.commit()
kursor.close()
```

W dokładnie ten sam sposób możesz wykonać aktualizację danych, kasowanie, czy dowolną operację DDL. Należałoby jedynie zmienić treść zapytania. Pytanie które na przykład mi przychodzi do głowy przy okazji takich insertów jak ten powyższy "a pod jakim ID wylądował dodany owoc i jak ten ID odczytać?". To dobre pytanie i miło mi że o to pytasz ;). Kolumna ID w tabeli "owoce" jest typu "serial" - czyli wartości w tej kolumnie są generowane po stronie bazy danych. Nie dodałem wartości która ma trafić do tej kolumny z poziomu Pythona, a jednak chciałbym wiedzieć pod jakim identyfikatorem wylądował "Granat". Czasami trzeba tę wartość odebrać np. ze względu na klucze obce gdy chcemy do tego wiersza dowiązać jakieś zależne wiersze w innych tabelach. Przykład: faktura i produkty zawarte na fakturze - musi istnieć relacja, a wiersze z produktami muszą wskazywać na wiersz faktury do której się odnoszą. Poniżej przykład jak się dobrać do takiego generowanego ID, albo jakiegokolwiek innej wartości generowanej po stronie bazy danych (np. przez trigger, czy pobieranej z sekwencji). Pomijam już elementy wielokrotnie powtarzane - nawiązywanie połączenia.

```
kursor = polaczenie.cursor()
sql="insert into owoce(nazwa) values ('Arbuz') returning numer"
kursor.execute(sql)
id=str(kursor.fetchone()[0])
print('id='+id)
polaczenie.commit()
kursor.close()
```

Zmieniłem nieco zapytanie. Aby można było odebrać taką dynamicznie generowaną wartość należy użyć klauzuli returning wskazującej nazwę kolumny z której taka dynamiczna wartość ma zostać odczytana. Samą wartość odczytujemy już za pomocą funkcji fetchone() uruchamianej na rzecz kursora.

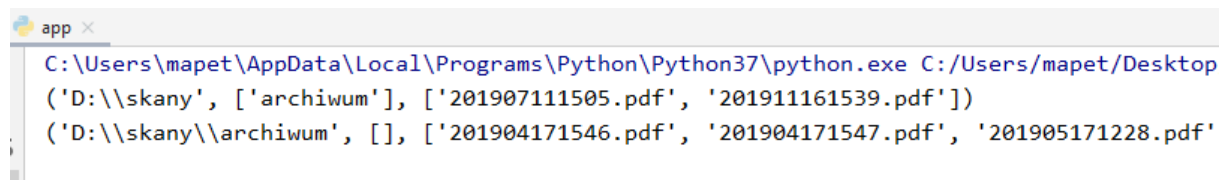
Moduł os i poruszanie się po systemie plików

Funkcja os.walk

os.walk to funkcja pozwalająca na przeszukiwanie katalogów. Jako argument przyjmuje ścieżkę startową (w minimalnej wersji konfiguracji), a następnie umożliwia nam iterację po wynikach. Ilustruje to poniższy przykład:

```
import os
for p in os.walk('D:\\skany'):
    print(p)
```

Wynika działania:

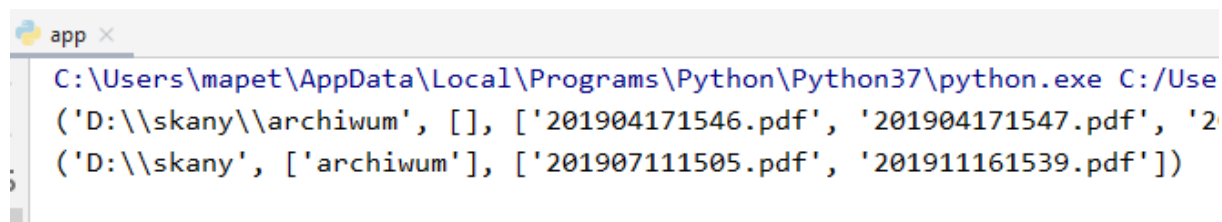


```
app x
C:\Users\mapet\AppData\Local\Programs\Python\Python37\python.exe C:/Users/mapet/Desktop
('D:\\skany', ['archiwum'], ['201907111505.pdf', '201911161539.pdf'])
('D:\\skany\\archiwum', [], ['201904171546.pdf', '201904171547.pdf', '201905171228.pdf'])
```

Iterując po wyniku otrzymujemy krotki. Pierwsza wartość (str) w elemencie to ścieżka katalogu którego dotyczą dalsze dane w krotce. Druga wartość w krotce to katalogi zawarte w danym folderze, trzecia to pliki. Walk pracuje rekurencyjnie, wejdziemy więc w głąb wszystkich katalogów. Możemy też zmienić kierunek przeszukiwania katalogów:

```
import os
for p in os.walk('D:\\skany', False):
    print(p)
```

Ustawienie w drugim argumencie funkcji walk wartości fałsz zmienia kierunek przeszukiwania:



```
app x
C:\Users\mapet\AppData\Local\Programs\Python\Python37\python.exe C:/Use
('D:\\skany\\archiwum', [], ['201904171546.pdf', '201904171547.pdf', '2
('D:\\skany', ['archiwum'], ['201907111505.pdf', '201911161539.pdf'])
```

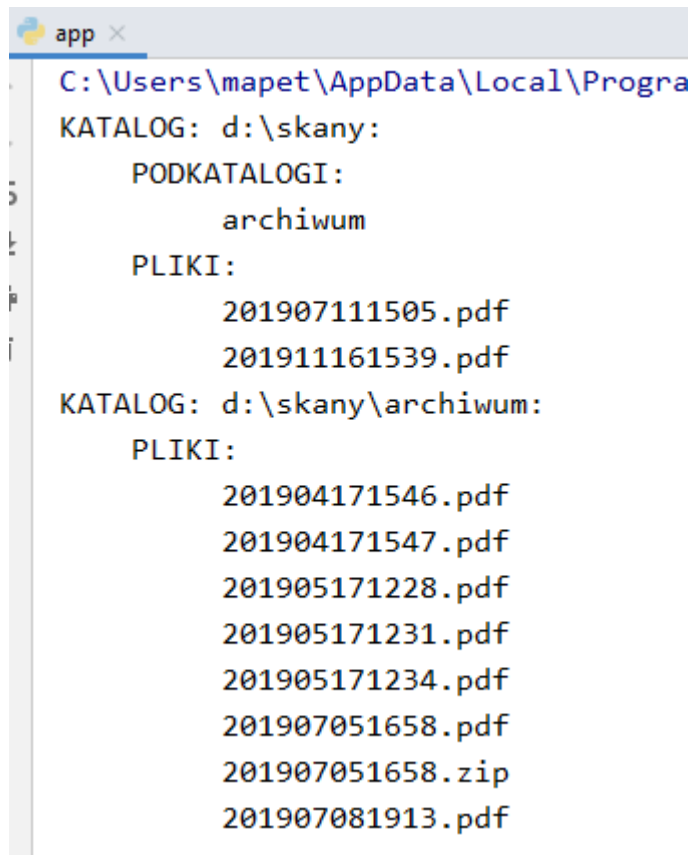
Ponieważ otrzymujemy krotki, możemy też odebrać dane do osobnych list. Przy okazji podrzucam patent na wizualizację drzewa:

```

import os
for katalog, podkatalogi, pliki in os.walk("d:\\skany"):
    print(f'KATALOG: {katalog}:')
    if(len(podkatalogi)>0):
        print(f'    PODKATALOGI: ')
        for po in podkatalogi:
            print(f'        {po}')
    if(len(pliki)>0):
        print(f'    PLIKI: ')
        for pl in pliki:
            print(f'        {pl}')

```

Wynik działania:



```

C:\Users\mapet\AppData\Local\Progra
KATALOG: d:\skany:
    PODKATALOGI:
        archiwum
    PLIKI:
        201907111505.pdf
        201911161539.pdf
KATALOG: d:\skany\archiwum:
    PLIKI:
        201904171546.pdf
        201904171547.pdf
        201905171228.pdf
        201905171231.pdf
        201905171234.pdf
        201907051658.pdf
        201907051658.zip
        201907081913.pdf

```


Poruszanie się po systemie plików

Zmiana i sprawdzenie aktualnego katalogu

Moduł os posiada wbudowane funkcje chdir i getcwd umożliwiające odpowiednio zmianę i sprawdzenie aktualnej ścieżki.

```
import os
os.chdir('d:\\skany')
print(os.getcwd())
```

Listowanie zawartości katalogu

Sprowadza się to do wywołania funkcji listdir:

```
print(os.listdir())
```

W efekcie uzyskamy zarówno pliki jak i katalogi w postaci jednej listy:

```
['201911161539.pdf', 'archiwum', 'nowe']
```

Sprawdzenie czy katalog istnieje

Sprowadza się to do wywołania funkcji exists:

```
import os
if os.path.exists('d:\\skany'):
    print('jest taki katalog')
else:
    print('nie ma takiego katalogu')
```

Sprawdzanie czy mamy do czynienia z plikiem czy z katalogiem

Moduł `os` ma dwie funkcje które umożliwią nam spełnić takie wymaganie – `isdir` i `isfile` weryfikujące odpowiednio czy mamy do czynienia z katalogiem czy z plikiem:

```
import os
if os.path.isdir('d:\\skany'):
    print('to jest katalog')
else:
    print('to jest plik')

if os.path.isfile('d:\\skany\\201907111505.pdf'):
    print('to jest plik')
else:
    print('to jest katalog')
```

Sprawdzanie wielkości pliku

Stosujemy funkcję `getsize`:

```
import os
size=os.path.getsize('d:\\skany\\201907111505.pdf')
print(f'wielkość pliku={size} bajtów')
```

Efekt na konsoli:

wielkość pliku=1262625 bajtów

Tworzenie i kasowanie katalogu

Istnieją do tego funkcje odpowiednio `mkdir` i `rmdir`

```
import os
os.mkdir('d:\\skany\\nowe')
os.rmdir('d:\\skany\\nowe')
```

Kasowanie pliku

Sprowadza się do wywołania funkcji `remove`:

```
import os
os.remove('d:\\skany\\201907111505.pdf')
```

Moduł subprocess i wywoływanie komend systemu operacyjnego

Moduł subprocess zastępuje: `os.system()`, `os.spawn*()`, `commands.*()`, `os.popen*()`, `popen2.*()` oferując nam wyższy model abstrakcji i większą wygodę wykorzystania. Omówimy tu dwie funkcje, jedna to `call`, druga `Popen`. Różnica polega na tym, że pierwsza powoduje wykonanie liniowe wywoływanej komendy w ramach wątku naszej aplikacji, druga generuje osobny proces w systemie operacyjnym.

Funkcja call

Wywołamy ping na Onecie. Argumenty do funkcji `call` podajemy w postaci listy. Podajemy w niej kolejne komendy i ich parametry. Zasadniczo elementy zostaną rozdzielone spacjami i tak wykonane w systemie:

```
import subprocess
result=subprocess.call(['ping' , '-n', '10', 'onet.pl'])
print(f"result={result}")
print('koniec')
```

Wynik na konsoli:

```
Pinging onet.pl [213.180.141.140] with 32 bytes of data:
Reply from 213.180.141.140: bytes=32 time=6ms TTL=55
Reply from 213.180.141.140: bytes=32 time=6ms TTL=55
Reply from 213.180.141.140: bytes=32 time=6ms TTL=55
Reply from 213.180.141.140: bytes=32 time=6ms TTL=55
Reply from 213.180.141.140: bytes=32 time=6ms TTL=55
Reply from 213.180.141.140: bytes=32 time=6ms TTL=55
Reply from 213.180.141.140: bytes=32 time=6ms TTL=55
Reply from 213.180.141.140: bytes=32 time=6ms TTL=55
Reply from 213.180.141.140: bytes=32 time=6ms TTL=55
Reply from 213.180.141.140: bytes=32 time=6ms TTL=55
```

```
Ping statistics for 213.180.141.140:
    Packets: Sent = 10, Received = 10, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 6ms, Maximum = 6ms, Average = 6ms
result=0
koniec
```

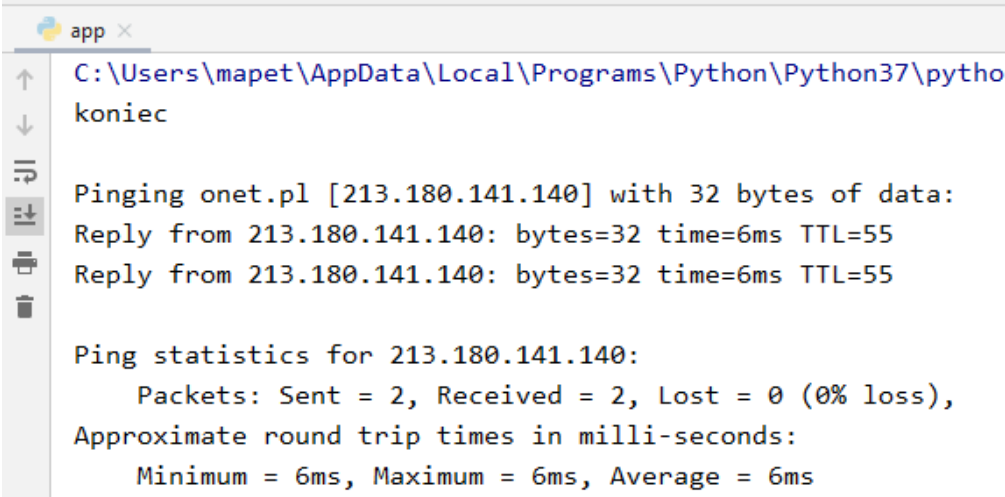
Zastanawiać może enigmatyczne „result” odbierane od funkcji. Jeśli komenda systemu operacyjnego wykona się z błędem, zostanie zwrócona wartość 1. Jeśli wszystko było ok dostaniemy 0. Wyświetliłem też komunikat „koniec” by zaprezentować liniowe wykonanie. Funkcja ta pozwala jeszcze na ustawienie argumentu `shell=True`, co spowoduje zaczytywanie zmiennych środowiskowych i generalnie wykonanie za pośrednictwem powłoki.

Funkcja Popen

Wykonam tę samą czynność co w przykładzie funkcją call. Tym razem jednak użyję funkcji Popen powodującej uruchomienie czynności w osobnym procesie systemu operacyjnego:

```
import subprocess
result=subprocess.Popen(['ping','-n','2','onet.pl'])
print('koniec')
print(result.communicate())
```

Efekt:



```
C:\Users\mapet\AppData\Local\Programs\Python\Python37\pytho
koniec

Pinging onet.pl [213.180.141.140] with 32 bytes of data:
Reply from 213.180.141.140: bytes=32 time=6ms TTL=55
Reply from 213.180.141.140: bytes=32 time=6ms TTL=55

Ping statistics for 213.180.141.140:
    Packets: Sent = 2, Received = 2, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 6ms, Maximum = 6ms, Average = 6ms
```

Tym razem komunikat „koniec” wyświetlił się na początku. Widzimy dzięki temu że proces faktycznie rozpoczął się w osobnym procesie.

Wyrażenia regularne

Wyrażenia regularne służą do wyszukiwania w tekście jego elementów wg. zadanych wzorców. Pozwalają na przykład odnaleźć i wyciąć wszystkie elementy pasujące do wzorca np numery telefonów czy adresy email. Zaczniemy od przykładów bardzo prostych. Wraz z kolejnymi zagadnieniami poznamy też kolejne funkcje służące do pracy z wyrażeniami regularnymi.

W pierwszych przykładach będę posługiwał się funkcją "findall" odnajdującą wszystkie wystąpienia pasujące do wzorca. Jest to po prostu najwygodniejsza funkcja na potrzeby przykładów, ale w wielu przypadkach jej zastosowanie nie będzie wydajne - czasem zechcesz np. znaleźć tylko pierwsze wystąpienie elementu w bardzo długim tekście i nie chcesz czytać i przetwarzać całości. Wszystkie funkcje które będą poruszane w ramach tego rozdziału znajdziesz w bibliotece "re" dostępnym natywnie w Pythonie.

Podstawowe wyszukiwanie

Poniżej bardzo naiwny przykład, ale służy zaprezentowaniu zasad działania funkcji "findall":

```
import re
tekst='siała baba mak i dostała 10 lat'
wzorzec='baba'
print (re.findall(wzorzec,tekst))
```

Wynik na konsoli:

```
['baba']
```

Pierwsza linia to oczywiście import odpowiedniej biblioteki. Zmienna "tekst" zawiera tekst w którym będziemy wyszukiwać. Zmienna "wzorzec" określa element szukany. Zwykle znajdzie się tutaj jakieś wyrażenie a nie zwyczajny ciąg tekstowy. Pod deklaracjami zmiennych drukujemy wynik działania funkcji "findall" która w postaci tablicy zwraca nam wszystkie elementy pasujące do wzorca. Jako pierwszy parametr podajemy wzorzec wyszukiwania, drugi podajemy tekst który zamierzamy przeszukiwać. Za chwilę przedstawię symbole określające rodzaj wyszukiwanych znaków oraz symbole określające ilość wystąpień, byśmy mogli już dalej bez przeszkód zająć się wyszukiwaniem.

Typy znaków

Poniżej znajduje się zestawienie symboli które pozwalają nam odnajdować elementy.

Symbol	Opis
\d	Cyfry
\D	Znaki nie będące cyfrą
\w	Alfanumeryki
\W	Wszystko co nie jest alfanumerykiem
\s	Białe znaki
\S	Wszystko co nie jest białym znakiem
.	Dowolny znak
[a-f]	Małe litery z zakresu a-f
[A-F]	Duże litery z zakresu A-F
[0-3]	Cyfry z zakresu 0-3
[a-z0-6]	Małe litery z zakresu a-z lub cyfry z zakresu 0-6
[^a-f]	Elementy nie zawierające się w zbiorze małych liter w zakresie a-f

Kwantyfikatory ilościowe

Poniżej zestawienie symboli określających krotność wystąpień poszukiwanego elementu.

Kwantyfikator	Opis
*	Dowolna ilość znaków - w tym zero!
+	Co najmniej jedno wystąpienie - ale może być też więcej!
?	Jedno lub zero wystąpień - ale nie więcej
{1,5}	Od jednego do pięciu wystąpień
{5,}	Co najmniej 5 wystąpień
{,5}	Nie więcej niż 5 wystąpień

Wykorzystanie symboli i kwantyfikatorów do wyszukiwania elementów według wzorca

Połączmy teraz poznane symbole i kwantyfikatory ilościowe i wyszukajmy coś w tekście:

```
import re
tekst='Badanie statystyczne 565 dzieł sztuki różnych wielkich
malarzy, przeprowadzone w 1999, wykazało, że ci artyści nie użyli
złotego podziału w wymiarach swoich płócien.' \
      'Badanie stwierdziło, że średni stosunek dwóch boków badanych
obrazów wynosi 1,34, ze średnimi dla poszczególnych malarzy
obejmującymi od 1,04 (Goya) do 1,46 (Bellini)[35]. ' \
      'Z drugiej strony, Pablo Tosto wymienił ponad 350 dzieł znanych
artystów, z których ponad 100 miało płótna o proporcjach złotego
prostokąta i pierwiastka z 5, natomiast inne' \
      ' proporcje takie jak pierwiastki z 2, 3, 4 i 6[36]. '
wzorzec='\d'
print (re.findall(wzorzec,tekst))
```

Wykorzystałem symbol "\d" do wyszukania wszystkich cyfr z tekstu. Wynik na konsoli:

```
['5', '6', '5', '1', '9', '9', '9', '1', '3', '4', '1', '0', '4', '1', '4', '6', '3', '5', '3', '5', '0', '1', '0', '0', '5', '2',
'3', '4', '6', '3', '6']
```

Jak widzimy każda cyfra jest jako osobny element zbioru, ponieważ szukaliśmy jednej cyfry - nie określiliśmy kwantyfikatora ilościowego. Tym razem poszukajmy liczb składających się z 3 lub 4 samych cyfr (bez przecinka rozdzielającego wartości całkowitych od dziesiętnych - liczb zawierających ułamki nie chcemy:

```
wzorzec='\d{3,4}'
```

Wynik:

```
['565', '1999', '350', '100']
```

Co tu się zadziało? We wzorcu mamy symbol "\d" oznaczający wyszukiwanie cyfr, oraz kwantyfikator ilościowy "{3,4}" określający ilość wystąpień od 3 do 4. Kwantyfikator ilościowy odnosi się do poprzedzającego elementu. Trochę trzeba się wypowiadać jak Yoda: "znajdź mi cyfry trzy lub cztery". Poszukajmy więc liczb zawierających część ułamkową. Wzorzec:

```
wzorzec=' \d, \d{2} '
```

Tym razem powiedzieliśmy : "znajdź element składający się z jednej cyfry, po którym następuje przecinek, po którym następują dwie kolejne cyfry. Wynik:

```
['1,34', '1,04', '1,46']
```

Znajdźmy teraz numer telefonu:

```
import re
tekst='Pod tym numerem możesz zamówić kebsika: 22 299 53 69. Trzeba
prosić do telefonu Panią Bożenkę. '
wz=' [\d ]{9,} '
print(re.findall(wz,tekst))
```

Wynik:

```
[' 22 299 53 69']
```

Co oznacza wzorzec "[\d]{9,}"? Element składający się z cyfr albo spacji o długości co najmniej 9 znaków. Może się też pojawić sytuacja w której mamy do wyszukania jakieś znaki mogące być potraktowane jako znaki specjalne np. ".". W takiej sytuacji należy wyłączyć ich specjalne znaczenie za pomocą znaku "\\".

Testy jednostkowe - framework py.test

Czy do stosowania testów automatycznych w ogóle muszę kogokolwiek przekonywać? Wyobrażasz sobie że przy każdej zmianie kodu testujesz ręcznie wszystkie funkcjonalności systemu, bo jakaś zmiana mogła zepsuć działanie którejś z funkcji? Albo może testować metodą "powinno działać" ;) lub japońską metodą "jako-tako" i zakładać że przecież klient przetestuje na produkcji (pozdrawiam Microsoft ;)) ? Automatyczne testy sprawdzą poprawność działania Twojego kodu, za każdym razem gdy zechcesz w sposób automatyczny, w ułamkowej części czasu jaki musiałbyś poświęcić na testy ręczne. Czy zatem warto pisać testy? Myślę że po postawieniu tak retorycznego pytania możemy po prostu przejść do opisu tego jak to się robi.

Podstawowe testy

Zacniemy od stworzenia prostego testu jednostkowego dla banalnej funkcji sumującej dwie liczby. Stworzyłem nowy projekt, a w nim plik o nazwie narzedzia.py, którego zawartość wygląda następująco:

```
def sumuj(a,b):  
    return a+b
```

Chciałbym teraz przetestować poprawność działania tej funkcji. Dodaję więc plik o nazwie test_narzedzia.py a w nim umieszczam poniższy kod:

```
import narzedzia as n  
  
def test_sumuj():  
    assert n.sumuj(5,3)==8
```

Nazwa pliku "test_narzedzia.py" nie jest przypadkowa. Które moduły i funkcje służą do testowania a które są testowane i jak je odróżnić? Pytest po uruchomieniu szuka automatycznie plików których nazwa zaczyna się od prefiksu "test_", a w nich poszukuje funkcji których nazwa również zaczyna się od "test_". Te właśnie funkcje uzna za testy do uruchomienia. Ot cała tajemnica.

Skoro już wiemy jak to działa, to przeanalizujemy zawartość pliku z testami (drugi przykład z kodem wyżej). W pierwszej kolejności trzeba oczywiście zaimportować moduł który będzie podlegał testom. Poniżej deklaruję funkcję o nazwie "test_sumuj" której zadaniem będzie przetestowanie funkcji "sumuj" z modułu "narzedzia". Ponieważ działanie funkcji "sumuj" sprowadza się do zwrócenia sumy dwóch liczb podanych przez argumenty - to poprawność takiego właśnie sposobu działania testujemy. Pojawia się tu słowo kluczowe "assert". Oznacza ono mniej więcej "upewnij się że". Linie:

```
assert n.sumuj (5, 3) == 8
```

moglibyśmy wytłumaczyć jako "upewnij się że wynik działania funkcji sumuj po podaniu jej 5 i 3 wynosi 8". Generalnie musi to być wyrażenie logiczne co do którego jesteśmy w stanie orzec czy jest prawdziwe czy nie. Równie dobrze mogłoby wyglądać tak:

```
assert 1==1
```

Czas uruchomić testy. Służy do tego konsolowe narzędzie "pytest". Wykona za nas całą pracę, musimy tylko je wywołać w odpowiedni sposób w odpowiednim miejscu. Przechodzimy do katalogu projektu z poziomu konsoli Windows, lub wybieramy "Terminal" w dolnej, lewej części interfejsu PyCharm. Wpisujemy "pytest" i naciskamy enter:

```
(venv) D:\data\workspace-python\testy_jednostkowe>pytest
===== test session starts =====
platform win32 -- Python 3.7.2, pytest-5.0.1, py-1.8.0, pluggy-0.12.0
rootdir: D:\data\workspace-python\testy_jednostkowe
collected 1 item

test_narzedzia.py . [100%]

===== 1 passed in 0.11 seconds =====

(venv) D:\data\workspace-python\testy_jednostkowe>
```

Jak widać na powyższej ilustracji, pytest sam odnalazł testy. Wykrył plik "test_narzedzia.py" który z racji posiadania w nazwie prefiksu "test_" został uznany za moduł z testami. Przeszukał plik w poszukiwaniu funkcji których nazwy również zawierają taki prefiks i je wykonał. Jako wynik widzimy potwierdzenie że 100% testów z pliku "test_narzedzia.py" zakończyło się z wynikiem pozytywnym.

Nie wiemy jednak jakie funkcje zostały wywołane. Aby się tego dowiedzieć, możemy użyć przełącznika "-v":

```
(venv) D:\data\workspace-python\testy_jednostkowe>pytest -v
===== test session starts =====
platform win32 -- Python 3.7.2, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -- d:\data\workspace-python\testy_jednostkowe\venv\scripts\python.exe
cachedir: .pytest_cache
rootdir: D:\data\workspace-python\testy_jednostkowe
collected 1 item

test_narzedzia.py::test_sumuj PASSED [100%]

===== 1 passed in 0.02 seconds =====

(venv) D:\data\workspace-python\testy_jednostkowe>
```

Tym razem widać, że wywołana została funkcja "test_sumuj". Teraz zmodyfikuję nieco funkcję testującą by umyślnie spowodować błąd:

```
def test_sumuj():
    assert n.sumuj(5,3)==18
```

Wynik działania:

```
(venv) D:\data\workspace-python\testy_jednostkowe>pytest -v
===== test session starts =====
platform win32 -- Python 3.7.2, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -- d:\data\workspace-python\testy_jednostkowe\venv\scripts\python.exe
cachedir: .pytest_cache
rootdir: D:\data\workspace-python\testy_jednostkowe
collected 1 item

test_narzedzia.py::test_sumuj FAILED [100%]

===== FAILURES =====
_____ test_sumuj _____

  def test_sumuj():
>     assert n.sumuj(5,3)==18
E       assert 8 == 18
E       -8
E       +18

test_narzedzia.py:4: AssertionError
===== 1 failed in 0.14 seconds =====

(venv) D:\data\workspace-python\testy_jednostkowe>
```

Tym razem obaliśmy test. Pytest pokazał nam nawet w którym miejscu testy nie przeszły i przy jakiej wartości test przechodził a jaka jest teraz. Mam tu na myśli te linijki z wartościami "-8, +18". Przy wartości 8 test był ok, przy wartości 18 nie przechodzi. Tą informację odnośnie konkretnych wartości wcześniej i teraz dostaniemy tylko korzystając z przełącznika -v. Bez niego zobaczymy tylko jaka metoda i na której linijce nie przeszła:

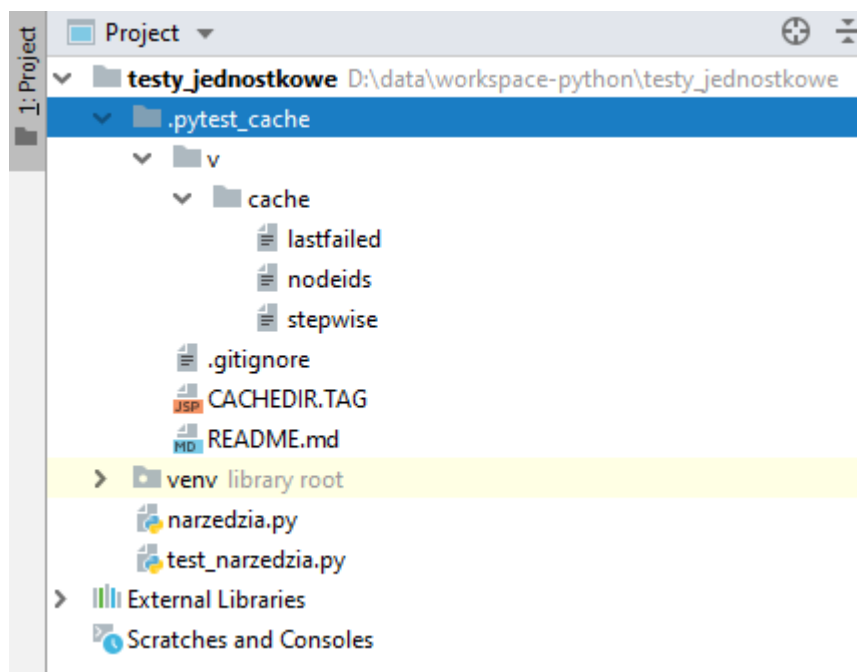
```
===== FAILURES =====
test_sumuj

def test_sumuj():
>     assert n.sumuj(5,3)==20
E     assert 8 == 20
E     + where 8 = <function sumuj at 0x000001DA972812F0>(5, 3)
E     +   where <function sumuj at 0x000001DA972812F0> = n.sumuj

test_narzedzia.py:4: AssertionError
===== 1 failed in 0.22 seconds =====
```

W tym przypadku mamy tylko jedną funkcję testującą, jednak warto wiedzieć że w przypadku obłania jednego z testów, pozostałe testy nadal są wykonywane.

Wracając jeszcze na moment do przełącznika "-v" i wartości jaka obowiązywała gdy test przechodził: skąd pytest to wie? Gdzieś musi gromadzić tego rodzaju informacje. Otóż w katalogu projektu tworzy sobie podkatalog ".pytest_cache" który zawiera te i inne informacje związane z pracą pytest.



Uruchamianie wybranych testów

Nie zawsze musimy chcieć uruchamiać wszystkie testy, biorąc pod uwagę zwłaszcza duże systemy gdzie ilość testów może sprawić, że sam proces testowania będzie trwał kilka minut. Dobrze jest więc wiedzieć jak uruchamiać tylko wybrane testy, a możliwości mamy tutaj kilka. Zanim omówimy konkretne przykłady, spójrzmy w jaki sposób zmodyfikowałem projekt. Do pliku "narzedzia.py" dodałem drugą funkcję, tak że w tej chwili zawartość tego pliku wygląda tak:

```
def sumuj(a,b):  
    return a+b  
  
def dajCyfry():  
    return list(range(1,11))
```

Nowa funkcja "dajCyfry" zwraca liczby w zakresie 1-10 w postaci listy. Rozbudowie uległ też moduł testujący:

```
import narzedzia as n  
  
def test_sumuj():  
    assert n.sumuj(5,3)==8  
  
def test_dajCyfryMin():  
    tab=n.dajCyfry()  
    assert min(tab)==1  
  
def test_dajCyfryMax():  
    tab=n.dajCyfry()  
    assert max(tab)==10  
  
def test_dajCyfryLen():  
    tab=n.dajCyfry()  
    assert len(tab)==10
```

Pojawiły się trzy dodatkowe funkcje testujące funkcję "dajCyfry" z modułu "narzedzia". Pierwszy sprawdza czy najmniejsza wartość w zwracanej liście to 1, drugi czy największa to 10, trzeci czy lista zawiera 10 elementów. Do tego wszystkiego dodałem jeszcze w projekcie podkatalog o nazwie "tests", a w nim umieściłem jeden plik "test_rzeczywistosci.py" który zawiera jedną funkcję:

```
def test_czySwiatStanalNaGlowie():  
    assert 2!=1
```

Przejdźmy teraz do wybiórczego uruchamiania testów. Jeśli uruchomię komendę "pytest" w katalogu projektu, odnalezione zostaną wszystkie pliki zaczynające się od prefiksu "test_" i wykonane z nich wszystkie funkcje zaczynające się od tego samego prefiksu. Przeszukiwanie dotyczy nie tylko katalogu w którym się znajdujemy, ale również wszystkich jego podkatalogów.

```
(venv) D:\data\workspace-python\testy_jednostkowe>pytest  
=====
```

```
platform win32 -- Python 3.7.2, pytest-5.0.1, py-1.8.0, pluggy-0.12.0  
rootdir: D:\data\workspace-python\testy_jednostkowe  
collected 5 items  
  
test_narzedzia.py ....  
tests\test_rzeczywistosci.py .  
  
=====
```

```
(venv) D:\data\workspace-python\testy_jednostkowe>
```

Jak widać powyżej pytest znalazł też dodatkowy zestaw testów w podkatalogu. Gdybym zechciał by zostały wykonane tylko testy z jednego pliku, podaję ścieżkę do niego jako argument pytest:

```
(venv) D:\data\workspace-python\testy_jednostkowe>pytest test_narzedzia.py  
=====
```

```
platform win32 -- Python 3.7.2, pytest-5.0.1, py-1.8.0, pluggy-0.12.0  
rootdir: D:\data\workspace-python\testy_jednostkowe  
collected 4 items  
  
test_narzedzia.py ....  
  
=====
```

```
(venv) D:\data\workspace-python\testy_jednostkowe>
```

Raczej można się było tego domyślić ;)

Możesz też nakazać wykonanie testów ze wskazanego katalogu wywołując pytest ze ścieżką tego katalogu jako argumentem:

```
(venv) D:\data\workspace-python\testy_jednostkowe>pytest tests

=====
platform win32 -- Python 3.7.2, pytest-5.0.1, py-1.8.0, pluggy-0.12.0
rootdir: D:\data\workspace-python\testy_jednostkowe
collected 1 item

tests\test_rzeczywistosci.py .

=====

(venv) D:\data\workspace-python\testy_jednostkowe>
```

Rzecz znacznie mniej oczywista to możliwość uruchamiania testów które zawierają określony ciąg w nazwie:

```
(venv) D:\data\workspace-python\testy_jednostkowe>pytest -k dajCyfry -v

=====
platform win32 -- Python 3.7.2, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -- d:
cachedir: .pytest_cache
rootdir: D:\data\workspace-python\testy_jednostkowe
collected 5 items / 2 deselected / 3 selected

test_narzedzia.py::test_dajCyfryMin PASSED
test_narzedzia.py::test_dajCyfryMax PASSED
test_narzedzia.py::test_dajCyfryLen PASSED

=====

(venv) D:\data\workspace-python\testy_jednostkowe>
```

Służy do tego przełącznik "-k" po którym podajemy fragment nazwy. Jak widać, uruchomione zostały trzy funkcje testujące - wszystkie miały w nazwie "dajCyfry" który to ciąg zadeklarowałem jako filtr. Użyłem tu dodatkowo przełącznika -v, tylko po to by wyświetlił mi funkcje które uruchomił (normalnie tego nie robi).

Nie zawsze chcemy lub możemy sobie pozwolić na zmianę nazwy funkcji testującej, np. po to by jak w powyższym przypadku wybierać część z nich. Na szczęście pytest dostarcza też dekoratory które umożliwiają oznaczanie i grupowanie funkcji:

```
import pytest
import narzedzia as n

@pytest.mark.podstawowe
def test_sumuj():
    assert n.sumuj(5, 3) == 8

@pytest.mark.szczegolowe
def test_dajCyfryMin():
    tab = n.dajCyfry()
    assert min(tab) == 1

@pytest.mark.szczegolowe
def test_dajCyfryMax():
    tab = n.dajCyfry()
    assert max(tab) == 10

@pytest.mark.podstawowe
def test_dajCyfryLen():
    tab = n.dajCyfry()
    assert len(tab) == 10
```

Oznaczenie "@pytest.mark.XXX" pozwala na wyznaczenie grup funkcji. Ja swoje podzieliłem na dwie grupy - testy podstawowe i testy szczegółowe. Zwróć uwagę na dodanie importu "import pytest" do pliku - bez tego powyższe dekoratory nie będą działać. Aby wywołać tylko testy oznaczone jakimś "tagiem" używam przełącznika "-m" pytesta. "-v" jak i wcześniej jest tu tylko po to by pokazywał jakie funkcje uruchamia:

```
(venv) D:\data\workspace-python\testy_jednostkowe>pytest -m podstawowe -v
=====
platform win32 -- Python 3.7.2, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -- d:\c
cachedir: .pytest_cache
rootdir: D:\data\workspace-python\testy_jednostkowe
collected 5 items / 3 deselected / 2 selected

test_narzedzia.py::test_sumuj PASSED
test_narzedzia.py::test_dajCyfryLen PASSED
=====
```


Parametryzacja testów

Przyjmijmy że mamy do czynienia z taką sytuacją: tworzymy system który łączy się różnymi bazami danych i wykonuje na nich różne zapytania. Jak więc będą wyglądały testy? Trzeba będzie podłączyć się do każdej z baz i wykonać próbne zapytanie, sprawdzając czy ta czynność nie spowoduje jakiegoś wyjątku. Przygotujmy więc background z testami, korzystając tylko z tego co wiemy dotychczas. Treść pliku "modulik.py":

```
podpietaBaza=None

def podepnijBaze(nazwa):
    global podpietaBaza
    podpietaBaza=nazwa

def wykonajZapytanie():
    global podpietaBaza
    print('Wykonuję zapytanie z użyciem bazy
{}').format(podpietaBaza)
    if(podpietaBaza=='MS SQL'):
        raise Exception('FUUUUUUUU')
    return "ok"
```

Metoda "wykonajZapytanie" będzie powodowała wyjątek gdy zostanie podłączony SQL Server. Zawartość modułu testów "test_modulik.py":

```
import modulik
def test_podepnijBaze():
    bazy=['Oracle', 'PostgreSQL', 'MS SQL', 'MySQL']
    for b in bazy:
        modulik.podepnijBaze(b)
        assert modulik.wykonajZapytanie()=='ok'
    pass
```

Funkcja testująca "test_podepnijBaze" podcina po kolei kolejne bazy z listy i usiłuje wykonać na nich zapytanie. Jak pewnie pamiętasz z poprzedniego listingu - funkcja "wykonajZapytanie" spowoduje wyjątek gdy trafi na "MS SQL". W pozostałych przypadkach zwróci ciąg tekstowy "ok". Tak więc dla pierwszych 2 baz test powinien przejść, a następnie wyłożyć się na trzecim.

Sprawdźmy zatem co się stanie. Po uruchomieniu testu:

```
(venv) D:\data\workspace-python\testy_jednostkowe>pytest test_modulik.py -s -v
=====
platform win32 -- Python 3.7.2, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -- d:\data\
cachedir: .pytest_cache
rootdir: D:\data\workspace-python\testy_jednostkowe
collected 1 item

test_modulik.py::test_podepnijBaze Wykonuję zapytanie z użyciem bazy Oracle
Wykonuję zapytanie z użyciem bazy PostgreSQL
Wykonuję zapytanie z użyciem bazy MS SQL
FAILED

=====

def test_podepnijBaze():
    bazy=['Oracle','PostgreSQL','MS SQL','MySQL']
    for b in bazy:
        modulik.podepnijBaze(b)
>         assert modulik.wykonajZapytanie()=='ok'

test_modulik.py:6:

-----

def wykonajZapytanie():
    global podpietaBaza
    print('Wykonuję zapytanie z użyciem bazy {}'.format(podpietaBaza))
    if(podpietaBaza=='MS SQL'):
>         raise Exception('FUUUUUUU')
E         Exception: FUUUUUUU

modulik.py:11: Exception
=====

(venv) D:\data\workspace-python\testy_jednostkowe>
```

Test zgodnie z oczekiwaniami nie przeszedł, mamy też informację w którym miejscu pojawił się wyjątek. Same testy jednak nie mówią nam dla jakiej wartości nastąpił ten wyjątek. Dodałem sobie drukowanie informacji o podpiętej bazie, i tylko po tym mogę ewentualnie poznać na której bazie się wyłożył test. Ponadto, jeśli na jednej bazie testy polegą, to nie przejdą do sprawdzania kolejnych, tylko zostaną przerwane. Aby rozwiązać oba te problemy, wykorzystamy dekorator "@pytest.mark.parametrize". Mała przeróbka modułu testowego:

```
import modulik
import pytest

dbs = ["Oracle", 'PostgreSQL', 'MS SQL', 'MySQL']

@pytest.mark.parametrize('baza', dbs)
def test_podepnijBaze(baza):
    modulik.podepnijBaze(baza)
    print('{ }\n'.format(baza))
    assert modulik.wykonajZapytanie()=='ok'
```

Powyższa funkcja będzie testowana tylukrotnie, ile wartości znajdzie się na liście "dbs". Dla każdej wartości pytest wygeneruje osobny test. Tym razem funkcja przyjmuje bazę danych przez parametr (w miejsce iteracji po liście baz wewnątrz funkcji). Wartość dla tego parametru zostaje podana dzięki dekoratorowi "@pytest.mark.parametrize". Pierwszym parametrem tego dekoratora jest nazwa parametru do którego wstrzykujemy wartość, drugim lista (lub inna kolekcja po której da się iterować) z której będą pobierane wartości do testów. Pytest dla każdej wartości w kolekcji "dbs" wywoła funkcję test_podepnijBaze raz, podając przez argument funkcji tę wartość.

Sprawdźmy teraz wyniki działania:

```
test_modulik.py::test_podepnijBaze[Oracle] Oracle

Wykonuję zapytanie z użyciem bazy Oracle
PASSED
test_modulik.py::test_podepnijBaze[PostgreSQL] PostgreSQL

Wykonuję zapytanie z użyciem bazy PostgreSQL
PASSED
test_modulik.py::test_podepnijBaze[MS SQL] MS SQL

Wykonuję zapytanie z użyciem bazy MS SQL
FAILED
test_modulik.py::test_podepnijBaze[MySQL] MySQL

Wykonuję zapytanie z użyciem bazy MySQL
PASSED
```

Obserwujemy spodziewany wynik działania. Dla każdej bazy został wykonany jeden test. Mimo porażki testu na "MS SQL" kolejne testy nadal były wykonywane.

Fikstury

Problematyka

Tworzę moduł który będzie robił za coś w stylu lokalnej pamięciowej bazy danych. Jak widać w poniższym fragmencie kodu, mamy w ramach tego modułu listę która jest ładowana przez funkcję "loadDB", a z której dane są pobierane przez funkcje "getData" i "getOne".

```
baza=[]

def loadDB():
    print("##### ŁADOWANIE BAZY #####")
    global baza
    baza=[
        (1, "Marian"),
        (2, "Czesław"),
        (3, "Zenon"),
        (4, "Florian")
    ]

def getData():
    global baza
    return baza

def getOne(x):
    global baza
    return baza[x]
```

Aby dwie ostatnie funkcje mogły cokolwiek zwracać, trzeba najpierw wywołać funkcję "loadDB". Przyjrzyjmy się teraz testom przygotowanym do tego modułu:

```
import nibyDB

def test_getData():
    nibyDB.loadDB()
    assert len( nibyDB.getData() ) > 0
    pass

def test_getOne():
    nibyDB.loadDB()
    assert nibyDB.getOne(0) [1] == 'Marian'
    pass
```

Testy będą działały tak długo, jak długo w powyższych funkcjach przed sięgnięciem do danych będę wywoływał funkcję loadDB. To jest pierwszy problem. Jeśli przy którymś testów o tym zapomnę to test się nie powiedzie, ale nie z powodu wadliwości testowanej funkcji. Drugi problem jest taki, że ładowanie następuje przy każdym teście. Wyobraź sobie teraz że funkcja ładująca pobiera duże ilości danych z jakiejś zdalnej bazy albo ogromnego pliku.

Uruchamiam test by sprawdzić jak to wygląda z tym ładowaniem bazy. W normalnym trybie pytest przechwytyje wszystkie komunikaty lecące na konsolę, więc jeśli chcesz by były one pokazywane, użyj przełącznika "-s":

```
(venv) D:\data\workspace-python\testy_jednostkowe>pytest test_nibyDB.py -s -v

=====

platform win32 -- Python 3.7.2, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -- d:\data\works
cachedir: .pytest_cache
rootdir: D:\data\workspace-python\testy_jednostkowe
collected 2 items

test_nibyDB.py::test_getData ##### ŁADOWANIE BAZY #####
PASSED
test_nibyDB.py::test_getOne ##### ŁADOWANIE BAZY #####
PASSED

=====

(venv) D:\data\workspace-python\testy_jednostkowe>
```

Ładowanie bazy zostało wykonane dwukrotnie, ponieważ wywoływał je każdy z testów. Teraz poza wielkim plikiem dodajmy sobie jeszcze setki takich testów... Dużo rozsądniejszym wyjściem będzie załadowanie danych przed wszystkimi testami jednokrotnie, zamiast każdorazowo przed każdym testem.

Funkcje `setup_module` i `teardown_module`

Tym razem nieco zmodyfikuję zawartość modułu z testami:

```
import nibyDB
def setup_module():
    print("\n##### setup #####")
    nibyDB.loadDB()

def teardown_module():
    print("\n##### bye #####")

def test_getData():
    assert len( nibyDB.getData() ) > 0
    pass

def test_getOne():
    assert nibyDB.getOne(0) [1] == 'Marian'
    pass
```

Pojawiły się dwie nowe funkcje - "`setup_module`" i "`teardown_module`". Nazwy nie są przypadkowe - pytest wywoła te funkcje automatycznie odpowiednio przed wszystkimi testami w danym module, oraz po nich. Możemy to wykorzystać do wstępnej inicjalizacji na początku i np. wyczyszczenia danych na końcu. Kod modułu testującego przerobiłem w taki sposób, że wywołanie funkcji "`loadDB`" pojawia się tylko raz - w funkcji `setup_module()` zamiast w każdej funkcji testującej.

Skutek działania:

```
(venv) D:\data\workspace-python\testy_jednostkowe>pytest test_nibyDB.py -s -v
=====
platform win32 -- Python 3.7.2, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -- d:\data
cachedir: .pytest_cache
rootdir: D:\data\workspace-python\testy_jednostkowe
collected 2 items

test_nibyDB.py::test_getData
##### setup #####
##### ŁADOWANIE BAZY #####
PASSED
test_nibyDB.py::test_getOne PASSED
##### bye #####
```

Jak widzimy na powyższej ilustracji, komunikat ładowania bazy pojawia się tylko raz.

Dekorator @pytest.fixture

Pytest dostarcza ponadto dekoratory które pozwalają uzyskać zbliżony efekt w inny sposób.

```
import nibyDB
import pytest

@pytest.fixture
def load_stuff():
    print("\n##### load #####")
    nibyDB.loadDB()

def test_getData(load_stuff):
    assert len( nibyDB.getData() ) > 0
    pass

def test_getOne(load_stuff):
    assert nibyDB.getOne(0)[1] == 'Marian'
    pass
```

Przyjrzyj się powyższemu przykładowi. W miejsce funkcji "setup_module" pojawia się funkcja "load_stuff" - tym razem jest to nazwa wymyślona przeze mnie. Nad tą funkcją mamy dekorator "@pytest.fixture". Ten dekorator powoduje automatyczne wywołanie funkcji "load_stuff" przed uruchomieniem każdej z funkcji testujących która ma podane w argumencie nazwę funkcji "load_stuff":

```
(venv) D:\data\workspace-python\testy_jednostkowe>pytest test_nibyDB.py -s -v

=====
platform win32 -- Python 3.7.2, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -- d:\data
cachedir: .pytest_cache
rootdir: D:\data\workspace-python\testy_jednostkowe
collected 2 items

test_nibyDB.py::test_getData
##### load #####
##### ŁADOWANIE BAZY #####
PASSED
test_nibyDB.py::test_getOne
##### load #####
##### ŁADOWANIE BAZY #####
PASSED

=====

(venv) D:\data\workspace-python\testy_jednostkowe>
```

W powyższym przykładzie wróciliśmy do wielokrotnego ładowania bazy. Wystarczy do naszego dekoratora dodać atrybut "scope=module" by działało to tak jak setup_module:

```
@pytest.fixture(scope='module')
def load_stuff():
    print("\n##### load #####")
    nibyDB.loadDB()
```

Poza tym nic nie zmieniałem w pozostałym kodzie. Tym razem ładowanie nastąpiło raz, w związku z uruchomieniem tego modułu testującego.

Tak więc jeśli chcesz by jakaś funkcja przygotowująca dane (lub jakakolwiek inna, to przecież bez znaczenia) była wykonywana każdorazowo przed każdą funkcją testującą to dodajesz do niej "@pytest.fixture", a do funkcji testującej dodajesz do argumentów nazwę funkcji przygotowującej (bez podania argumentów czy nawiasów). Jeśli zaś chcesz uruchomić funkcję przygotowującą na przed wszystkimi testami jednorazowo, do dekoratora dodajesz ponadto "(scope='module')"

```
(venv) D:\data\workspace-python\testy_jednostkowe>pytest test_nibyDB.py -s -v
=====
platform win32 -- Python 3.7.2, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -- d:\da
cachedir: .pytest_cache
rootdir: D:\data\workspace-python\testy_jednostkowe
collected 2 items

test_nibyDB.py::test_getData
##### load #####
##### ŁADOWANIE BAZY #####
PASSED
test_nibyDB.py::test_getOne PASSED
=====

(venv) D:\data\workspace-python\testy_jednostkowe>
```

Jeśli drażni Cię (podobnie jak i mnie) konieczność podawania nazwy funkcji przygotowującej jako argument funkcji testującej, możesz wykorzystać przełącznik "autouse":

```
import nibyDB
import pytest

@pytest.fixture(autouse=True)
def load_stuff():
    print("\n##### load #####")
    nibyDB.loadDB()

def test_getData():
    assert len( nibyDB.getData() ) > 0
    pass

def test_getOne():
    assert nibyDB.getOne(0) [1] == 'Marian'
    pass
```

Autouse powoduje po prostu że funkcja której dekorator dotyczy zostanie wywołana przed każdą funkcją testującą (co stałoby się również po prostu po dodaniu tego dekoratora bez żadnych przełączników), z tą różnicą że teraz nie będzie trzeba podawać nazwy funkcji przygotowującej jako argument funkcji testującej. Możesz również użyć obu przełączników : autouse=True i scope='module' w jednym dekoratorze. Skutek jest łatwy do przewidzenia. Nie trzeba będzie modyfikować argumentów funkcji testujących, a funkcja przygotowująca zostanie wywołana raz przed wszystkimi testami. I to jest chyba najbardziej sensowny wariant w tego typu sytuacjach. Kod całego modułu testującego będzie wyglądał ostatecznie w ten sposób:

```
import nibyDB
import pytest

@pytest.fixture(autouse=True, scope="module")
def load_stuff():
    print("\n##### load #####")
    nibyDB.loadDB()

def test_getData():
    assert len( nibyDB.getData() ) > 0
    pass

def test_getOne():
    assert nibyDB.getOne(0) [1] == 'Marian'
    pass
```

I skutek jego działania:

```
(venv) D:\data\workspace-python\testy_jednostkowe>pytest test_nibyDB.py -s -v
=====
platform win32 -- Python 3.7.2, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -- d:\data\
cachedir: .pytest_cache
rootdir: D:\data\workspace-python\testy_jednostkowe
collected 2 items

test_nibyDB.py::test_getData
##### load #####
##### ŁADOWANIE BAZY #####
PASSED
test_nibyDB.py::test_getOne PASSED
=====
```

Po przebrnięciu przez te wszystkie przykłady czas odpowiedzieć sobie na podstawowe pytanie: *Czym więc jest fikstura?* Fikstura to funkcja która przygotowuje dane, lub wykonuje czynności inicjalizacyjne na potrzeby testów.

Makiety (Mocks)

Makiety służą zastępowaniu prawdziwych danych na czas testów. Przyjrzyjmy się przykładowi takiej makiety. Stosowanie makiet samo w sobie nie jest w żaden sposób powiązane z pytest.

```
from unittest import mock
makieta=mock.Mock()
makieta.pole1=20
makieta.pole2='Element tekstowy'
print('pole1={}, pole2={}'.format(makieta.pole1,makieta.pole2))
```

W powyższym przykładzie widzimy że tworzę obiekt klasy Mock, do dwóch jego pól przypisuję a następnie wyświetlam wartości. "Też mi cuda", mógłby powiedzieć ktoś kto ma pojęcie choćby o podstawach obiektowości w Pythonie. Przecież mogę w ten sposób tworzyć pola w dowolnym obiekcie, nie potrzebuję do tego klasy Mock! I to jest jak najbardziej prawda. Popatrzmy co dalej możemy tutaj zrobić. Dynamicznie mogę tworzyć również metody. Całość:

```
from unittest import mock
makieta=mock.Mock()
makieta.pole1=20
makieta.pole2='Element tekstowy'
print('pole1={}, pole2={}'.format(makieta.pole1,makieta.pole2))

makieta.dawajPi.return_value=3.14
print(makieta.dawajPi)
```

Wynik na konsoli:

```
pole1=20, pole2=Element tekstowy
<Mock name='mock.dawajPi' id='2765903240720'>
```

Process finished with exit code 0

Wyświetlenie zawartości dynamicznie tworzonych pól nikogo nie zaskakuje, to już ustaliliśmy. W ostatnich dwóch liniijkach odwołuję się jednak do czegoś co się nazywa "dawajPi". Jest to funkcja którą dynamicznie tworzę dla obiektu makiety. Z pomocą linii:

```
makieta.dawajPi.return_value=3.14
```

Tworzę i deklaruję zwracaną wartość tejże funkcji. Takie obiekty możesz teraz użyć do testów, w miejsce danych pobieranych np. z bazy w której w ramach testów nie chcielibyśmy mieszać.

Dane testowe

Testy fajnie się pisze jeśli musisz przetestować funkcję dla kilku wartości. Kilku Janów Kowalskich czy Nowaków zawsze się wymyśli. Co jednak gdy chodzi o różne ciągi tekstowe (tu pewnie zwykle następuje "dfgfdgdsdgsdfg" :D) czy daty, albo choćby i te wspomniane nazwiska - jednak ilościowo idące w tysiące? Warto wiedzieć że dla Pythona dostępna jest ciekawa biblioteka "Faker". Pozwala ona generować takie właśnie losowe dane. Poniżej przykład:

```
from unittest import mock
import faker

m=mock.Mock()
f=faker.Faker()

m.losowaOsoba=f.name()
m.losowaSentencja=f.sentence()
m.losowaData=f.date()

print(m.losowaOsoba)
print(m.losowaSentencja)
print(m.losowaData)
```

Możesz powyższy tekst skopiować i uruchomić u siebie. Dane są losowe, za każdym razem będzie to coś innego. U mnie np. wyświetliło:

Brooke Glover

Light during throughout receive.

1981-08-08

Sprawdzanie pokrycia kodu testami

Warto jest sprawdzać stopień pokrycia kodu testami. Możemy dzięki temu sprawdzić które funkcje czy moduły nie zostały jeszcze przetestowane i wymagają dorobienia testów. Z jego użyciem możemy też wykryć nieużywane fragmenty kodu który jako martwy możemy usunąć. Aby rozpocząć pracę musimy zainstalować wtyczkę "pytest-cov" do pytest'a. Robimy to z poziomu pip'a:

pip install pytest-cov

Od tego momentu do pytest możemy dodawać przełącznik "--cov", dzięki któremu możemy przetestować wskazany moduł lub cały projekt pod kątem pokrycia testami. Poerwszy wariant - dla całego projektu, drugi dla wybranego modułu:

pytest --cov

pytest test_modulik.py --cov

Wynik działania:

```
(venv) D:\data\workspace-python\testy_jednostkowe>pytest test_modulik.py --cov
=====
platform win32 -- Python 3.7.2, pytest-5.0.1, py-1.8.0, pluggy-0.12.0
rootdir: D:\data\workspace-python\testy_jednostkowe
plugins: cov-2.7.1
collected 4 items

test_modulik.py ....

----- coverage: platform win32, python 3.7.2-final-0 -----
Name                                                    Stmts   Miss  Cover
-----
modulik.py                                              6       0   100%
test_modulik.py                                         7       0   100%
```

Istnieje również możliwość generowania raportów pokrycia testami w formacie html. Trzeba tylko użyć dodatkowego przełącznika --cov-report:

pytest test_modulik.py --cov --cov-report=html

Po uruchomieniu testów w ten sposób, w katalogu projektu (lub miejscu w którym uruchamiałeś testy) powstaje katalog "htmlcov" pełen raportów w formacie html. Do modułu "modulik.py" dodałem umyślnie jedną nic nie robiącą funkcję. Generuję raporty w formacie html. Uruchamiam znajdujący się w katalogu "htmlcov" plik "index.html":

Module ↓	statements	missing	excluded	coverage
modulik.py	9	2	0	78%
test_modulik.py	7	0	0	100%

Widzę że "modulik" nie jest w 100% pokryty testami. Klikam więc na jego nazwę (która jest linkiem), by zobaczyć co nie jest pokryte testami i oto otrzymuję wynik:

Coverage for modulik.py : 78%
9 statements 7 run 2 missing 0 excluded

```
1 | podpietaBaza=None
2 |
3 | def podepnijBaze(nazwa):
4 |     global podpietaBaza
5 |     podpietaBaza=nazwa
6 |
7 | def funkcjaKtoraRobiNic():
8 |     print('siema, jestem martwym kodem!')
9 |     pass
10 |
11 | def wykonajZapytanie():
12 |     global podpietaBaza
13 |     print('Wykonuję zapytanie z użyciem bazy {}'.format(podpietaBaza))
14 |     # if(podpietaBaza=='MS SQL'):
15 |     #     raise Exception('FUUUUUUU')
16 |     return "ok"
```

« index coverage.py v4.5.4, created at 2019-08-07 12:36

Wizualizacja danych

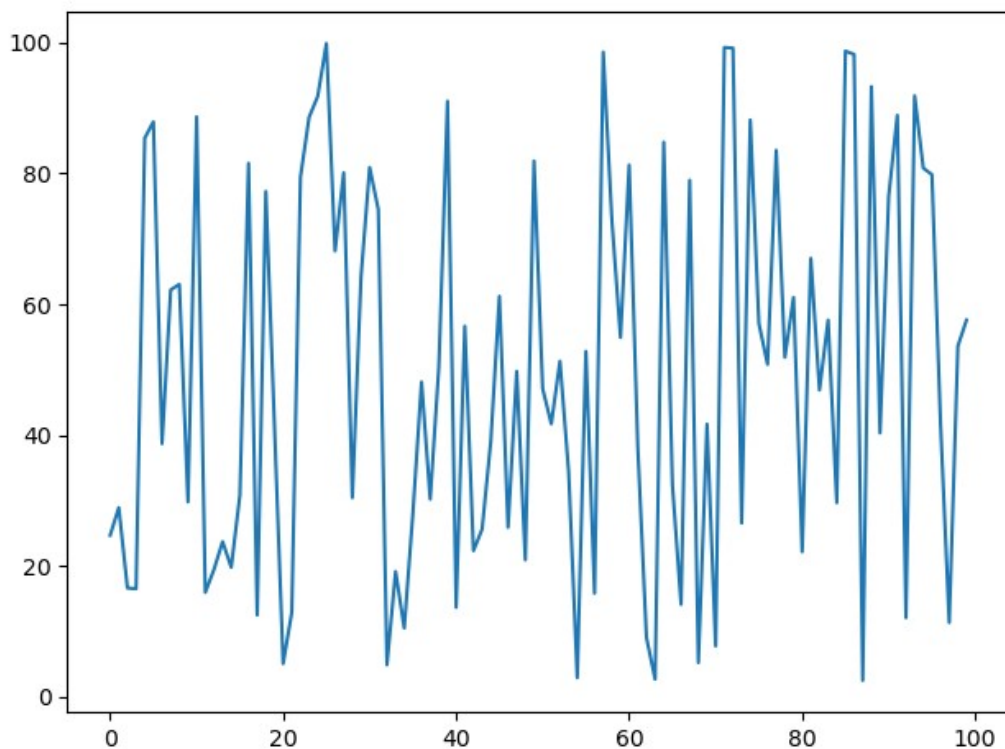
W tym rozdziale zajmiemy się wizualizacją danych w formie wykresów. Umiejętność ta może się przydać zarówno analitykom danych, jak i administratorom.

Pierwszy wykres liniowy

Zacznijmy od narysowania prostego wykresu liniowego, opartego o 100 losowych wartości z zakresu 0-100.

```
import matplotlib.pyplot as plt
from random import random
y=[random()*100 for e in range(100)]
plt.plot(y)
plt.show()
```

W pierwszej kolejności należy zaimportować moduł pyplot, co czynimy w pierwszej linii. Import funkcji random nie jest związany z samym rysowaniem wykresów, wykorzystujemy ją tylko do generowania losowych wartości na potrzeby wykresu. Robimy to w linii trzeciej, tworząc listę losowych wartości z zakresu 0-100. Wartości używane w wykresach mogą być zarówno całkowite jak i zmiennoprzecinkowe. Taką listę przekazujemy do funkcji plot która rysuje linię. Wykres jednak nie pokaże się dopóki nie wywołamy funkcji show() na końcu. Poniżej efekt działania powyższego kodu:

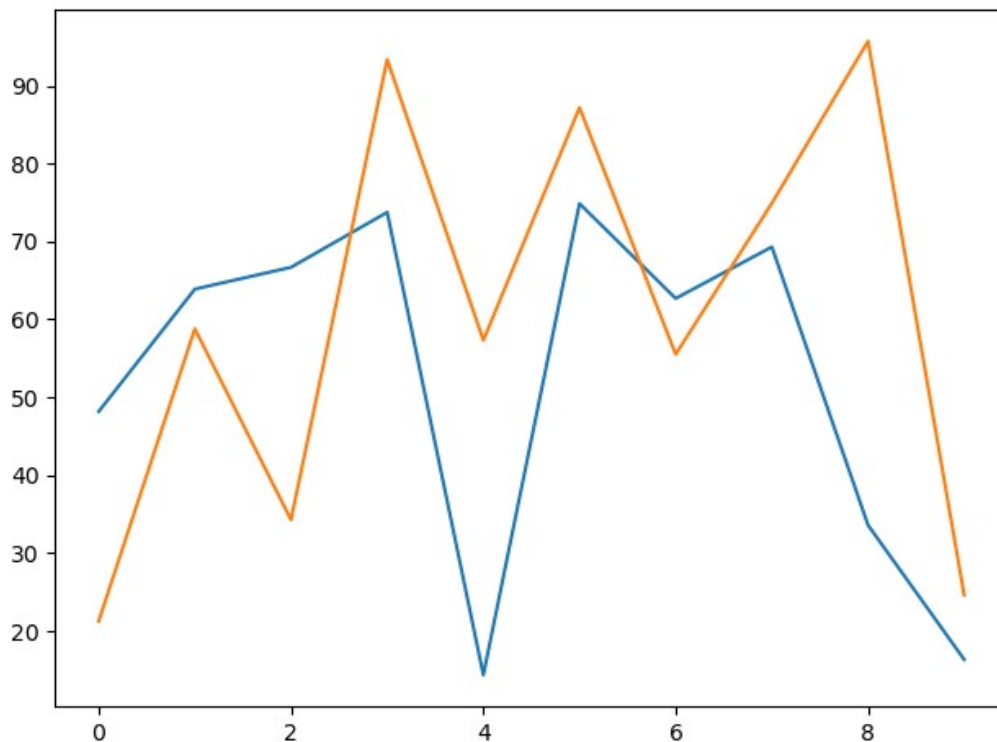


Nanoszenie dodatkowych serii

Często zdarza się, że musimy nałożyć kilka serii (linii) na siebie, choćby w celach porównawczych. Wystarczy ponownie wywołać funkcję plot podając kolejną serię danych:

```
import matplotlib.pyplot as plt
from random import random
y=[random()*100 for e in range(10)]
z=[random()*100 for e in range(10)]
plt.plot(y)
plt.plot(z)
plt.show()
```

Powyższy kod to nieco zmodyfikowany poprzedni przykład. Dodałem nową serię „z”, a następnie przekazałem ją do kolejnego wywołania funkcji plot. Efekt działania:

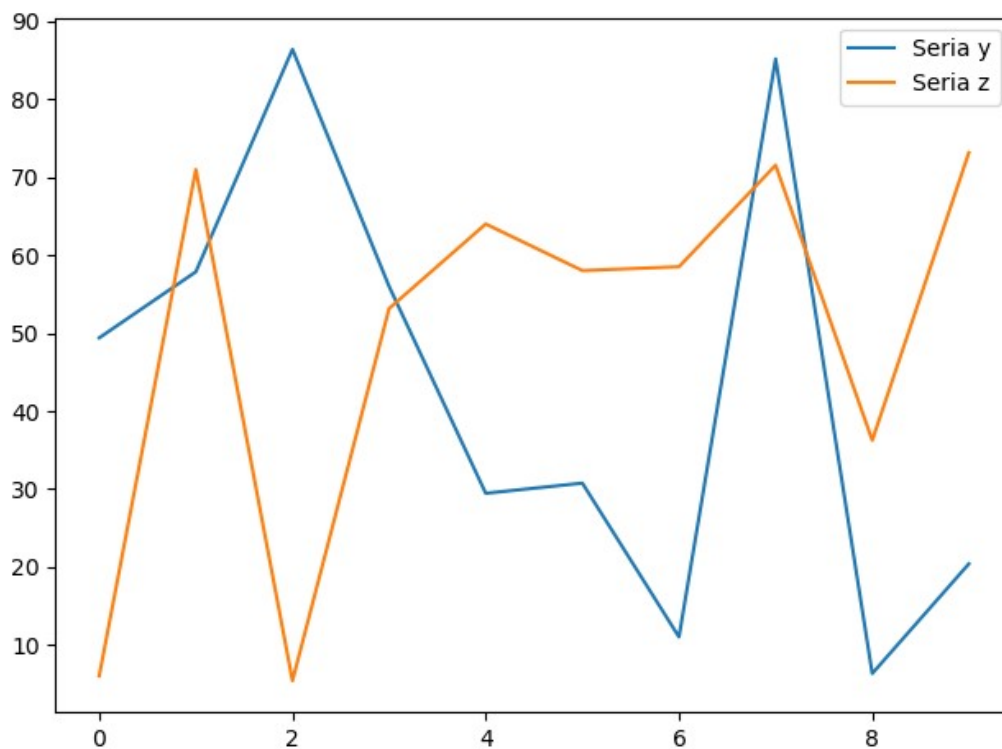


Dodawanie legendy do wykresu

Aby dodać legendę do wykresu trzeba zrobić dwie rzeczy. Po pierwsze nadać nazwy seriom, po drugie wywołać funkcję `legend()`:

```
import matplotlib.pyplot as plt
from random import random
y=[random()*100 for e in range(10)]
z=[random()*100 for e in range(10)]
plt.plot(y,label='Seria y')
plt.plot(z,label='Seria z')
plt.legend()
plt.show()
```

Jeśli przy części serii nie podamy etykiety, nie zostanie ona uwzględniona w legendzie. Wynik działania:

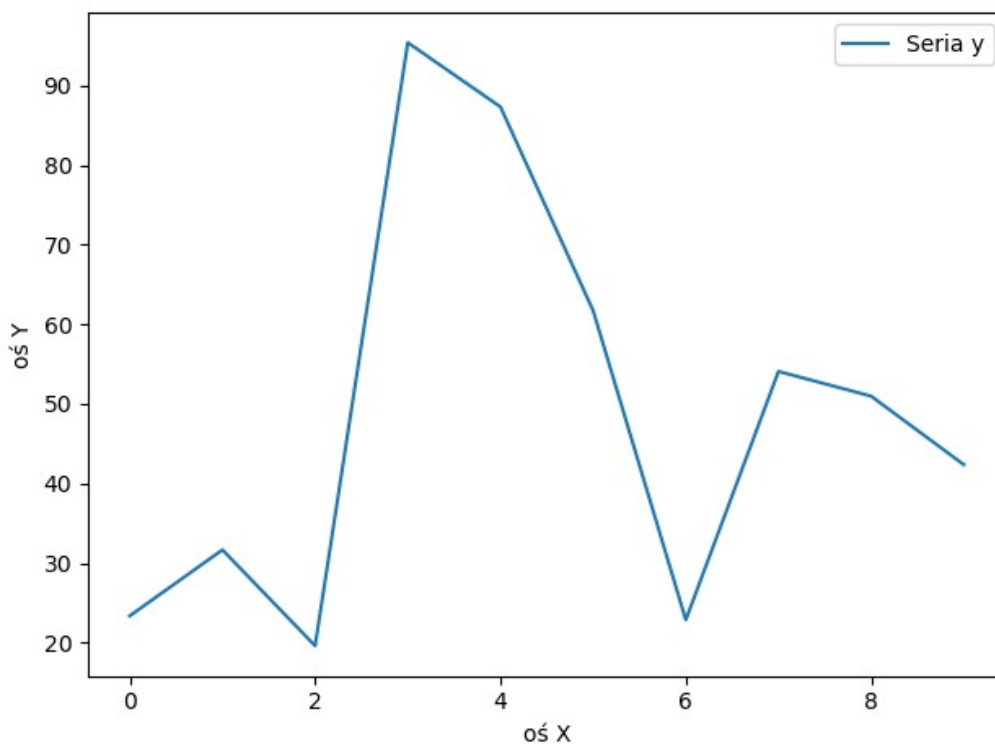


Etykiety osi X i Y

Aby ustawić etykiety osi x i y, wystarczy wywołać odpowiednio funkcje xlabel i ylabel:

```
import matplotlib.pyplot as plt
from random import random
y=[random()*100 for e in range(10)]
plt.plot(y,label='Seria y')
plt.xlabel('oś X')
plt.ylabel('oś Y')
plt.legend()
plt.show()
```

Efekt działania:

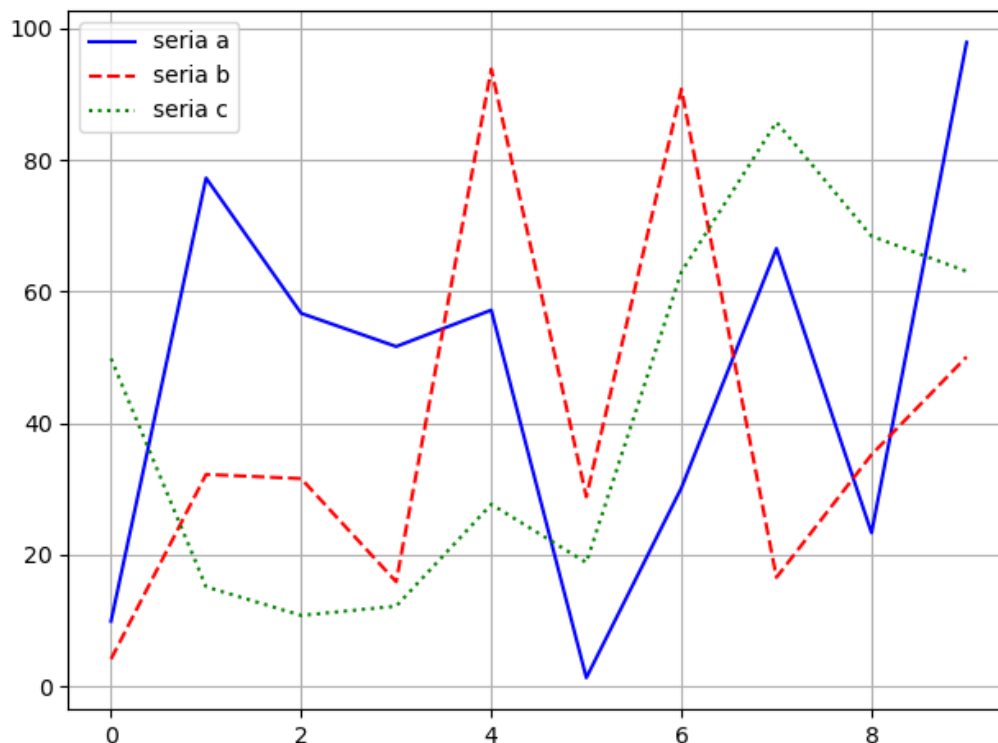


Zmiana rodzaju i koloru linii

Funkcjonalność bardzo przydatna zwłaszcza gdy zechcesz nanieść na wykres kilka serii – łatwiej będzie Ci odróżnić.

```
import matplotlib.pyplot as plt
from random import random
a=[random()*100 for e in range(10)]
b=[random()*100 for e in range(10)]
c=[random()*100 for e in range(10)]
plt.plot(a,'b-',label='seria a')
plt.plot(b,'r--',label='seria b')
plt.plot(c,'g:',label='seria c')
plt.legend()
plt.grid()
plt.show()
```

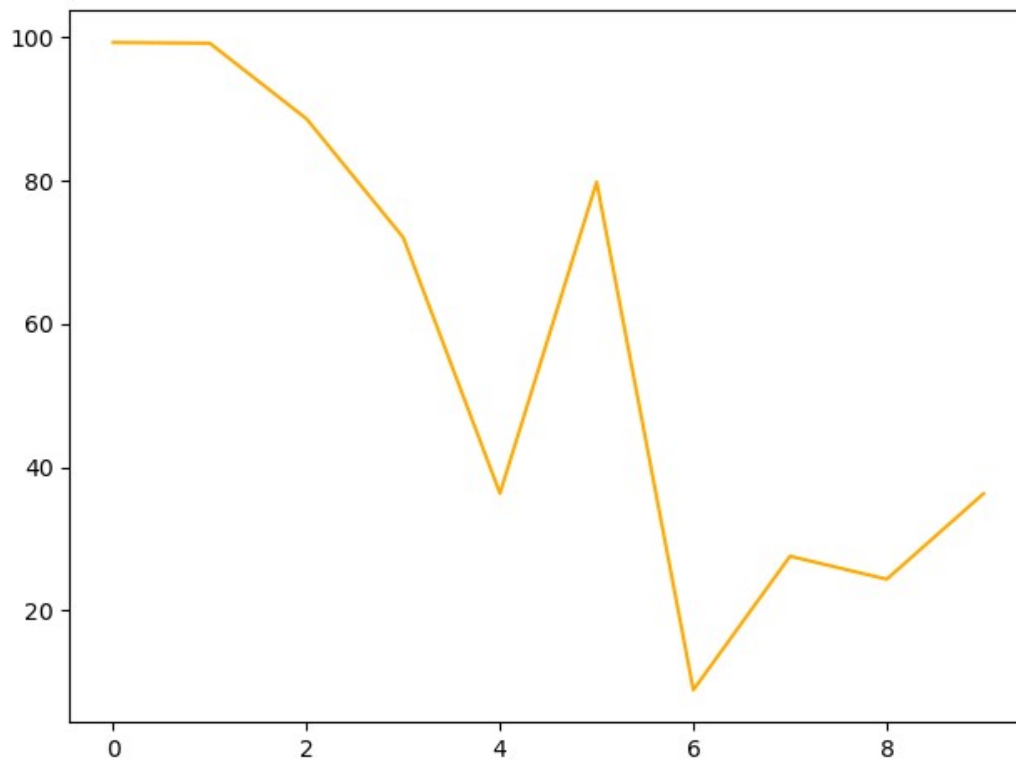
W powyższym fragmencie kodu pokazuję zmianę koloru i stylu linii w oparciu o pozycję argumentu. W tym przypadku zmiana koloru i stylu linii odbywa się przez drugi argument funkcji plot. Pierwszy znak – odpowiednio b,r,g, odnosi się do koloru (blue, red, green). Powinno być rgb (hermetyczny żarcik 😊). Następujące po literkach znaki odnoszą się do rodzaju linii – odpowiednio linia, przerywana linia i kropki. Wariantów jest oczywiście więcej i tu w myśl rtfm odsyłam do dokumentacji 😊. Dodałem też wartość dla argumentu „label”, by pokazać że można stosować te ustawienia jednocześnie.



Kolory serii możesz zmieniać również korzystając podając w drugim argumencie kod koloru hexem:

```
import matplotlib.pyplot as plt
from random import random
a=[random()*100 for e in range(10)]
plt.plot(a, '#FFAA00')
plt.show()
```

Efekt:



Możesz też użyć argumentu o nazwie „color”:

```
import matplotlib.pyplot as plt
from random import random
a=[random()*100 for e in range(10)]
plt.plot(a,color='#FFAABB')
plt.show()
```

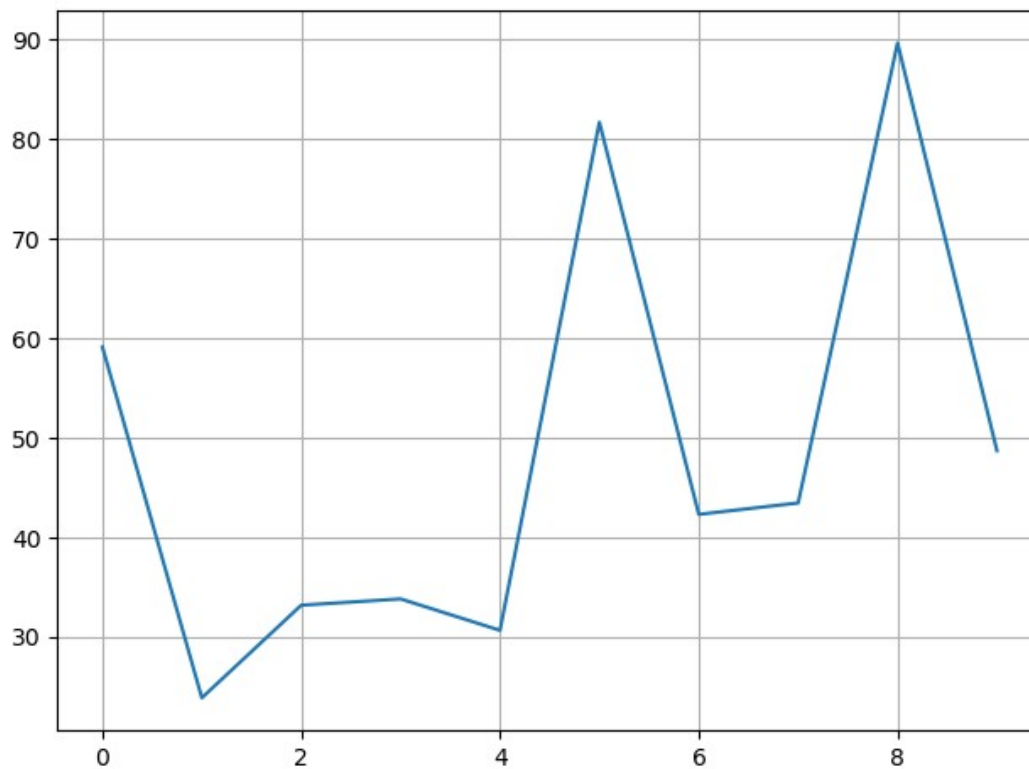
W którym możesz podać też nazwę koloru – red, blue etc.

Siatka na wykresie

Wywołujemy funkcję grid():

```
import matplotlib.pyplot as plt
from random import random
y=[random()*100 for e in range(10)]
plt.plot(y)
plt.grid()
plt.show()
```

Skutek:



Zapisywanie wykresu do pliku

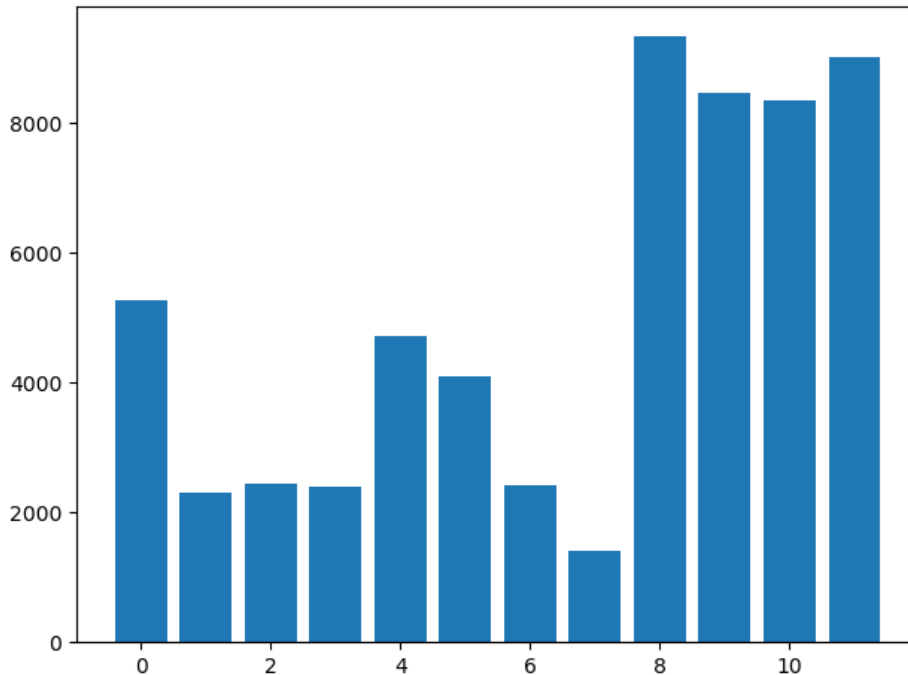
Każdy stworzony wykres można zapisać do pliku graficznego. Sprowadza się to do wywołania funkcji `savefig`. Nazwę pliku podajemy z rozszerzeniem, funkcja sama dopasuje odpowiedni format zapisu. Plik zostanie umieszczony w tym przypadku wewnątrz projektu – użyłem samej nazwy, ale możesz również podać ścieżkę bezwzględną:

```
import matplotlib.pyplot as plt
from random import random
a=[random()*100 for e in range(10)]
plt.plot(a)
plt.savefig('wykres.png')
plt.show()
```


Wykresy słupkowe

Generowanie wykresów słupkowych wygląda bardzo podobnie do generowania wykresów liniowych, z drobnymi zmianami. Dla wykresów słupkowych działają te same funkcjonalności – siatka, zapisywanie wykresu do pliku, legenda etc w taki sam sposób jak dla wykresów liniowych.

```
import matplotlib.pyplot as plt
from random import random
x=list(range(12))
y=[random()*10000 for e in range(12)]
plt.bar(x,y)
plt.show()
```

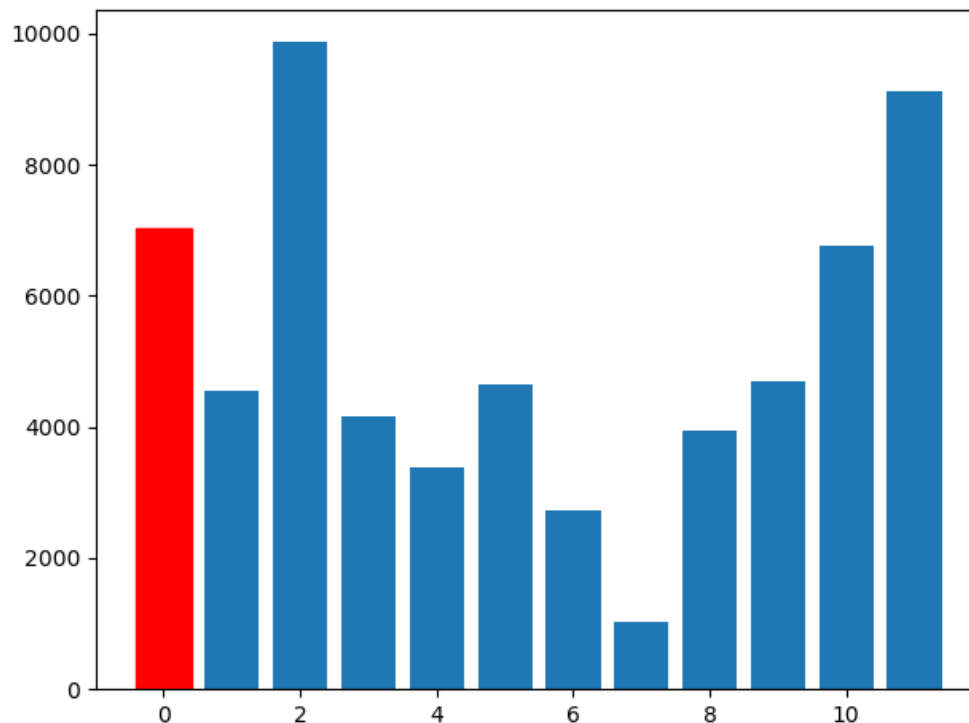


Aby stworzyć wykres słupkowy, zamiast funkcji plot używamy funkcji bar. Pojawia się też dosyć enigmatyczny argument x – w wykresach liniowych ładowaliśmy tylko dane dla serii. W przypadku wykresów słupkowych należy podać dwie listy – jedna zawierająca wartości dla osi X, druga wartości liczbowe dla których mają powstać słupki. Ilość elementów w obu listach musi być taka sama.

Zmiana koloru słupków

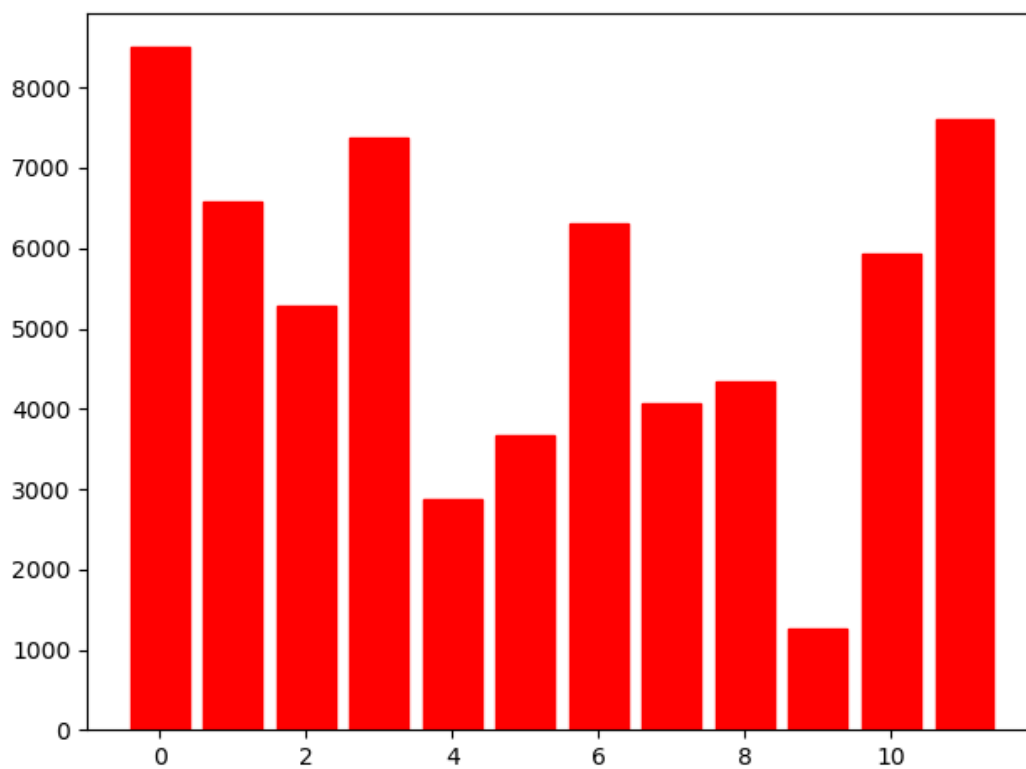
Aby zmienić kolor słupków musimy z funkcji bar odebrać handler dla wykresu. W kolejnym kroku ustawiamy kolor (hexadecymalnie albo literalnie). Możemy zmieniać kolory słupków pozycyjnie, co prezentuję na poniższym przykładzie. Zmieniam kolor pierwszego słupka (wykres[0]).

```
import matplotlib.pyplot as plt
from random import random
x=list(range(12))
y=[random()*10000 for e in range(12)]
wykres=plt.bar(x,y)
wykres[0].set_color('r')
plt.show()
```



Jeśli zechcemy zmienić na taki sam kolor wszystkich słupków, musimy posłużyć się iteracją:

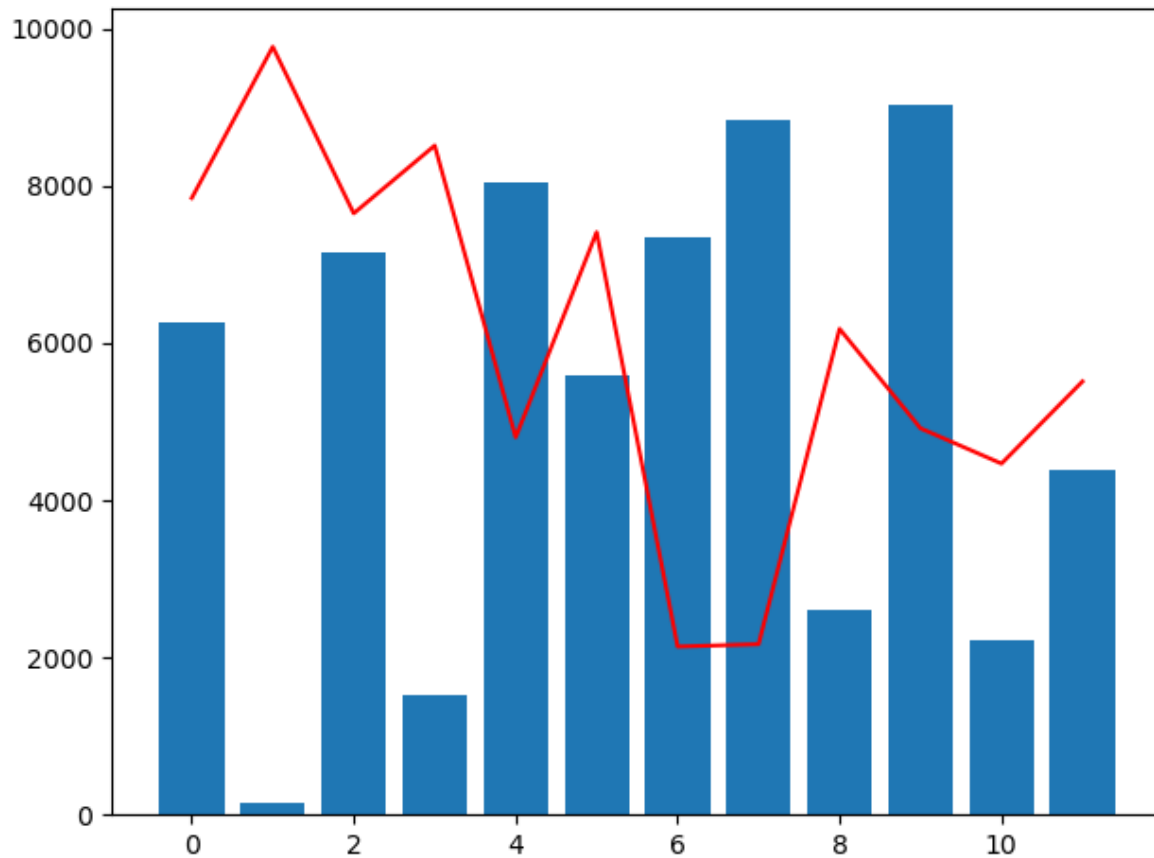
```
import matplotlib.pyplot as plt
from random import random
x=list(range(12))
y=[random()*10000 for e in range(12)]
wykres=plt.bar(x,y)
for i in range(len(wykres)):
    wykres[i].set_color('r')
plt.show()
```



Nakładanie serii słupkowych i liniowych na siebie

Aby uzyskać na jednym wykresie zarówno słupki jak i linie, wykorzystujemy funkcję `plot` i funkcję `bar` jednocześnie:

```
import matplotlib.pyplot as plt
from random import random
x=list(range(12))
y=[random()*10000 for e in range(12)]
z=[random()*10000 for e in range(12)]
plt.bar(x,y)
plt.plot(z,'r')
plt.show()
```



Obiektość w Pythonie

Klasy pozwalają nam deklarować własne obiekty złożone posiadające wbudowane pola a także funkcje.

Deklaracja klasy i pola

Klasy mogą posiadać pola, funkcje, konstruktory. Aby powołać do życia obiekt i korzystać z tych wszystkich dobrodziejstw, trzeba najpierw zadeklarować klasę. Klasę dodajemy w zasadzie w dowolnym miejscu pliku, jednak korzystać z niej będziemy mogli tylko w liniach znajdujących się pod jej deklaracją. Z tego też powodu najlepiej deklarować klasy na początku pliku (lub w ogóle w osobnym pliku który następnie będziesz importować). Elementy związane z klasą rozpoznać można po ich odsunięciu od lewej krawędzi, podobnie jak elementy związane z pętlą czy instrukcjami warunkowymi. W pierwszej kolejności zadeklarujemy prostą klasę posiadającą jedynie pola:

```
class Osoba:
    imie='Andrzej'
    nazwisko='Klusiewicz'
    wiek=33
    pustePole=None
```

Obiekty powyższej klasy będą posiadały 4 pola. Każdy nowo powołany do życia obiekt tej klasy będzie posiadał już przypisane wartości do 3 z 4 pól. Jeśli chesz by do jakiegoś pola nie zostało przypisane nic, wystarczy przypisać "None".

Aby stworzyć obiekt takiej klasy używamy tak zwanego konstruktora:

```
o=Osoba()
print(o.imie,o.nazwisko,o.wiek, o.pustePole)
```

Po stworzeniu obiektu "o" klasy Osoba, wypisuje zawartość jego pól. Domyślnie możemy sięgać do wszystkich pól, czyli jest to odpowiednik "public" znanego Ci drogi czytelniku być może z języków Java czy C#. Efektem działania powyższego kodu jest następujący tekst na konsoli:

Andrzej Klusiewicz 33 None

Klasę możesz umieścić również w innym module i zwyczajnie zaimportować:

```
from inna import Osoba
o=Osoba()
print(o.imie,o.nazwisko,o.wiek, o.pustePole)
```

Zawartość pól w obiektach można oczywiście również podmieniać:

```
class Osoba:
    imie='Andrzej'
    nazwisko='Klusiewicz'
    wiek=33
    pustePole=None

o=Osoba()
o.imie='Krzysztof'
o.nazwisko='Jarzyna'
print(o.imie,o.nazwisko)
```

W polach imie i nazwisko obiektu o znajdowały się dotychczas dane autora niniejszego kursu, aktualnie zostały one zastąpione przez Krzysztofa Jarzynę (ze Szczecina ;)). Po uruchomieniu powyższego kodu zostaje wyświetlone już nowe imię i nazwisko:

Krzysztof Jarzyna

Pewnym zaskoczeniem dla programistów innych języków może być wynik działania poniższego kodu:

```
class Osoba:
    imie=None
    nazwisko=None
    ksywka=None
o1=Osoba()
o1.imie='Jerzy'
o1.nazwisko='Kiler'
o1.ksywka='Killer'
o2=Osoba()
o2.imie='Stefan'
o2.nazwisko='Siarzewski'
o2.ksywka='Siara' #i wszystko jasne
Osoba.imie='zamienione...'
print(o1.imie,o2.imie)
o3=Osoba()
print(o3.imie)
```

I tu pytanie do programistów - jakiego wyniku się spodziewacie? Można by przypuszczać że pole "imie" jest z jakiegoś powodu statyczne skoro można przypisać wartość na zasadzie "Klasa.pole=xxx", ale nie bo przecież przypisujemy wartość do pola obiektu również tak "obiekt.pole=xxx" w odniesieniu do tego samego pola... Taki unikatowy przypadek w Pythonie :) Najpierw zobaczmy co pojawia się na konsoli:

Jerzy Stefan

zamienione...

Teraz o co chodzi - od momentu przypisania wartości niejako do pola klasy, wszystkie obiekty tej klasy stworzonego od tego momentu (do końca pliku OFC), będą miały nową wartość domyślną w tym polu.

Sprawa powinna się troszkę przejaśnić gdy uruchomisz to:

```
class Osoba:
    imie='Andrzej'
print(Osoba.imie)
```

W wyniku działania tego kodu na konsoli zobaczysz:

Andrzej

Krótko mówiąc, takie odniesienie "Klasa.pole" oznacza odwołanie do wartości domyślnej tego pola. Za mało namieszane? To przyjrzyjmy się temu:

```
class Osoba:
    imie=None
    nazwisko=None
    ksywka=None
o1=Osoba()
o1.imie='Jerzy'
o1.nazwisko='Kiler'
o1.ksywka='Killer'
o1.wtf='omg! '

print(o1.imie,o1.nazwisko,o1.ksywka,o1.wtf)
```

Czy mogę przypisywać coś do pola którego nie deklarowałem? Czy program zadziała jeśli odwołam się do pola którego nie ma (nie ma ??)? Robię to nawet dwukrotnie - raz przypisuję wartość, a raz się do niej odwołuję. Wynik działania:

Jerzy Kiler Killer omg!

Sponsorem Twojego zdziwienia jest elastyczna składnia języka Python :) A teraz o co chodzi - w chwili przypisania wartości do "nieistniejącego" pola w obiekcie, takie pole po prostu w nim powstało. W tym jednak przypadku będzie to dotyczyło wyłącznie tego obiektu, a nie innych - niezależnie od tego czy zadeklarowanych przed czy po.

Funkcje w klasie

Dotychczas wielokrotnie wyświetlaliśmy wszystkie pola obiektu, za każdym razem odwołując się do każdego z pól osobno. Czy nie byłoby wygodniej gdybyśmy mieli jakąś wbudowaną w obiekt funkcję która by wyświetlała wszystkie jego pola?

```
class Osoba:
    imie='Andrzej'
    nazwisko='Klusiewicz'
    wiek=33
    def wypiszMnie(self):
        print(self.imie, self.nazwisko, self.wiek)

o=Osoba()
o.wypiszMnie()
```

Wynik działania:

Andrzej Klusiewicz 33

Funkcje mogą również przyjmować argumenty:

```
class Witacz:
    def przywitaj(self, imie, nazwisko):
        print("Witaj "+imie+" "+nazwisko+"! I kto jest teraz
debeściak?")

w=Witacz()
w.przywitaj("Jerzy", "Ryba")
```

Tradycyjnie wynik działania kodu:

Witaj Jerzy Ryba! I kto jest teraz debeściak?

Struktura funkcji nie różni się tu jakoś specjalnie od struktury funkcji jakie znaliśmy do tej pory. Jediną różnicą jest pojawienie się argumentu "self". To jest wskaźnik do obiektu w którym się

znajdujemy i pozwala np odwoływać się do jego pól. Jeśli zechcemy odwołać się do takich pól to robimy to tak:

```
class Programista:
    jezyk="Python"
    def funkcja(self):
        print('programuję w języku '+self.jezyk)

p=Programista()
p.funkcja()
```

A teraz dla odmiany nie będzie wyniku działania :) Powinieneś już wiedzieć co się stanie gdy uruchomisz ten kod.

Funkcja może również coś zwracać :

```
class Polska:
    def roszczenie(self):
        return "Mienie bezspadkowe"

p=Polska()
print(p.roszczenie())
```

Porównywanie obiektów tej samej klasy

Sprawdźmy działanie poniższego kodu:

```
class Owoc:
    pass

jablko=Owoc()
pomarancza=Owoc()
print(jablko)
print(pomarancza)

if(jablko==pomarancza):
    print('to samo')
else:
    print('inne')
```

Zadeklarowałem pustą klasę Owoc. Klasa ta nie posiada żadnych pól ani metod. Stworzyłem dwa obiekty tej klasy. Wypisuję je na konsoli i sprawdzam czy są sobie równe. Dla programistów innych języków obiektowych zaskoczenia pewnie nie będzie:

```
<__main__.Owoc object at 0x000001CC289333C8>
```

```
<__main__.Owoc object at 0x000001CC28933278>
```

```
inne
```

Ponieważ obiekty te nie posiadają żadnych pól, toteż w porównaniu nie może być brana zawartość obiektów. Jeśli porównujesz dwa obiekty w taki sposób, w rzeczywistości sprawdzasz czy chodzi o ten sam obiekt - w skrócie czy oba obiekty odnoszą się do tego samego adresu w pamięci operacyjnej. Oczywiście tak nie jest, dlatego kod zwraca nam informację że obiekty są różne.

Przeciążanie funkcji

Niet! W Pythonie przeciążanie funkcji nie bangła. Dla osób które nie wiedzą czym jest przeciążanie - chodzi o możliwość posiadania kilku funkcji o takiej samej nazwie, a różniących się tylko ilością (ewentualnie typem) parametrów. W innych językach programowania w takiej sytuacji to która metoda zostałaby wywołana zależałoby od podanych do niej parametrów. W Pythonie jest to zrobione zupełnie inaczej. Przeciążanie nie działa, każda kolejna funkcja o tej samej nazwie przesłania jej poprzednie wystąpienie. Sprawdźmy to w praktyce:

```
class Witacz:
    def powitaj(self):
        print("No siemka!")
    def powitaj(self,kogo):
        print("No siemka {}!".format(kogo))
w=Witacz()
w.powitaj()
```

Wywołajmy kod:

Traceback (most recent call last):

File "D:/data/workspace-python/nauka/materialy/02.py", line 7, in <module>

w.powitaj()

TypeError: powitaj() missing 1 required positional argument: 'kogo'

Python krzyczy na mnie, ponieważ spodziewa się otrzymania parametru "kogo" - a tego nie podałem. Ta druga funkcja "powitaj" z dodatkowym parametrem zasłoniła jej poprzednią bezparametrową wersję. Kod zadziała poprawnie jeśli odwrócimy kolejność deklaracji funkcji:

```
class Witacz:
    def powitaj(self,kogo):
        print("No siemka {}!".format(kogo))
    def powitaj(self):
        print("No siemka!")
w=Witacz()
w.powitaj()
```

Metody specjalne

W klasach Pythona możemy deklarować metody specjalne - czyli takie które spełniają specjalną funkcję. Umożliwiają one na przykład określenie na przykład co ma się wydarzyć gdy dodasz do siebie dwa obiekty (przeciążanie operatorów), gdy zechcesz je zamienić na ciąg tekstowy, czy co ma się stać gdy obiekt jest tworzony. Wszystkie funkcje specjalne zaczynają się i kończą podwójnym znakiem "__".

`__init__`

Funkcja wywoływana automatycznie przy tworzeniu obiektu. Pozwala ona na inicjowanie np atrybutów tworzonego obiektu. Programistom innych języków może się to skojarzyć z konstruktorami, słusznie - choć funkcja "`__init__`" konstruktorem nie jest. Może być wykorzystywany jako konstruktor, ale warto pamiętać że funkcja "`__init__`" wywoływana jest już PO stworzeniu obiektu. Taka drobna różnica, ale powodująca że ta funkcja nie jest konstruktorem formalnie.

```
class Samochod:
    def __init__(self):
        print('to jest funkcja init!')

s = Samochod()
```

Po uruchomieniu powyższego kodu, na konsoli zostaje oczywiście wypisane "to jest funkcja init!".

```
class Samochod:
    marka=None
    model=None
    def __init__(self):
        self.marka="nie podano"
        self.model="nie podano"

s = Samochod()
print(s.marka,s.model)
```

Tym razem moja klasa ma dwa pola: "marka" i "model". Wewnątrz funkcji "`__init__`" inicjalizuje wartości tych pól. Po stworzeniu obiektu i wypisaniu zawartości pól na konsoli widzimy że pola przybrały wartości określone w funkcji "`__init__`".

"`__init__`" musi otrzymywać przez parametr obiekt reprezentujący "siebie". Nie musi się on nazywać akurat "self", możesz go nazwać choćby "ja". Jest on potrzebny by móc się odnosić do atrybutów i

funkcji obiektu. Podajemy go tylko na etapie deklaracji funkcji "__init__", nie przekazujemy przy wywołaniu konstruktora (s=Samochod()), jest on podstawiany automagicznie.

Funkcja "__init__" może też opcjonalnie przyjmować parametry:

```
class Samochod:
    marka=None
    model=None
    def __init__(self,marka,model):
        self.marka=marka
        self.model=model

s = Samochod("Renault","Kadjar")
print(s.marka,s.model)
```

na konsoli zostaje wypisane:

Renault Kadjar

Do funkcji "__init__" dodałem teraz dwa dodatkowe parametry: "marka" i "model". Zwróć uwagę na zawartość tej funkcji. Ilekroć odnoszę się na zasadzie "marka" we wnętrzu funkcji - mam na myśli parametr funkcji. Gdy odnoszę się do pola w klasie - "self.marka". W związku z tym, że nowa wersja "__init__" oczekuje teraz podania 2 parametrów, to tworząc obiekt klasy Samochód muszę te wartości podać:

```
s = Samochod("Renault","Kadjar")
```

Programiści innych języków zapewne zadadzą sobie teraz pytanie - "a, czyli mogę zadeklarować jedną funkcję bez parametrów, a drugą z, by później tworząc obiekt podawać dane albo nie?". Otóż nie...

```
class Samochod:
    marka=None
    model=None
    def __init__(self):
        print("to się nie zadzieje")
    def __init__(self,marka,model):
        self.marka=marka
        self.model=model

s = Samochod("Renault", "Kadjar")
print(s.marka,s.model)
s2=Samochod()
print(s2.marka,s2.model)
```

Zrobiłem dwie funkcje "__init__", jedną z parametrami a drugą bez. W Javie czy C# oczekiwilibyśmy przeciążania konstruktora - czyli możliwości stworzenia obiektu podając albo nie podając wartości parametrów, stosownie wywołując jeden albo drugi konstruktor. Nie w Pythonie, tutaj nie funkcjonuje przeciążanie funkcji czy konstruktorów. Po prostu każda kolejna definicja funkcji o nazwie istniejącej już nad nią funkcji skończy się przesłonięciem (a nie przeciążeniem!) tej pierwszej. Sprawdźmy więc:

Renault Kadjar

Traceback (most recent call last):

File "D:/data/workspace-python/nauka/materialy/02.py", line 12, in <module>

s2=Samochod()

TypeError: __init__() missing 2 required positional arguments: 'marka' and 'model'

Python pokrzyczał na mnie że nie podałem mu parametrów dla "__init__" w linii 12 czyli w :

```
s2=Samochod()
```

Dzieje się tak za sprawą opisanego przed momentem mechanizmu.

`__str__`

To jest funkcja będąca odpowiednikiem metody "toString" występującej w Javie czy C#. Jej zadaniem jest zwracanie obiektu w postaci tekstowej.

Domyślnie próba wypisania obiektu zakończy się w ten sposób:

```
class Samochod:
    marka=None
    model=None

s=Samochod()
print(s)
```

Dostajemy:

```
<__main__.Samochod object at 0x000001E44AF724E0>
```

W zasadzie wolałbym by wyświetlane były zawartości pól - na potrzeby na przykład sprawdzenia jakie dane znajdują się na którym etapie przetwarzania skryptu. To jest właśnie miejsce na pojawienie się przesłonięcia funkcji "___str___".

```
class Samochod:
    marka=None
    model=None
    def __str__(self):
        return "marka={} model={}".format(self.marka,self.model)

s=Samochod()
print(s)
```

Tym razem dostajemy:

```
marka=None model=None
```


`__add__` i przeciążanie operatorów

Tym razem od praktyki do teorii. Przeanalizujmy poniższy kod:

```
class Samochod:
    marka=None
    model=None
    def __init__(self,marka,model):
        self.marka=marka
        self.model=model
    def __str__(self):
        return "marka={} model={}".format(self.marka,self.model)

s=Samochod("Audi","A4")
print(s)
s2=Samochod("Bmw","e46")
s3=s+s2
print(s3)
```

Nic nowego merytorycznie tutaj nie ma. Sparametryzowana funkcja "`__init__`" pozwalająca mi tworzyć obiekty podając jednocześnie parametry których wartości trafiają do pól, do tego funkcja "`__str__`" pozwalająca mi odebrać obiekt w postaci tekstowej. Tworzę dwa obiekty klasy "Samochód". W przedostatniej linii próbuję je do siebie dodać i wypisać. Uruchomienie tego kodu kończy się błędem:

marka=Audi model=A4

Traceback (most recent call last):

File "D:/data/workspace-python/nauka/materialy/02.py", line 13, in <module>

s3=s+s2

TypeError: unsupported operand type(s) for +: 'Samochod' and 'Samochod'

Zaskoczenia pewnie nie ma, trudno sobie wyobrazić dodanie do siebie dwóch samochodów. Co jednak jeśli stworzylibyśmy hipotetyczną klasę "Zamówienie" i chcielibyśmy umożliwić dodawanie do siebie i łączenie 2 zamówień? Wystarczy zaimplementować funkcję specjalną "`__add__`". Przerobię powyższy przykładowy kod klasy Samochód w celu pokazania tego.

```

class Samochod:
    marka=None
    model=None
    def __add__(self, other):
        return "Kraksa 2 Sebastianów w dresie. Jeden jechał {}, drugi
{}".format(self,other)
    def __init__(self,marka,model):
        self.marka=marka
        self.model=model
    def __str__(self):
        return "marka={} model={}".format(self.marka,self.model)

s=Samochod("Audi", "A4")
print(s)
s2=Samochod("Bmw", "e46")
s3=s+s2
print(s3)

```

Jedyne co uległo tu zmianie, to pojawiła się funkcja "__add__" która instruuje Pythona co ma się dziać gdy ktoś spróbuje dodać do siebie dwa obiekty klasy Samochod. Ja zwracam akurat ciąg tekstowy, ale równie dobrze mógłby to być jakiś obiekt innej niż "str" klasy. Skutek działania tak przerobionego kodu:

```

marka=Audi model=A4
Kraksa 2 Sebastianów w dresie. Jeden jechał marka=Audi model=A4, drugi marka=Bmw
model=e46

```

`__getitem__` i `__setitem__`

Funkcje `__getitem__` i `__setitem__` umożliwiają stworzenie własnego tworu zbliżonego do list czy słownika. Przeanalizujemy poniższy wstępny przykład:

```
class MyDictionary:
    dict=dict()
    def add_element(self,key,value):
        self.dict[key]=value
    def get_element(self,key):
        return self.dict[key]
```

```
md = MyDictionary()
md.add_element('A',1000)
print(md.get_element('A'))
```

Stworzyłem w zasadzie klasę opakowującą słownik z możliwością uzupełniania i pobierania wartości z tego słownika. Chciałbym jednak móc stosować zapis taki jak przy słownikach i móc przypisywać wartości i odczytywać je tak jakby to był słownik tj. np:

```
ml['A']='Wartosc'
print(ml['A'])
```

Do tego celu przesłonimy funkcje `__getitem__` i `__setitem__` których zadanie będzie takie jak wcześniejszych metod `add_element` i `get_element`. Wykorzystanie metod "magicznych" pozwala nam jednak na przypisywanie i odczytywanie wartości w sposób zaprezentowany powyżej:

```
class MyDictionary:
    dict=dict()
    def __setitem__(self, key, value):
        self.dict[key]=value
    def __getitem__(self, key):
        return self.dict[key]
```

```
md = MyDictionary()
md['klucz']='wartość'
print(md['klucz'])
```

Teraz możemy naszą klasę wzbogacić o elementy których w tradycyjnych słownikach nie ma

Metody statyczne

W większości obiektowych języków programowania występuje pojęcie metod statycznych. Są to takie metody które możemy wywołać nie tworząc obiektu klasy. W poniższym przykładzie zadeklarowałem dwie metody - jedną zwykłą i jedną statyczną, nazwy chyba całkiem trafne:

```
class Ms:
    def zwykla(self):
        print('hej, jestem zwykłą metodą')
        pass
    @staticmethod
    def statyczna():
        print('hej, jestem metodą statyczną')
        pass

ms=Ms()
ms.zwykla()
Ms.statyczna()
```

Jeszcze tylko skutek działania powyższego kodu i przechodzimy do tłumaczenia:

hej, jestem zwykłą metodą

hej, jestem metodą statyczną

Aby wywołać metodę "zwykla", musimy powołać do życia obiekt klasy Ms i z niego dopiero wywołać tę metodę. Klasa Ms może jednak mieć bardzo wiele pól, lub pól które są od razu inicjalizowane dużą ilością danych, a to nie koniecznie musi być pożądanym przez nas skutkiem. My chcemy tylko wywołać metodę która akurat znajduje się w tej klasie. W takiej sytuacji możemy wykorzystać właśnie metodę statyczną. Przyjrzyj się wywołaniu "Ms.statyczna()" - wywołuję tę metodę z klasy a nie z obiektu (!!)

"ms", który odróżnić można po wielkości liter. Zwykłej metody w ten sam sposób wywołać nie mogę. Metoda statyczna staje się taką za sprawą 2 rzeczy - adnotacji "staticmethod", oraz faktu nie podania parametru "self".

W zasadzie to jest jedna sztuczka na wywołanie metody niestatycznej tak jak statycznej, może nawet warto ją znać, ale raczej nie będziesz jej często stosować. By tak się dało zrobić, przy wywołaniu metody zwykłej z klasy (a nie z obiektu!) jako jej parametr podajemy "None":

```
class Ms:
    def zwykla(self):
        print('hej, jestem zwykłą metodą')
        pass
    @staticmethod
    def statyczna():
        print('hej, jestem metodą statyczną')
        pass

Ms.zwykla(None)
```

Na konsoli pojawia się:

hej, jestem zwykłą metodą

i wszystko działa poprawnie, ale tylko dlatego że w metodzie "zwykla" nie odwołuję się do żadnych atrybutów ani metod obiektu.

Dziedziczenie

Dziedziczenie występuje w Pythonie jak chyba w każdym języku obiektowym, choć i tu nie obędzie się bez niespodzianek. Dziedziczenie to rozszerzanie klas, dla przykładu możemy przyjąć że mamy klasę "Samochod" posiadającą funkcje "jedz", "skrec", "hamuj". Następnie tworzymy klasę "SuperSamochod". Chcemy by klasa "SuperSamochod" zawsze posiadała to co klasa "Samochod", plus jakieś dodatkowe funkcje np "turbo". To jest właśnie miejsce na zastosowanie dziedziczenia. Przeanalizujmy taki właśnie przykład:

```
class Samochod:
    def jedz(self):
        print("no to jedziemy...")
    def hamuj(self):
        print("khhhhh.....")
    def skrec(self):
        print("na ręcznym!")

class SuperSamochod(Samochod):
    def turbo(self):
        print("włączamy podtlenek gazotu w 1.6 i mamy turbo na miarę  
'Szerszenia'")

s=Samochod()
s.jedz()
s.hamuj()
s.skrec()
ss=SuperSamochod()
ss.jedz()
ss.hamuj()
ss.skrec()
ss.turbo()
```

Zbieżność osób i nazwisk, a także odniesienia do "Szerszenia" z "Blok Ekipy" zupełnie przypadkowe ;) A więc... nie zaczyna się zdania od "a więc", ale ja mogę bo to moja książka ;). A więc... Mam klasę "Samochod", oraz klasę "SuperSamochod" dziedziczącą po klasie Samochod. Skąd wiemy że klasa "SuperSamochod" dziedziczy po klasie "Samochod" - z zapisu:

```
class SuperSamochod(Samochod):
```

Taki zapis oznacza że klasa deklarowana dziedziczy po klasie której nazwę obejmujemy w nawiasy. Co tu się dzieje dalej? Stworzyłem obiekt klasy "Samochod" i wywołałem na nim trzy obecne metody. Następnie stworzyłem obiekt klasy "SuperSamochod". Z niego też wywołałem te same metody, oraz dodatkową metodę "turbo". Posiadam w tej klasie również funkcje z klasy "Samochód" co wprost wynika z dziedziczenia właśnie.

Dziedziczenie po wielu klasach

Programiści Javy mogliby odnieść mylne wrażenie że to jakaś pomyłka. Otóż nie jest to pomyłka, taka konstrukcja jest możliwa w Pythonie. Klasa dziedzicząca będzie posiadać pola i metody wszystkich klas po których dziedziczy. Są tu oczywiście pewne ograniczenia, ale o tym za chwilę. Przyjrzyjmy się poniższym klasom:

```
class Pojazd:
    def jedz(self):
        print("wroom!")

class Armata:
    def strzelaj(self):
        print("jeb, jeb, jeb!")

class Czolg(Pojazd, Armata):
    pass

c=Czolg()
c.jedz()
c.strzelaj()
```

Mamy klasę "Pojazd" która posiada funkcję "jedz" i klasę "Armata" która posiada funkcję "strzelaj". Na końcu stworzyłem klasę Czolg dziedziczącą po obu wymienionych. Obiekty klasy "Czolg" będą więc posiadały obie metody. Uruchomienie spowoduje wyświetlenie obu komunikatów z obu funkcji. Co jednak gdyby okazało się że obie klasy posiadają funkcję o takiej samej nazwie? Którą z nich będziemy w takiej sytuacji wywoływać? Czy to w ogóle możliwe? Tak, jest to możliwe a wszystko sprowadza się do kolejności wymieniania klas po których dziedziczymy. Poniżej deklaracja trzech klas. Klasa C dziedziczy zarówno po klasie B jak i A. Klasy B i A posiadają taką samą funkcję "hello".

```
class A:
    def hello(self):
        print('siema, tu A')

class B:
    def hello(self):
        print('siema, tu B')

class C(B,A):
    pass
```

Po stworzeniu obiektu klasy C i jej wywołaniu :

```
c=C()  
c.hello()
```

otrzymujemy na konsoli komunikat:

siema, tu B

świadczący o tym że wywołana została metoda z klasy B. Teraz odwróćmy kolejność klas w dziedziczeniu:

```
class C(A,B):  
    pass  
  
c=C()  
c.hello()
```

Tym razem na konsoli pojawia się:

siema, tu A

co prowadzi nas do wniosku, że w takiej sytuacji obowiązują metody z klasy wymienionej jako pierwszej w dziedziczeniu.

Polimorfizm

Polimorfizm to po prostu wielopostaciowość. Bazując na naszym przykładzie z klasami "Samochod" i "SuperSamochod" - Każdy super samochód jest też samochodem i może być traktowany. Tu pojawia się jednak kilka zagadnień które mogą nieco skomplikować sprawę. Przyjmijmy że klasa "SuperSamochod" dziedzicząca po klasie "Samochod" posiada również funkcję "jedz" obecną w klasie po której dziedziczy. Co w takim przypadku? Co jeśli ta metoda jedz w klasie "SuperSamochod" będzie chciała wywołać tę metodę "jedz" z klasy dziedziczonej? Sprawdźmy to!

```
class Samochod:
    def jedz(self):
        print("pyr pyr")

class SuperSamochod(Samochod):
    def jedz(self):
        print("WROOOOOOOOM!!!!")

s=Samochod()
ss=SuperSamochod()
ss.jedz()
```

Obie klasy posiadają metodę "jedz", przy czym klasa "SuperSamochod" dziedziczy po klasie "Samochod". Jeśli wywołuję "ss.jedz()" to która z metod zostaje wywołana? Na konsoli pojawia się :

WROOOOOOOOM!!!!

Wywołana została metoda z klasy "SuperSamochod" - czego prawdę mówiąc można się było spodziewać. Dzieje się tak za sprawą przesłonięcia metody dziedziczonej przez metodę w klasie dziedziczącej. Gdybyś zechciał odwołać się do metody z klasy po której dziedziczysz, musisz posłużyć się takim zapisem:

```
class SuperSamochod(Samochod):
    def jedz(self):
        print("WROOOOOOOOM!!!!")
        super().jedz()
```

To "super()" odnosi się do klasy po której dziedziczymy. Zapis "super().jedz()" oznacza więc że wywołana zostaje metoda jedz() z klasy po której dziedziczymy.

Funkcje prywatne

Funkcje prywatne to takie funkcje, które można wywołać tylko z wnętrza klasy. Aby funkcja stała się prywatną należy przed jej nazwą dodać dwa znaki "_":

```
class FunkcjePrywatne:
    def funkcjaPubliczna(self):
        print("Cześć, jestem funkcją publiczną!")
    def __funkcjaPrywatna(self):
        print("Cześć, jestem funkcją PRYWATNĄ!")

fp = FunkcjePrywatne()
fp.funkcjaPubliczna()
fp.__funkcjaPrywatna()
```

Po uruchomieniu tego kodu na konsoli dostaniemy:

Cześć, jestem funkcją publiczną!

Traceback (most recent call last):

File "D:/data/workspace-python/nauka/materialy/02.py", line 9, in <module>

fp.__funkcjaPrywatna()

AttributeError: 'FunkcjePrywatne' object has no attribute '__funkcjaPrywatna'

Process finished with exit code 1

Dzieje się tak dlatego, że spoza obiektu nie możemy wywołać funkcji prywatnej. Teraz jednak zmienimy nieco kod:

```
class FunkcjePrywatne:
    def funkcjaPubliczna(self):
        print("Cześć, jestem funkcją publiczną!")
        self.__funkcjaPrywatna()
    def __funkcjaPrywatna(self):
        print("Cześć, jestem funkcją PRYWATNĄ!")

fp = FunkcjePrywatne()
fp.funkcjaPubliczna()
```

Tym razem na konsoli pojawia się:

Cześć, jestem funkcją publiczną!

Cześć, jestem funkcją PRYWATNĄ!

Tym razem udało się wywołać funkcję prywatną, ale tylko dlatego że wywoływaliśmy ją z wnętrza obiektu. Przy okazji chciałem zwrócić uwagę na pewną rzecz która mogła Ci umknąć - odwołałem się

z jednej funkcji do drugiej znajdującej się NAD funkcją wywołującą! W klasach to działa, w przeciwieństwie to funkcji poza klasami.

Prywatne mogą być również pola, sprawiamy że stają się prywatne dokładnie tak samo jak w przypadku funkcji - poprzedzając ich nazwę "__":

```
class PracownikJanuszexu:
    imie='Wania'
    nazwisko='Typowy'
    __wyplata='Czy pindzisiont za godzine'
    def info(self):
        print(self.imie,self.nazwisko,self.__wyplata)

pj = PracownikJanuszexu()
pj.info()
print(pj.nazwisko,pj.imie,pj.__wyplata)
```

Zaskoczenia pewnie nie będzie:

Traceback (most recent call last):

Wania Typowy Czy pindzisiont za godzine

File "D:/data/workspace-python/nauka/materialy/02.py", line 10, in <module>

print(pj.nazwisko,pj.imie,pj.__wyplata)

AttributeError: 'PracownikJanuszexu' object has no attribute '__wyplata'

Abstrakcja

Załóżmy że chcemy stworzyć klasę bazową reprezentującą bliżej nieokreślona figurę i kilka klas dziedziczących po klasie bazowej, takich jak "Kwadrat" i "Koło". Wymagamy by wszystkie klasy dziedziczące posiadały zdefiniowaną i dziedziczoną metodę "pokaz_nazwe" oraz pole "nazwa".

Dla każdej figury pole obliczać będziemy w inny sposób, dlatego też każda klasa dziedzicząca powinna posiadać swoją adaptację metody "oblicz_pole". Nie możemy zdefiniować sposobu działania tej metody w klasie "Figura" ponieważ pole każdej z figur będzie obliczane inaczej.

W klasie "Figura" określimy tylko rzeczy wspólne dla wszystkich figur. Aby osiągnąć opisany efekt będziemy musieli stworzyć metodę abstrakcyjną w klasie "Figura" i zaimplementować ją w klasach dziedziczących. Będzie to oznaczało że wszystkie klasy dziedziczące po klasie "Figura" będą musiały zaimplementować metodę obliczającą pole. W związku z tym że klasa "Figura" będzie posiadała metodę abstrakcyjną sama będzie klasą abstrakcyjną i nie będzie można stworzyć instancji tej klasy. Przyjrzyjmy się więc klasie "Figura":

```
from abc import ABC, abstractmethod
```

```
class Figura(ABC):
```

```
    nazwa=None
```

```
    def pokaz_nazwe(self):  
        print(self.nazwa)
```

```
    @abstractmethod
```

```
    def oblicz_pole(self):  
        pass
```

Zwróć uwagę że nasza klasa Figura dziedziczy po klasie ABC - jest to niezbędne by tworzyć metody abstrakcyjne, a co za tym idzie klasy abstrakcyjne.

Klasa ta definiuje pole "nazwa" które będzie posiadała każda klasa dziedzicząca po klasie "Figura".

Podobnie sprawy się mają z metodą "pokaz_nazwe". Wszystkie klasy dziedziczące klasę "Figura" będą już posiadały tą metodę odziedziczoną i zaimplementowaną. Nowością tutaj jest metoda "oblicz_pole" oznaczona adnotacją "@abstractmethod". Jak widzisz nie posiada ona implementacji. Każda klasa dziedzicząca po klasie "Figura" będzie więc musiała mieć własną implementację tej metody. Próba stworzenia obiektu tej klasy:

```
f=Figura()
```

spotka się z błędem:

TypeError: Can't instantiate abstract class Figura with abstract methods oblicz_pole

Dodajemy klasę "Kwadrat" dziedziczącą po klasie "Figura". Tworzymy konstruktor który umożliwia nam wstrzyknięcie do obiektu długość boku i dodatkowo uzupełnia pole "nazwa":

```
class Kwadrat(Figura):  
    def __init__(self, dlugosc_boku):  
        self.nazwa = 'Kwadrat'  
        self.dlugosc_boku = dlugosc_boku  
  
    def oblicz_pole(self):  
        return pow(self.dlugosc_boku, 2)
```

Tym razem możemy już stworzyć obiekt klasy, ponieważ mamy zaimplementowaną dziedziczoną metodę abstrakcyjną "oblicz_pole". Wywołajmy zatem:

```
kw = Kwadrat(5)  
print(kw.nazwa)  
print(kw.oblicz_pole())
```

Wynik na konsoli:

```
Kwadrat  
25
```

Dodajmy kolejną klasę, stwórzmy jej obiekt i wywołajmy wyświetlenie pola nazwa i wynik działania metody "oblicz_pole" która tym razem zaimplementowana jest inaczej:

```
class Prostokat(Figura):  
    def __init__(self, bok_a, bok_b):  
        self.nazwa = 'Prostokąt'  
        self.bok_a = bok_a  
        self.bok_b = bok_b  
  
    def oblicz_pole(self):  
        return self.bok_a * self.bok_b
```

```
pr = Prostokat(4, 5)  
print(pr.nazwa)  
print(pr.oblicz_pole())
```

Wynik na konsoli:

Prostokąt
20

Tworzenie własnych wyjątków

Przyjmijmy że mamy do zrealizowania program który będzie obliczał bmi na podstawie podanego wzrostu i masy:

```
def bmi(m,w):  
    return round(m/pow(w,2),2)  
  
print( bmi(80,1.76) )
```

Wymagane jest by wzrost został podany w metrach a masa w kilogramach. Gdyby ktoś podał wzrost w centymetrach, wynik działania tej funkcji będzie bzdurą choć wartość liczbową zostanie zwrócona. Chcielibyśmy w jakiś sposób reagować na taką sytuację. Możemy rzucić ogólnym wyjątkiem w ten sposób:

```
def bmi(m,w):  
    if w<1 or w>2.5:  
        raise Exception  
    return round(m/pow(w,2),2)  
  
print( bmi(80,176) )
```

Taki wyjątek niewiele powie użytkownikowi naszej funkcji:

Traceback (most recent call last):

File "D:/data/workspace-python/python_sredniozaawansowany/main.py", line 6, in <module>

print(bmi(80,176))

File "D:/data/workspace-python/python_sredniozaawansowany/main.py", line 3, in bmi

raise Exception

Exception

Wywołując wyjątek ogólny możemy przez argument konstruktora podać treść wyjątku:

```
def bmi(m,w):
    if w<1 or w>2.5:
        raise Exception('Wzrost poza zakresem')
    return round(m/pow(w,2),2)

print( bmi(80,176) )
```

Wynik działania:

Traceback (most recent call last):

```
File "D:/data/workspace-python/python_sredniozaawansowany/main.py", line 6, in <module>
    print( bmi(80,176) )
File "D:/data/workspace-python/python_sredniozaawansowany/main.py", line 3, in bmi
    raise Exception('Wzrost poza zakresem')
```

Exception: Wzrost poza zakresem

Co jednak jeśli zechcielibyśmy stworzyć własną klasę wyjątku która byłaby reużywalna? Musimy stworzyć własną klasę dziedziczącą po klasie Exception by następnie wywołać tak stworzony wyjątek:

```
class HeightOutOfRangeException(Exception):
    def __init__(self):
        super().__init__('Wzrost poza zakresem')

def bmi(m,w):
    if w<1 or w>2.5:
        raise HeightOutOfRangeException
    return round(m/pow(w,2),2)

print( bmi(80,176) )
```

W `__init__` przekazujemy argument dla funkcji `__init__` klasy po której dziedziczy - tj. `Exception`.
Wynik działania:

Traceback (most recent call last):

```
File "D:/data/workspace-python/python_sredniozaawansowany/main.py", line 10, in <module>
    print( bmi(80,176) )
File "D:/data/workspace-python/python_sredniozaawansowany/main.py", line 7, in bmi
    raise HeightOutOfRangeException
__main__.HeightOutOfRangeException: Wzrost poza zakresem.
```

Możemy również do wyjątku przekazać jakąś informację. W tym przypadku będzie to wzrost:

```
class HeightOutOfRangeException(Exception):
    def __init__(self,w):
        super().__init__(f'Wzrost poza zakresem. Podany wzrost: {w}')

def bmi(m,w):
    if w<1 or w>2.5:
        raise HeightOutOfRangeException(w)
    return round(m/pow(w,2),2)

print( bmi(80,176) )
```

Wynik działania:

Traceback (most recent call last):

```
File "D:/data/workspace-python/python_sredniozaawansowany/main.py", line 10, in <module>
    print( bmi(80,176) )
File "D:/data/workspace-python/python_sredniozaawansowany/main.py", line 7, in bmi
    raise HeightOutOfRangeException(w)
__main__.HeightOutOfRangeException: Wzrost poza zakresem. Podany wzrost: 176
```


Iteratory

Iterator to obiekt pozwalający na sekwencyjny dostęp do kolejnych elementów. Aby stworzyć iterator musimy w klasie zaimplementować dwie funkcje - "__iter__()" i "__next__()". Funkcja "__iter__()" zwraca obiekt iteratora, a "__next__()" zwraca kolejne elementy. "__next__()" wywołuje też wyjątek "StopIteration" kiedy nie ma już więcej elementów do oddania. Przeanalizujemy poniższy przykład:

```
class IncrementIterator:
```

```
    def __init__(self, n):
        self.n = n
        self.i = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.n == self.i:
            raise StopIteration
        self.i += 1
        return self.i
```

```
for e in IncrementIterator(10):
    print(e)
```

Przesłaniam metodę "__init__" w celu przekazania do obiektu maksymalnej wartości do której chcemy iterować. Metoda "__iter__" musi zwracać obiekt iterowalny, tak więc w tym przypadku zwracam "siebie" czyli obiekt w którym się znajduję. Metoda "__next__" powiększa o 1 i zwraca wewnątrzobiekтовую zmienną. Zwróć uwagę że w związku z taką konstrukcją obiekt "pamięta" jaką wartość oddał ostatnio. Przy każdym kolejnym wywołaniu "__next__" jest też sprawdzane czy nie dotarliśmy do maksimum do którego mieliśmy iterować, a jeśli tak to wywołujemy wyjątek "StopIteration". Pod deklaracją klasy iterujemy po naszym iteratorze. W wyniku działania dostajemy na konsoli kolejne liczby od 1 do 10. Jeśli raz przeiterujemy po naszym iteratorze, to nie ma możliwości jej "zresetowania". Aby zacząć liczyć od nowa, musimy stworzyć nowy obiekt.

Poprzedni przykład zakładał limit ilości zwracanych elementów. W Pythonie możemy tworzyć również nieskończone iteratory. Poniżej przykład:

```
class NieskonczonyIterator:
```

```
    x=0
```

```
    def __iter__(self):
```

```
        return self
```

```
    def __next__(self):
```

```
        self.x+=1
```

```
        return self.x
```

```
ni=NieskonczonyIterator()
```

```
for i in range(10):
```

```
    print( next(ni) )
```

Tym razem nie mamy metody "__init__" któraby ustawiała jakiś maksymalny próg zwracanych wartości. W metodzie "__next__" nie rzucam w związku z tym wyjątkiem "StopIteration". Mój iterator będzie więc podawał kolejne liczby bez ograniczeń. Pod przykładem iteratora prezentuję przykład pobierania wartości z takiego iteratora. Tym razem muszę posłużyć się funkcją "next" od obiektu iteratora by pobierać kolejne wartości.

Wątki

Wątki umożliwiają wykonywanie kilku czynności równolegle. Są to części jednego programu (procesu) które mogą działać jednocześnie i współdzielić zasoby.

Zanim zaczniemy omawiać implementację wielowątkowości w Pythonie, musimy nieco wyjaśnić kilka ważnych spraw.

Python jest językiem programowania ale posiada kilka implementacji. Są to m.in:

- CPython - implementacja w C
- PyPy - implementacja w Pythonie
- Jython - implementacja w Javie
- IronPython - implementacja w .NET

CPython i PyPy nie są przystosowane do wykonywania wielu czynności jednocześnie, dlatego też posiadają GIL (Global Interpreter Lock) . GIL powoduje że tylko jeden wątek ma dostęp do interpretera w danej chwili. Jest to rozwiązanie problemu z jednoczesnym dostępem do tych samych zasobów w tym samym czasie. Wszystkie wątki w związku z działaniem GILa będą szeregowane, czyli w rzeczywistości nie będą wykonywane równocześnie, a naprzemiennie. Dziać się to będzie jednak na tyle szybko, że będzie to wyglądało jakby były wykonywane równolegle.

Pakiet threading

Przyjrzyjmy się poniższemu przykładowi:

```
import time
import threading
def zamul(nazwa):
    print(f'start wątku {nazwa}\n')
    time.sleep(3)
    print(f'koniec wątku {nazwa}\n')

x=threading.Thread(target=zamul,args=('boczny',))
x.start()
zamul('główny')
```

Funkcja "zamul" wypisuje informacje o rozpoczęciu wątku, wstrzymuje działanie na 3 sekundy i wypisuje informację o zakończeniu wątku. Korzystając z klasy Thread znajdującej się w pakiecie threading tworzę obiekt wątku. Tworząc go, jako argumenty przekazuję funkcję do wykonania oraz argumenty które zostaną przekazane do tej funkcji. Za pomocą "x.start()" rozpoczynam ten wątek, a chwilę później uruchamiam funkcję "zamul" bezpośrednio w głównym wątku programu. Wynik działania na konsoli:

start wątku boczny

start wątku główny

koniec wątku główny

koniec wątku boczny

Jeśli uruchomisz ten kod u siebie kolejność wyświetlonych komunikatów na konsoli może być różna przy każdym uruchomieniu. Wynika to z tak zwanego zjawiska wyścigów, które są bezpośrednim skutkiem niedeterministyczności chwili wywołania (przerywania i przełączania) wątków.

Wątek jako demon

Zauważ że dotychczas uruchamiane wątki wykonywały się do końca, wątek główny czekał na zakończenie wszystkich wątków. Możesz też uruchomić wątek w trybie demona. Spowodujesz w ten sposób że wszystkie poboczne wątki będą kończyły pracę z zakończeniem głównego wątku programu, a główny wątek programu nie będzie czekał na zakończenie wątków bocznych. Przetestuj poniższy kod:

```
import time
import threading
def odliczaj(nazwa_watku, ile):
    for x in range(ile):
        print(x)
        time.sleep(1)

f=threading.Thread(target=odliczaj,args=('pierwszy',10),daemon=True)
f.start()
```

Program wypisuje na konsoli tylko 0, po czym kończy działanie. Gdybyś zechciał by główny wątek poczekał na zakończenie wątku demona, trzeba wywołać funkcję "join()" na obiekcie wątku:

```
import time
import threading
def odliczaj(nazwa_watku, ile):
    for x in range(ile):
        print(x)
        time.sleep(1)

f=threading.Thread(target=odliczaj,args=('pierwszy',10),daemon=True)
f.start()
f.join()
```

Wywołanie join spowoduje że główny wątek zatrzyma się na tej linii i będzie czekał z dalszym wykonaniem programu do momentu zakończenia działania wątku dziecka.

Odbieranie wartości z wątku

Aby odebrać wartości zwracaną z funkcji uruchomionej w ramach "bocznego" wątku musimy posłużyć się inną wysokopoziomową biblioteką Pythona:

```
import time
import concurrent.futures
```

```
def oddaj():
    time.sleep(3)
    return 'koza'
```

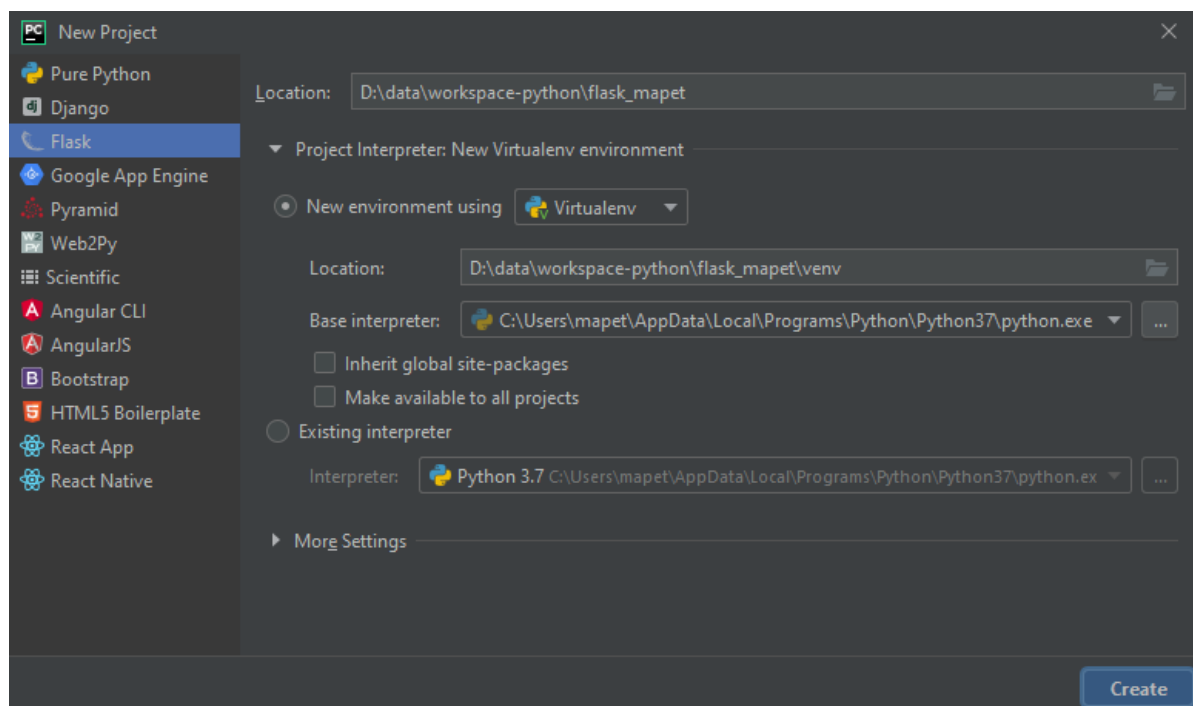
```
wykonawca=concurrent.futures.ThreadPoolExecutor()
zadanie = wykonawca.submit(oddaj)
zwrot = zadanie.result()
print(zwrot)
```

Flask – aplikacje internetowe

W tym rozdziale, podobnie jak i innych rozdziałach dotyczących aplikacji webowych w Pythonie, będę używał Pycharm w wersji Professional ze względu na wsparcie dla Flaska i Django. Można pobrać wersję trial na potrzeby nauki, ale do profesjonalnej pracy proponuję faktycznie wykupić subskrypcję. Wprawdzie kosztuje ona niecałe 200zł miesięcznie, ale przyspiesza pracę tak bardzo, że moim zdaniem warto. Inna sprawa – profesjonalny rysownik też nie używa produkcyjnie darmowych ołówków z Ikea.

Tworzenie projektu i mapowanie pierwszego adresu

Jeśli wybrałeś proponowane przeze mnie oprogramowanie, to tworząc nowy projekt powinieneś mieć taki mniej więcej widok:



W zasadzie jedyne co trzeba tu zrobić to nadać nazwę projektowi – ja nadałem nazwę „flask_mapet”. W projekcie powinien zostać stworzony plik „app.py” który będzie naszym głównym plikiem uruchomieniowym.

Jego zawartość poniżej:

```
app.py x
1  from flask import Flask
2
3  app = Flask(__name__)
4
5
6  @app.route('/')
7  def hello_world():
8      return 'Hello World!'
9
10
11  if __name__ == '__main__':
12      app.run()
13
```

Co oznaczają poszczególne elementy? Linia 1 – importujemy klasę „Flask” z modułu „flask” którą będziemy wykorzystywać za chwilę. Linia 3 – konieczne jest stworzenie instancji obiektu modułu Flask, ponieważ później się do niego odnosimy na przykład uruchamiając apkę (w linii 12) czy w @app.route (w linii 6). Konstruktor oczekuje podania nazwy aplikacji jako parametru.

Linie 6-8. W linii 6 używamy dekoratora pozwalającego nam zadeklarować mapowany przez funkcję url (czyli w reakcji na jaki adres http w ramach naszej aplikacji ma odpowiadać dana funkcja). Wartość argumentu: „/” oznacza że poniższa funkcja będzie mapować stronę główną. Funkcja nie przyjmuje żadnych argumentów.

Zwraca za to w linii 8 kod który ma zostać wyświetlony w reakcji na wejście na url mapowany przez funkcję.

Linie 11-12. To jest kod zapewniający że serwer jest uruchamiany kiedy odpalamy plik app.py będący głównym skryptem aplikacji. W zasadzie chodzi o wywołanie funkcji „run()” na obiekcie instancji obiektu modułu Flask.

Warunek służy temu, by nie następowały kolejne próby uruchamiania w sytuacji gdyby np. ten moduł był importowany. Jeśli to skrypt „app.py” będzie uruchamiany, to w „__name__” dostaniemy „__main__”.

Aplikację możemy uruchomić z konsoli wywołując skrypt jak każdy inny:

```
(venv) D:\data\workspace-python\flask_mapet>python app.py
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```


Flask ma wbudowany własny serwer http, będzie on domyslnie chodził na porcie 5000. Po wejściu przez przeglądarkę na ten port powinniśmy zobaczyć wynik działania funkcji „hello_world” która obsługuje główną stronę aplikacji:



Konfiguracja portu nasłuchu serwera i automatyczna implementacja zmian.

Odrobinę zmodyfikuję teraz skrypt „app.py”:

```
10
11 ► if __name__ == '__main__':
12     app.run(debug=True, port=80)
13
14
```

Dodałem argumenty do wywołania funkcji run. Parametr „port” pozwala ustawić na którym porcie ma nasłuchiwać serwer. Zmieniam na 80 by móc wywoływać aplikację po prostu „localhost” zamiast „localhost:5000”. Parametr „debug” powoduje, że zmiany na plikach z kodem będą od razu aktualizowane na serwerze, bez potrzeby restartu. Poniżej przykład takiego przeładowania. To jest zrzut logów. Dodałem do skryptu enter, zapisałem i aplikacja została przeładowana:

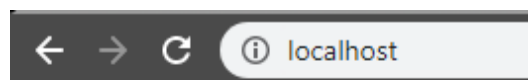
```
venv) D:\data\workspace-python\flask_mapet>python app.py
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Restarting with stat
* Debugger is active!
* Debugger PIN: 295-470-866
* Running on http://127.0.0.1:80/ (Press CTRL+C to quit)
* Detected change in 'D:\\data\\workspace-python\\flask_mapet\\app.py', reloading
* Restarting with stat
* Debugger is active!
* Debugger PIN: 295-470-866
* Running on http://127.0.0.1:80/ (Press CTRL+C to quit)
```

Kod i szablon kodu HTML

Funkcja obsługująca mapowanie adresu może zamiast zwykłego tekstu, zwracać również kod HTML:

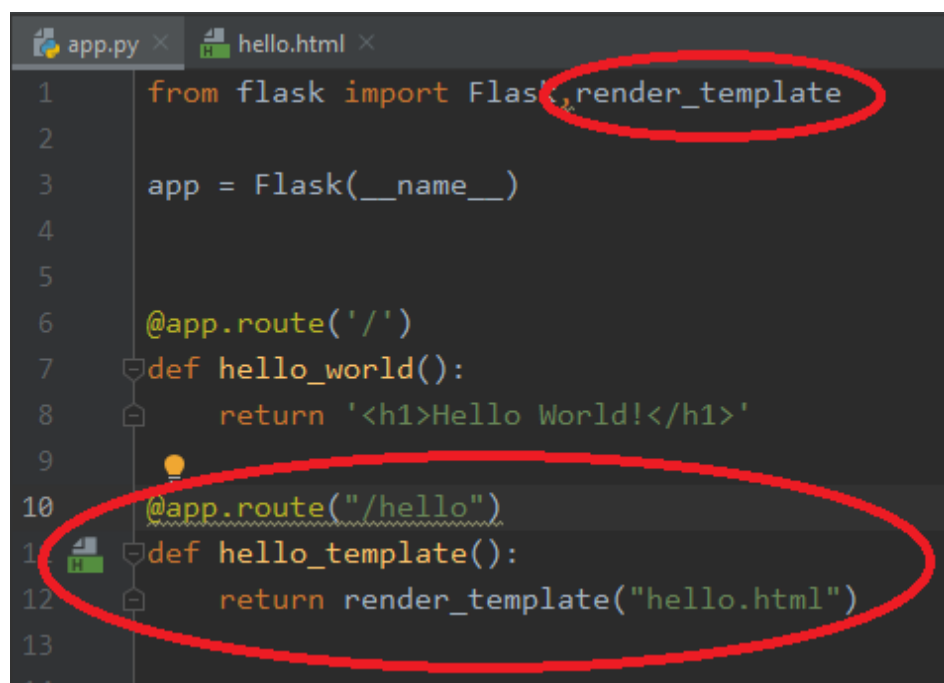
```
6     @app.route('/')
7     def hello_world():
8         return '<h1>Hello World!</h1>'
9
10
11  ►  if __name__ == '__main__':
12      app.run(debug=True, port=80)
13
```

Efekt działania:



Hello World!

Na dłuższą metę, lub przy większej ilości kodu takie działanie będzie po prostu bardzo niewygodne. Wyobraź sobie 1000 linikową stronę generowaną w ten sposób. Biorąc jeszcze pod uwagę że zwykle będą jakieś elementy wspólne na wielu podstronach, a podmieniana będzie tylko jakaś nieduża część widoku, to jest to całkowicie pozbawione sensu. Co więc możemy zrobić? Wykorzystać szablon HTML:

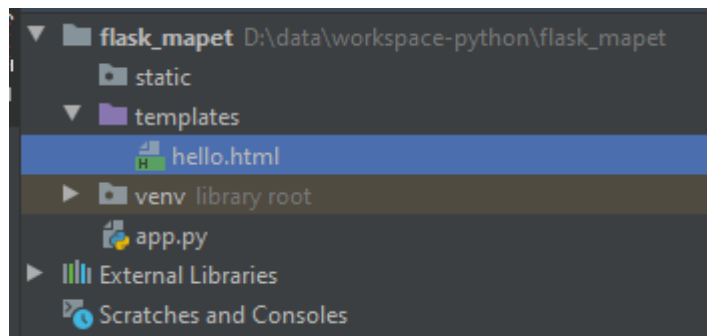


The image shows a code editor with two tabs: 'app.py' and 'hello.html'. The 'app.py' tab is active, displaying the following Python code:

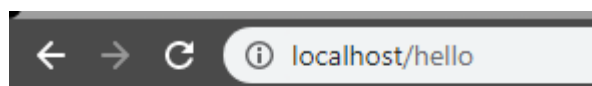
```
1 from flask import Flask, render_template
2
3 app = Flask(__name__)
4
5
6 @app.route('/')
7 def hello_world():
8     return '<h1>Hello World!</h1>'
9
10 @app.route("/hello")
11 def hello_template():
12     return render_template("hello.html")
13
14
```

Two red annotations are present: a red oval around the `render_template` import on line 1, and a larger red oval around the `@app.route("/hello")` decorator and the `hello_template` function definition on lines 10 and 11.

Wykorzystamy funkcję „render_template”(linia 12) którą trzeba uprzednio zaimportować (linia 1). Funkcja jako argument będzie przyjmowała nazwę pliku który ma zostać pokazany. Co ważne, plik ten musi znajdować się w podkatalogu „templates”, ponieważ Flask tam właśnie będzie go szukał.



Efekt działania:



To jest template....

Przekazywanie danych do widoku i jinja2

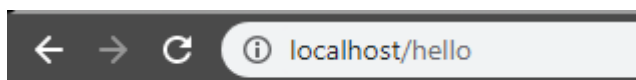
Wszystko pięknie, ale widok ten jest statyczny. Jak zatem przekazać dane do widoku? Kod z poprzedniego przykładu nieco przerobiłem. Do funkcji „render_template” poza samą nazwą widoku, przekazuję też dwie zmienne: imię i nazwisko. Pod takimi nazwami zostaną do widoku przekazane dane. Typ danych nie ma tu wielkiego znaczenia, jeśli okazałoby się że jest to kolekcja, różnica będzie tylko w sposobie wyświetlania.

```
9
10 @app.route("/hello")
11 def hello_template():
12     return render_template("hello.html", imie="Andrzej", nazwisko="Klusiewicz")
13
```

Mała przeróbka po stronie szablonu HTML:

```
app.py x  hello.html x
1 <html>
2 <body>
3 <h1>To jest template....</h1>
4 <h3>Imię: {{ imie }}</h3>
5 <h3>Nazwisko: {{ nazwisko }}</h3>
6 </body>
7 </html>
```

Pomiędzy podwójnymi klamrami umieszczamy nazwę pod jaką przekazaliśmy dane do widoku w argumentach funkcji „render_template”. Efekt działania:



To jest template....

Imię: Andrzej

Nazwisko: Klusiewicz

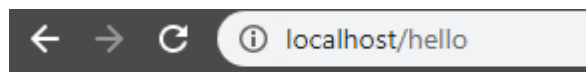
To są pojedyncze zmienne. Jak jednak przekazać listę, a następnie przeiterować po niej na poziomie widoku? W pierwszej kolejności modyfikuję funkcję by przekazać dane:

```
10 @app.route("/hello")
11 def hello_template():
12     kraje=["Polska","Niemcy","Rosja","Ukraina","Czechy"]
13     return render_template("hello.html", imie="Andrzej", nazwisko="Klusiewicz", kraje=kraje)
14
```

Utworzyłem sobie listę krajów którą przekazałem wraz z imieniem i nazwiskiem do widoku. Po stronie widoku korzystam ze składni jinja2:

```
6 <ul>
7     {% for k in kraje %}
8         <li>{{ k }}</li>
9     {% endfor %}
10 </ul>
```

Wynik działania:



To jest template....

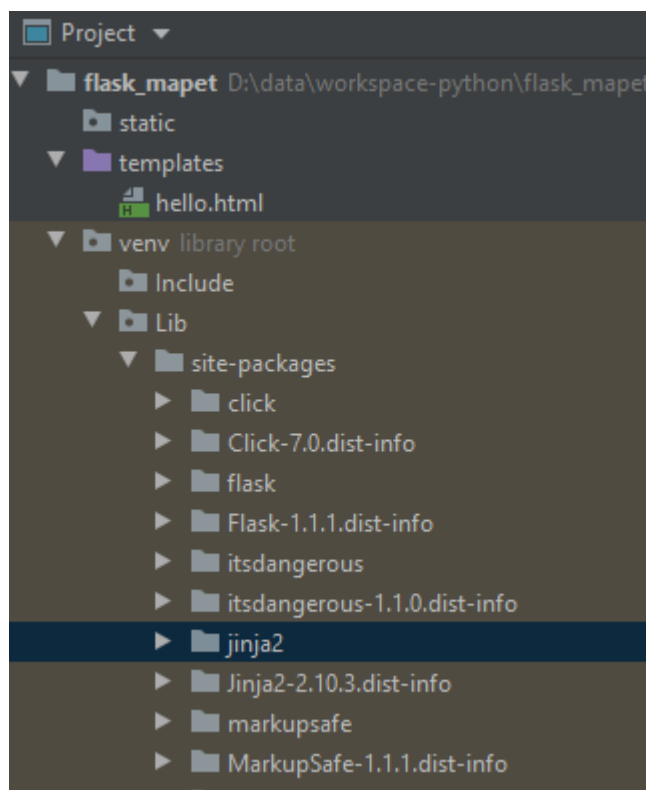
Imię: Andrzej

Nazwisko: Klusiewicz

- Polska
- Niemcy
- Rosja
- Ukraina
- Czechy

Zaraz, zaraz.... jinja? Jinja2 to silnik szablonów dla Pythona. Taki powiedzmy mikro język skryptowy który umożliwia nam zawieranie pewnej logiki biznesowej po stronie widoku. Daje nam możliwość wyświetlania danych przekazanych do `render_template`, iterowanie po listach na poziomie widoków, warunkowe wykonywanie czynności, a także kilka innych możliwości które omówimy nieco później.

Sama Jinja2 to osobna biblioteka którą musicie dodać do projektu wykonując „pip install jinja2”. Jeśli korzystacie z Pycharm Professional, nie musicie się o to marwić bo IDE zadbało już o to za Was. Gdyby Pycharm podkreślał Ci składnię jinja2, upewnij się że Jinja2 znajduje się w venv projektu:



Wracając jednak do składni:

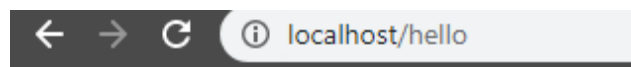
```
6 <ul>
7     {% for k in kraje %}
8         <li>{{ k }}</li>
9     {% endfor %}
10 </ul>
```

Iterujemy po liście o nazwie „kraje”, ponieważ pod taką nazwą została przekazana nasza lista.

```
10 @app.route("/hello")
11 def hello_template():
12     kraje=["Polska","Niemcy","Rosja","Ukraina","Czechy"]
13     return render_template("hello.html", imie="Andrzej", nazwisko="Klusiewicz", kraje=kraje)
14
```

Typo: In word 'Klusiewicz'

Odwołujemy się potem do tej listy w linii 7 naszego kodu HTML we fragmencie „for k in kraje”. Klamry i znaczniki „%” są elementem tej składni. Fragment „for k in” to deklaracja w jaki sposób będę się odnosił w pętli do elementów po których iteruję. Wewnątrz pętli posługuję się notacją taką jak przy wyświetleniu zwykłych zmiennych „{{ k }}”, z tą różnicą że odnoszę się do lokalnego „k”. Nie zapominamy też o „{% endfor %}” który zamyka pętlę. Efekt działania:



To jest template....

Imię: Andrzej

Nazwisko: Klusiewicz

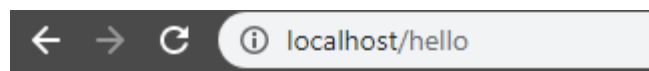
- Polska
- Niemcy
- Rosja
- Ukraina
- Czechy

Jinja2 umożliwia też warunkowe wykonanie. Poniżej przykład pewnego wariantu naszej aplikacji. Postanowiłem że na liście „Polska” ma być pogrubiona:

```
6  <ul>
7    {% for k in kraje %}
8      {% if k=="Polska" %}
9        <li><b>{{ k }}</b></li>
10     {% else %}
11     <li>{{ k }}</li>
12     {% endif %}
13   {% endfor %}
14 </ul>
```


We wnętrzu naszej pętli umieściłem jeszcze blok warunkowy. Jeśli przetwarzany element będzie równy ciągowi tekstowemu „Polska”, wyświetlany element będzie dodatkowo otoczony takami „” i „”.

Efekt działania:



To jest template....

Imię: Andrzej

Nazwisko: Klusiewicz

- **Polska**
- Niemcy
- Rosja
- Ukraina
- Czechy

Taka instrukcja warunkowa może przyjmować oczywiście więcej wariantów, podobnie jak w czysto pythonowym ifie. W poniższym przykładzie Rosja zostanie podkreślona:

```
6  <ul>
7    {% for k in kraje %}
8      {% if k=="Polska" %}
9        <li><b>{{ k }}</b></li>
10     {% elif k=="Rosja" %}
11       <li><u>{{ k }}</u></li>
12     {% else %}
13       <li>{{ k }}</li>
14     {% endif %}
15   {% endfor %}
16 </ul>
```

Efekt działania:

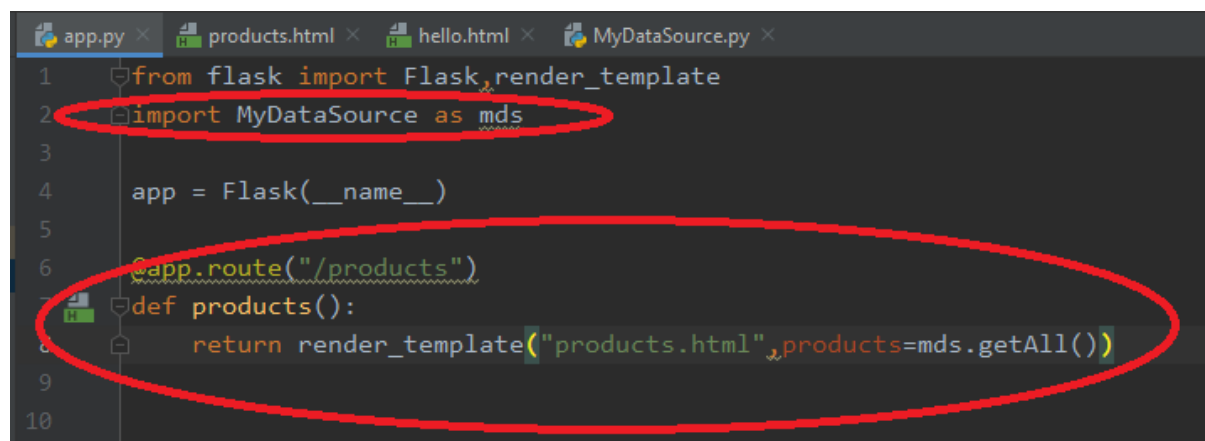
- **Polska**
- Niemcy
- Rosja
- Ukraina
- Czechy

Odczyt parametrów z paska

Możliwość odczytu parametrów z paska to funkcjonalność „pierwszej potrzeby”. Będzie wykorzystywana wszędzie tam gdzie zechcesz np. zrobić podgląd szczegółów jakiejś encji, edycję jakiegoś wpisu w bazie danych etc. We wszystkich tego rodzaju przypadkach trzeba będzie przekazać do widoku unikalny identyfikator podglądanego, kasowanego czy edytowanego obiektu. Na potrzeby przykładu dodałem do projektu nowy plik „MyDataSource”. Nie ma tu nic nadzwyczajnego, ani też nic nowego. Jest to dla nas typowo użytkowa klasa symulująca DAO. Mamy więc klasę „Product” której obiekty będą wyświetlać. Funkcja „getAll” zwraca listę takich obiektów, a „getOne” jeden taki obiekt:

```
1 class Product:
2     def __init__(self, id, name, description):
3         self.id=id
4         self.name=name
5         self.description=description
6
7     dane=[
8         Product(1, 'Bulbulator', 'Robi bul bul'),
9         Product(2, 'Wihajster', 'Nie wiadomo co robi'),
10        Product(3, 'Dzyndzel', 'Z przyczłapem')
11    ]
12
13
14    def getAll():
15        return dane
16
17    def getOne(id):
18        return [e for e in dane if int(e.id)==int(id)][0]
19
```

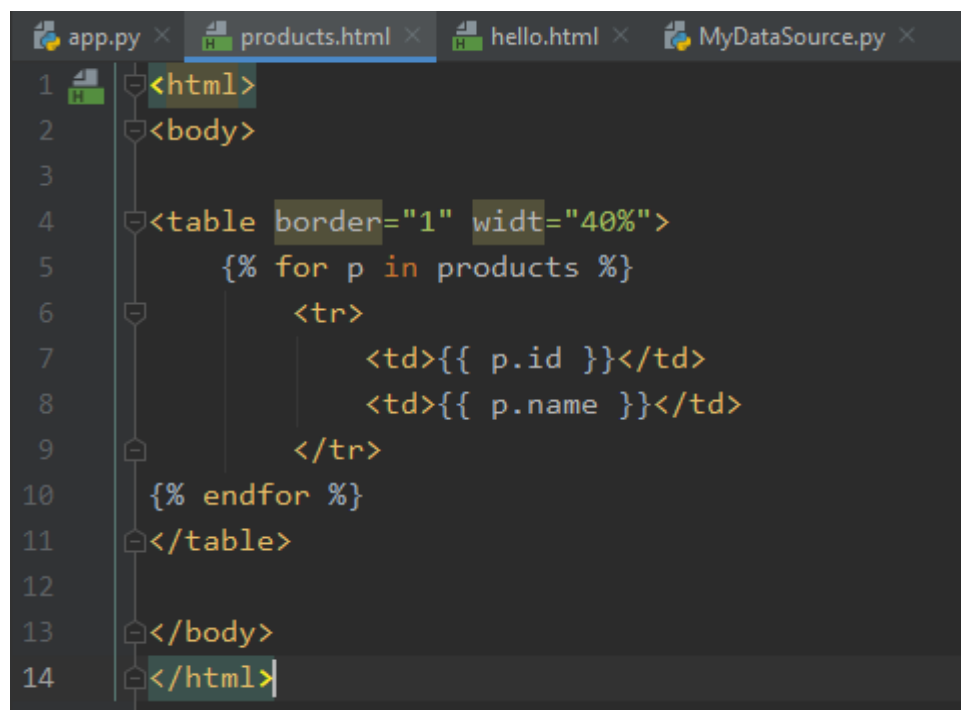
W dalszej kolejności dodaję do aplikacji dodatkowy ekran, będzie widoczny pod adresem „/products” (linia 6). W reakcji na wywołanie tego url wyświetlam plik „products.html” (z katalogu templates), oraz przekazuję pod nazwą „products” listę obiektów zwróconych przez funkcję „getAll”.



```
1 from flask import Flask, render_template
2 import MyDataSource as mds
3
4 app = Flask(__name__)
5
6 @app.route("/products")
7 def products():
8     return render_template("products.html", products=mds.getAll())
9
10
```

The screenshot shows a code editor with four tabs: app.py, products.html, hello.html, and MyDataSource.py. The app.py file contains the following code: Line 1: `from flask import Flask, render_template`; Line 2: `import MyDataSource as mds` (circled in red); Line 3: blank; Line 4: `app = Flask(__name__)`; Line 5: blank; Line 6: `@app.route("/products")` (circled in red); Line 7: `def products():` (circled in red); Line 8: `return render_template("products.html", products=mds.getAll())` (circled in red); Line 9: blank; Line 10: blank.

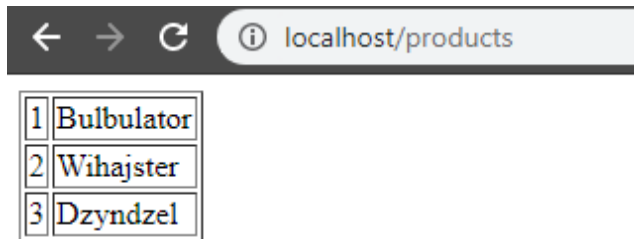
Po stronie widoku również na razie nie robię nic nadzwyczajnego. Iteruję po przekazanej liście by wyświetlić przekazane produkty:



```
1 <html>
2 <body>
3
4 <table border="1" width="40%">
5     {% for p in products %}
6         <tr>
7             <td>{{ p.id }}</td>
8             <td>{{ p.name }}</td>
9         </tr>
10    {% endfor %}
11 </table>
12
13 </body>
14 </html>
```

The screenshot shows a code editor with four tabs: app.py, products.html, hello.html, and MyDataSource.py. The products.html file contains the following HTML code: Line 1: `<html>`; Line 2: `<body>`; Line 3: blank; Line 4: `<table border="1" width="40%">`; Line 5: `{% for p in products %}`; Line 6: `<tr>`; Line 7: `<td>{{ p.id }}</td>`; Line 8: `<td>{{ p.name }}</td>`; Line 9: `</tr>`; Line 10: `{% endfor %}`; Line 11: `</table>`; Line 12: blank; Line 13: `</body>`; Line 14: `</html>`.

Wynik działania nie należy do szczególnie widowiskowych, ot tabelka z id i nazwą produktów:



1	Bulbulator
2	Wihajster
3	Dzyndzel

Dopiero teraz zaczyna się właściwa zabawa. Naturalnie nie wyświetlam opisów produktów, ponieważ te mogą być bardzo długie, zajmować sporą część ekranu, a przy przeglądaniu listy produktów tak szczegółowe informacje o nich nie są nam potrzebne. Dorobimy sobie zatem podstronę szczegółów produktu. Będzie nam potrzebna podstrona, której struktura adresu powinna wyglądać mniej więcej tak: „showProduct?id=2”. Wartość występująca po „id=” to id produktu którego szczegóły chcemy oglądać. Będziemy więc musieli dorobić linki typu „pokaż szczegóły”. W pierwszej kolejności dokonuję pewnej przeróbki w kodzie html (plik products.html):

```
4 <table border="1" width="40%">
5   {% for p in products %}
6     <tr>
7       <td>{{ p.id }}</td>
8       <td>{{ p.name }}</td>
9       <td>showProduct?id={{ p.id }}</td>
10    </tr>
11  {% endfor %}
12 </table>
```

Pojawiła nam się dodatkowa kolumna. Umyślnie zrobiłem tak by na razie był to po prostu tekst do wyświetlenia. Chciałem w ten sposób pokazać, że tagi jinja można również stosować w ten sposób, np. dynamicznie wstrzykiwać wartości do linków. Pod takie adresy będą przekierowywały linki gdy te napisy się nimi staną:

1	Bulbulator	showProduct?id=1
2	Wihajster	showProduct?id=2
3	Dzyndzel	showProduct?id=3

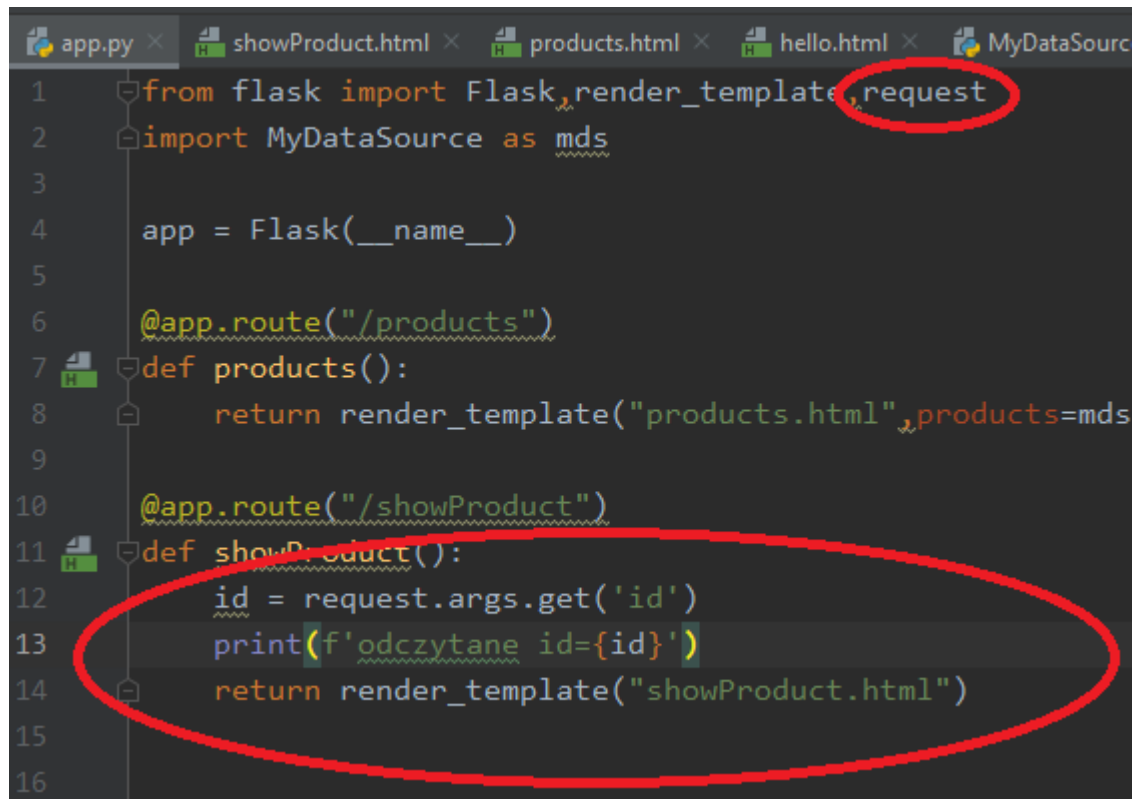
Jak widzimy, dzięki takiemu zabiegowi każdy produkt ma link różniący się wartością parametru ID. W kolejnym kroku dokonuję takiej przeróbki by wcześniej napisy, teraz stały się adresami pod które odsyłają linki:

```
4 <table border="1" width="40%">
5     {% for p in products %}
6         <tr>
7             <td>{{ p.id }}</td>
8             <td>{{ p.name }}</td>
9             <td><a href="showProduct?id={{ p.id }}">Pokaż</a></td>
10        </tr>
11    {% endfor %}
12</table>
```

Efekt po uruchomieniu:

1	Bulbulator	Pokaż
2	Wihajster	Pokaż
3	Dzyndzel	Pokaż

Pozostaje nam zadbać by po kliknięciu pojawiła się odpowiednia strona prezentująca szczegóły produktu. Dodaję więc kolejną metodę do obsługi nowego widoku:

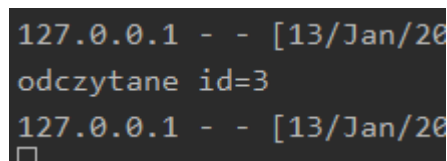


```
1 from flask import Flask, render_template, request
2 import MyDataSource as mds
3
4 app = Flask(__name__)
5
6 @app.route("/products")
7 def products():
8     return render_template("products.html", products=mds.products)
9
10 @app.route("/showProduct")
11 def showProduct():
12     id = request.args.get('id')
13     print(f'odczytane id={id}')
14     return render_template("showProduct.html")
15
16
```

Przykład jak dostać się do parametrów z paska pokazuję w linii 12. Odnosimy się do obiektu request który trzeba uprzednio zaimportować (linia 1). Można znaleźć w tym obiekcie całkiem sporo ciekawych rzeczy, takich jak aktualny URL, host użytkownika etc. Odsyłam do

<https://flask.palletsprojects.com/en/1.1.x/api/#flask.Request.args>.

W tym przypadku odczytane ID wyświetlam po prostu na konsoli i wyświetlam na razie pusty plik „showProducts.html”. Po uruchomieniu i kliknięciu na link dostajemy taki efekt:



```
127.0.0.1 - - [13/Jan/20
odczytane id=3
127.0.0.1 - - [13/Jan/20
□
```

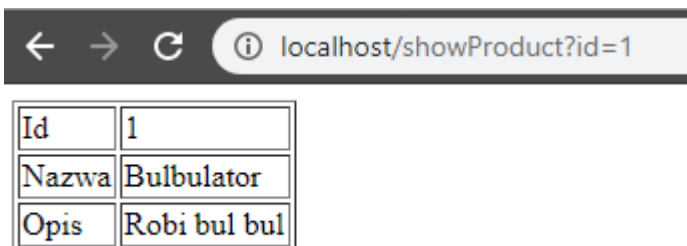
Pozostaje nam teraz odczytanie obiektu którego szczegóły chcemy przeglądać, przekazanie go do widoku i wyświetlenie. Korzystając z mojego pseudo DAO pobieram pojedynczy obiekt klasy Product (linia 13). Produkt ten przekazuję jak zawsze do widoku (linia 14):

```
10 @app.route("/showProduct")
11 def showProduct():
12     id = request.args.get('id')
13     p=mds.getOne(id)
14     return render_template("showProduct.html",product=p)
```

Po stronie widoku wszystko odbywa się jak dotychczas, z taką różnicą że zagłębiam się w pola obiektu (linie 4-6):

```
app.py x showProduct.html x products.html x hello.html x MyDataSource.py x
1 <html>
2 <body>
3 <table border="1">
4     <tr><td>Id</td><td>{{ product.id }}</td></tr>
5     <tr><td>Nazwa</td><td>{{ product.name }}</td></tr>
6     <tr><td>Opis</td><td>{{ product.description }}</td></tr>
7 </table>
8 </body>
9 </html>
```

Po kliknięciu w link przy produkcie „Bulbulator” dostajemy taki efekt:



The screenshot shows a web browser window with the address bar displaying `localhost/showProduct?id=1`. Below the address bar, there is a table with the following content:

Id	1
Nazwa	Bulbulator
Opis	Robi bul bul

Pobieranie i umieszczanie danych w sesji

Flask umożliwia pobieranie i umieszczanie różnych rzeczy w sesji. Mogą to być proste dane, ale też złożone obiekty. Poniżej przykład jednocześnie ustawiający i pobierający wartość z sesji:

```
1 from flask import Flask, render_template, request, session
2 import MyDataSource as mds
3
4 app = Flask(__name__)
5 app.config['SECRET_KEY'] = 'mojehaslo'
6
7 @app.route("/testSession")
8 def testSession():
9     print("umieszczanie użytkownika w sesji")
10    session['loggedUser'] = "ThePaniHalynaKsiegowa"
11    print("użytkownik zalogowany to " + session['loggedUser'])
12    return ""
13
14 @app.route("/products")
```

Należy pamiętać o dodaniu importu do „session” (linia 1), oraz umieszczeniu jakiejś wartości w konfiguracji „SECRET_KEY” (linia 5). Jest ona używana do zabezpieczenia po stronie klienta, może to być naprawdę dowolna wartość, byleby taka konfiguracja się pojawiła. Jeśli tego nie zrobisz, na ekranie zamiast spodziewanej treści pojawi się błąd o braku wartości dla klucza „SECRET_KEY”. „session” pozwalający pobierać i ustawiać wartości w sesji jest zwyczajnym słownikiem. Możesz więc pod dowolnym kluczem umieścić dowolną wartość. Ja pod kluczem „loggedUser” umieściłem ciąg tekstowy „ThePaniHalynaKsiegowa” (linia 10), by w kolejnej linii ją wyświetlić:

```
umieszczanie użytkownika w sesji
użytkownik zalogowany to ThePaniHalynaKsiegowa
```

Funkcja nic nie zwraca sensownego, do tego przykładu nie dorabiałem widoku.

Obsługa formularzy

Do obsługi formularza potrzebne będą dwa elementy. Jeden dbający o to co się ma stać gdy wejdziemy na formularz (GET), oraz co się ma stać gdy formularz zatwierdzimy (POST). Funkcja na potrzeby wyświetlenia formularza nie odbiega od wcześniej tworzonych i sprowadza się do wyświetlenia strony html:

```
7  @app.route("/addProduct")
8  def addProduct():
9      return render_template("addProduct.html")
```

Musimy zrobić również formularz w HTML:

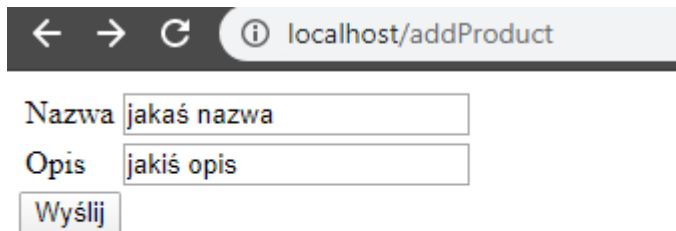
```
3  <form method="POST">
4      <table>
5          <tr>
6              <td>Nazwa</td>
7              <td><input type="text" name="name"/></td>
8          </tr>
9          <tr>
10             <td>Opis</td>
11             <td><input type="text" name="desc"/></td>
12         </tr>
13     </table>
14     <input type="submit" value="Wyślij"/>
15 </form>
```

Tutaj również nie ma nic nadzwyczajnego, ot zwykły formularz w HTML. Zwróć jednak uwagę na atrybut „name” obu inputów. Po tych właśnie nazwach będziemy odbierać dane na poziomie kontrolera.

Ciekawsze rzeczy znajdują się w obsłudze zatwierdzenia formularza:

```
11 @app.route("/addProduct", methods=['POST'])
12 def addProductPost():
13     nazwa=request.form['name']
14     opis=request.form['desc']
15     print(f"nazwa={nazwa}, opis={opis}")
16     return render_template("addProduct.html")
17
```

Pierwsza różnica ujawnia się w linii 11. Dochodzi nam dodatkowy argument „methods”. Dodajemy go by Flask wiedział kiedy ta funkcja ma zostać wywołana. Domyślną wartością jest „GET”. Dalej dzięki słownikowi „form” zawartym w obiekcie request (który trzeba zaimportować jak wcześniej) odczytuję na podstawie nazw inputów dane z formularza. Dalej (linie 15-16) wyświetlam na konsoli odczytane wartości i ponownie prezentuję formularz.



← → ↻ ⓘ localhost/addProduct

Nazwa

Opis

Konsola:

```
127.0.0.1 - - [13/Jan/2020 15:09:51] "GET /addProduct HTTP/1.1"
nazwa=jakaś nazwa, opis=jakiś opis
```

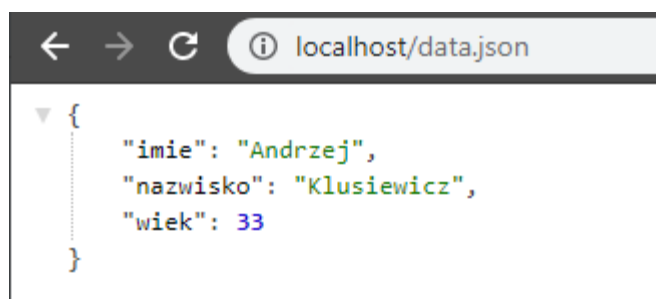
Usługi sieciowe we Flask

Usługi sieciowe zwracające dane

Usługi sieciowe są coraz częściej stosowane, zwłaszcza na fali popularności mikroserwisów. Flask idealnie się do tego nadaje, można szybko i prosto stworzyć lekką, a funkcjonalną aplikację. Można to samo osiągnąć z użyciem Django, ale próg wejścia jest nieco wyższy. Przykładowy kodzik:

```
1  from flask import Flask
2
3  app=Flask(__name__)
4
5  dane={
6      "imie": "Andrzej",
7      "nazwisko": "Klusiewicz",
8      "wiek": 33
9  }
10
11  @app.route("/data.json")
12  def getAllJson():
13      return dane
14
15  if __name__ == "__main__":
16      app.run(debug=True, port=80)
```

I efekt działania:

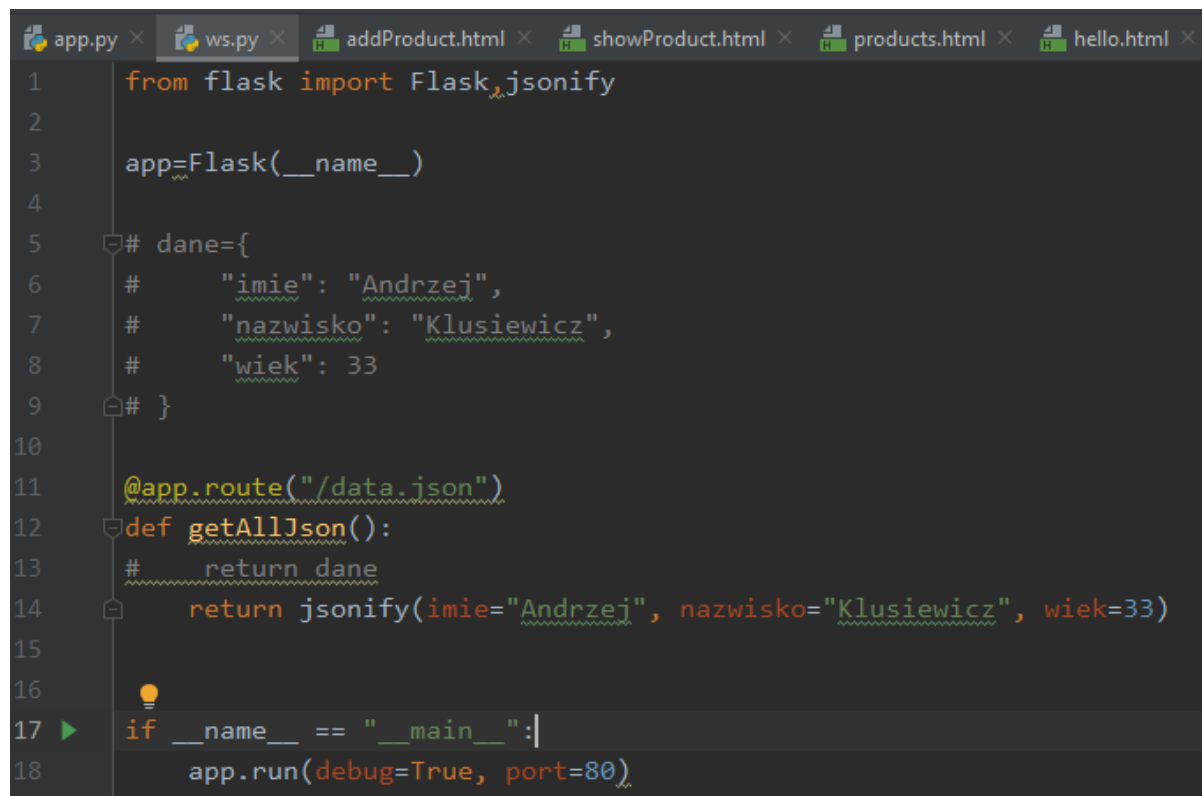


The screenshot shows a web browser window with the address bar displaying 'localhost/data.json'. The page content shows a JSON object with the following structure:

```
{
  "imie": "Andrzej",
  "nazwisko": "Klusiewicz",
  "wiek": 33
}
```

Skrypt pod usługę sieciową niewiele się różni od takiego pod zwykłą stronę. Z nowych rzeczy – między liniami 5-9 pojawił się słownik z danymi. W linii 13 zamiast nazwy widoku i użycia funkcji „render_template”, bądź kodu html, zwracam po prostu dane.

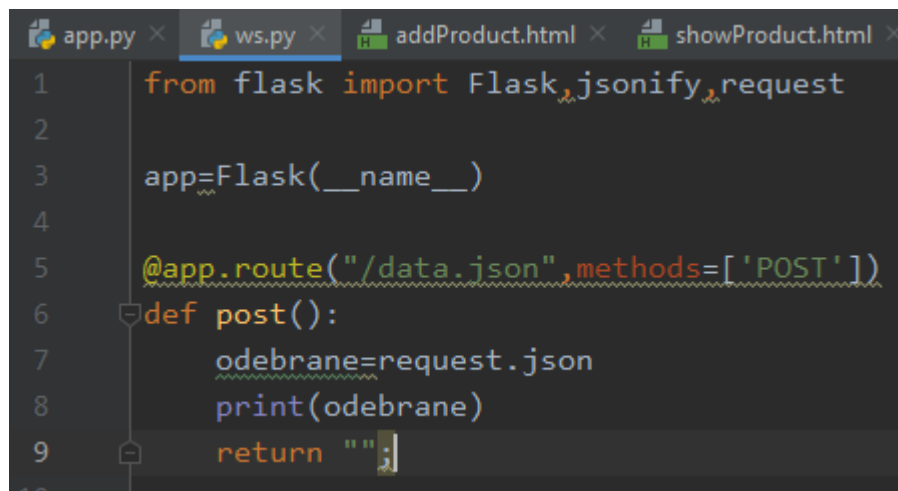
Zamiast deklarować słownik, mogę się też posłużyć funkcją jsonify serializującą dane. Muszę ją jednak uprzednio zaimportować. Korzystając z niej mogę osiągnąć taki sam efekt, ale bez deklaracji słownika:



```
1 from flask import Flask, jsonify
2
3 app=Flask(__name__)
4
5 # dane={
6 #     "imie": "Andrzej",
7 #     "nazwisko": "Klusiewicz",
8 #     "wiek": 33
9 # }
10
11 @app.route("/data.json")
12 def getAllJson():
13     # return dane
14     return jsonify(imie="Andrzej", nazwisko="Klusiewicz", wiek=33)
15
16
17 if __name__ == "__main__":
18     app.run(debug=True, port=80)
```

Usługi sieciowe przyjmujące dane

Usługi sieciowe mogą nie tylko zwracać dane, ale również je przyjmować. Na początek wariant „minimum”:



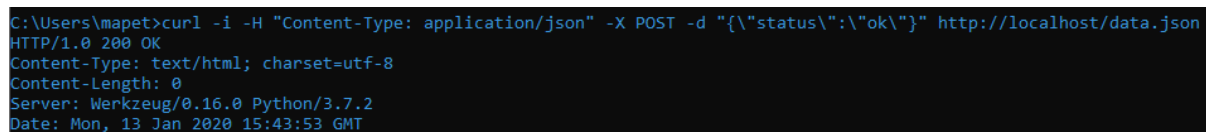
```
1 from flask import Flask, jsonify, request
2
3 app=Flask(__name__)
4
5 @app.route("/data.json", methods=['POST'])
6 def post():
7     odebrane=request.json
8     print(odebrane)
9     return ""
```

Różnice w stosunku do zwracania danych – linia 5 -argument „methods” tak jak w przypadku formularzy i na tej samej zasadzie, oraz w linii 7 tajemnicze „request.json”. To są po prostu przesłane do nas dane. Spróbujemy teraz wywołać naszą usługę sieciową z pomocą curl.

CURL nie jest natywną aplikacją dla systemu Windows, więc jeśli jej nie masz, musisz ją zainstalować i dodać do zmiennej środowiskowej „PATH”. Wywołanie naszej usługi sieciowej z przekazaniem danych, za pomocą CURL wygląda tak:

```
curl -i -H "Content-Type: application/json" -X POST -d '{"status":"ok"}' http://localhost/data.json
```

Teraz co tu jest co. Przetłacznik „-i” służy temu by w odpowiedzi dostać nagłówki http. Znajdują się w nim takie rzeczy jak nazwa serwera, data utworzenia dokumentu etc. „-H” po nim następuje header który jest zawierany w żądaniu wysłanym do usługi sieciowej. W naszym przypadku informujemy go, że otrzyma dane typu JSON (mógłby być choćby XML). Bez dodania tego usługa sieciowa ma „None” w miejscu spodziewanych danych. „-X POST” to oczywiście nazwa użytej metody http, „-d” oznacza że po nim nastąpią dane do przesłania. Dane muszą być w formacie JSON. Tutaj użyłem też „\” mających za zadanie wyłączyć spośród znaków specjalnych znaj ‘ ’ ’, ponieważ na Windowsie pojedyncze „ ’ ” nie działają poprawnie. Ostatni argument to adres na który wysyłam dane. Poniżej zrzut z konsoli:



```
C:\Users\mapet>curl -i -H "Content-Type: application/json" -X POST -d '{"status":"ok"}' http://localhost/data.json
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 0
Server: Werkzeug/0.16.0 Python/3.7.2
Date: Mon, 13 Jan 2020 15:43:53 GMT
```

Konsola Python:

```
127.0.0.1 - - [13/Jan/2020 16:43:14] "POST /data.json HTTP/1.1" 200 -  
{'status': 'ok'}  
127.0.0.1 - - [13/Jan/2020 16:43:53] "POST /data.json HTTP/1.1" 200 -
```

Poprawnie odebraliśmy dane. Przez Pythona dane typu JSON są traktowane jako słowniki.

W tej chwili tylko odebraliśmy dane, w żaden sposób nie odpowiedzieliśmy na żądanie. Klient będzie oczekiwał na potwierdzenie poprawności odebrania danych, dlatego warto mu odesłać status 201 oznaczający właśnie to. Sprowadza się to do całkiem niedużej przeróbki funkcji obsługującej żądanie:

```
5 @app.route("/data.json", methods=['POST'])  
6 def post():  
7     odebrane=request.json  
8     print(odebrane)  
9     return jsonify({'status': True}), 201
```

Zwracane dane są odbierane przez klienta, można więc zwrócić np ID zapisywanego do bazy obiektu:

```
5 @app.route("/data.json", methods=['POST'])  
6 def post():  
7     odebrane=request.json  
8     print(odebrane)  
9     return jsonify({'status': True, "id": 123456}), 201
```

Po stronie klienckiej odebrane dane wyglądają tak:

```
C:\Users\mapet>curl -i -H "Content-Type:
HTTP/1.0 201 CREATED
Content-Type: application/json
Content-Length: 38
Server: Werkzeug/0.16.0 Python/3.7.2
Date: Mon, 13 Jan 2020 16:33:31 GMT

{
  "id": 123456,
  "status": true
}
```

ORM – SQLAlchemy

SQLAlchemy pozwala na translację pomiędzy strukturą relacyjną w relacyjnych bazach danych, a strukturą obiektową i odwrotnie. Jest to zdecydowanie wygodniejsze rozwiązanie niż używanie takich bibliotek jak cx_Oracle czy pycopg2. Aby rozpocząć pracę z SQLAlchemy musimy zainstalować niezbędny pakiet :

```
pip install flask-sqlalchemy
```

Poza samym modulem potrzebujesz też lokalnej bazy PostgreSQL jeśli chcesz przetestować mój kod. Po zainstalowaniu PostgreSQL trzeba jeszcze utworzyć bazę danych i użytkownika poniższymi komendami z poziomu PGAdmina:

```
create database demo;  
create user demo with password 'demo';
```

Łączenie z bazą i pierwsza encja

Aby rozpocząć pracę z SQLAlchemy trzeba najpierw zainicjalizować podając jej właściwy kontekst:

```
from flask import Flask  
from flask_sqlalchemy import SQLAlchemy  
app = Flask(__name__)  
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://demo:demo@localhost/demo'  
db = SQLAlchemy(app)
```

Obiekt „db” to obiekt który będziemy wykorzystywać odwołując się do bazy. Przyszedł czas na obiektową reprezentację danych relacyjnych. Dla każdej tabelki/widoku tworzymy klasę dziedziczącą po klasie db.Model:

```
class Product(db.Model):  
    __tablename__ = "products"  
    productId = db.Column(db.Integer, name="product_id", primary_key=True)  
    productName = db.Column(db.String, name="product_name", unique=True)  
    productDescription = db.Column(db.String, name="product_description", nullable=True)  
    productPrice = db.Column(db.Numeric, name="product_price", index=True)
```


W powyższym przykładzie użyłem kilku różnorodnych argumentów by zaprezentować co ciekawsze możliwości. Nieco enigmatyczne pole „__tablename__” określa nazwę tabeli jeśli jest inna niż klasa – przydatne gdy w Pythonie używasz camelCase’a a w bazach takiego nazewnictwa. Każde pole tej klasy jest obiektem klasy Column której argumenty inicjalizacyjne wymieniam poniżej:

- Pierwszy argument (np. db.Integer) określa typ danych zawartych w kolumnie. Odpowiednio
 - db.Integer – liczby całkowite
 - db.String – tekst
 - db.Numeric – liczby zmiennoprzecinkowe
- name – określa nazwę kolumny. Jest to parametr opcjonalny, jeśli go nie podasz to SQLAlchemy będzie szukał/tworzył kolumnę o nazwie takiej samej jak nazwa pola.
- primary_key – określa że dana kolumna jest kluczem głównym
- unique – określa że wartości w kolumnie muszą być unikalne. Wiąże się to z założeniem odpowiedniej reguły po stronie bazy danych.
- nullable – określa czy wstawiając wiersze do tabeli dana kolumna może pozostać nieuzupełniona. Uwaga! Jeśli nie określisz jawnie, kolumna przy tworzeniu przez SQLAlchemy otrzyma własność NOT NULL!
- index – określa czy na daną kolumnę przy tworzeniu ma zostać nałożony indeks. Protip: jeśli zamierzasz wyszukiwać dane na podstawie wartości zawartej w tej kolumnie, bądź pobierać dane tylko z niej – koniecznie użyj tej funkcjonalności by poprawić wydajność zapytań.

Jeśli chciałbyś aby tabelki utworzyły się same, wywołaj jednokrotnie metodę:

```
db.create_all()
```

Po jej uruchomieniu w bazie danych powstaną tabelki dla wszystkich klas dziedziczących po db.Model. Skutek po stronie PostgreSQL:

```
> 1.3 Sequences
v Tables (1)
  v products
    v Columns (4)
      product_id
      product_name
      product_description
      product_price
    > Constraints
```

Jeśli jest jakiś problem z kontaktem z bazą, ujawni się na tym etapie.



Tabelka została utworzona przez SQLAlchemy automatycznie. Jeśli nie chcesz tworzyć tabel tylko korzystać z istniejących tabel, zwyczajnie nie wywołuj metody `create_all()`. Gdyby przy zapytaniu okazało się że tabela jednak nie istnieje, otrzymamy wyjątek. Istnieje też metoda `db.drop_all()`, a jej nazwa chyba wyjaśnia wszystko 😊. Na potrzeby dalszych działań wzbogaciłem moją klasę o dwie dodatkowe metody i w całości wygląda ona teraz tak:

```
class Product(db.Model):
    __tablename__ = "products"
    productId = db.Column(db.Integer, name="product_id", primary_key=True)
    productName = db.Column(db.String, name="product_name", unique=True)
    productDescription = db.Column(db.String, name="product_description", nullable=True)
    productPrice = db.Column(db.Numeric, name="product_price", index=True)

    def __init__(self, pn, pd, pp):
        self.productName=pn
        self.productDescription=pd
        self.productPrice=pp

    def __str__(self):
        return f'productId={self.productId}, productName={self.productName}, productDe-
scription={self.productDescription}, productPrice={self.productPrice}'
```

Dodawanie danych do tabeli

Aby dodać dane do bazy należy w pierwszej kolejności utworzyć obiekty encyjne – tj. obiekty które mamy zamiar utracić. W poniższym przykładowym kodzie tworzę 3 tracie obiekty, a następnie przekazuję je jako argument do metody add. Jest to zapis transakcyjny, toteż aby zmiany w bazie były widoczne trzeba zatwierdzić transakcję, co czynimy wywołując metodę commit():

```
def loadSomeData():
    p1 = Product('Bulbulator', 'Robi bul bul', 100)
    p2 = Product('Półś do Jelcza', '3ma koło', 80)
    p3 = Product('Wahacz to taczki', 'Do taczek wyścigowych', 500)
    db.session.add(p1)
    db.session.add(p2)
    db.session.add(p3)
    db.session.commit()
```

Całość opakowałem w funkcję by móc osobno wywołać proces jednorazowo, co też po deklaracji uczyniłem. Tobie proponuję zrobić to samo, zanim przejdziemy do dalszej części.

Pobieranie i filtrowanie danych

Przypomnij sobie deklarację klasy Product. Był tam taki fragment:

```
class Product(db.Model):
```

oznaczający że nasza klasa dziedziczy po klasie Model. Dzięki temu dziedziczeniu użytkujemy kilka ciekawych możliwości. Między innymi nasze obiekty będą posiadały metody umożliwiające pobieranie danych z bazy:

```
def getAllProducts():
    return Product.query.all()
```

Aby pobierać dane z bazy należy zatem po nazwie klasy podać „query” i metodę zwracającą dane. W powyższym przykładzie pokazuję pobieranie wszystkich danych, stosuję zatem metodę all(). Teraz zastosujemy filtrowanie:

```
def getProductsPriceOver(price):
    return Product.query.filter(Product.productPrice>price).all()
```

Tym razem pojawia nam się jeszcze metoda filter. Przyjrzyjmy się jej zawartości. Jest nazwa klasy, nazwa pola i warunek „>price”, gdzie price to argument metody. Funkcja ta zwróci wszystkie produkty o cenie wyższej niż podana przez argument. Gdybyś zechciał dodać kolejne warunki, robisz to w ten sposób:

```
def getProductsPriceOver(price):
    return Product.query.filter(Product.productPrice>price).filter(1==1).all()
```

Czyli po kropce stosujesz kolejne wywołania metody filter. Cały czas mówimy o zwracaniu listy elementów. Co jednak jeśli zechcemy zrobić funkcję zwracającą zawsze jeden obiekt. Nie ma sensu opakowywać go w listę poprzez metodę all(). W sytuacji gdy wiemy że zawsze będzie zwracany dokładnie jeden obiekt (bo filtrujemy z użyciem unikalnej kolumny) możemy wykorzystać metodę „first()”:

```
def getOneProductById(productId):  
    return Product.query.filter(Product.productId==productId).first()
```

Sortowanie wyniku

Do sortowania wyniku stosujemy metodę „order_by”. Umożliwia ona sortowanie rosnące i malejące. Jeśli chcesz posortować dane rosnąco robisz to w ten sposób:

```
def getAllProductsOrdered():  
    return Product.query.order_by(Product.productPrice).all()
```

Należy podać nazwę pola po którym dane mają być sortowane. Gdybyś wolał sortowanie malejące:

```
def getAllProductsOrdered():  
    return Product.query.order_by(Product.productPrice.desc()).all()
```

Filtracja i sortowanie w jednym

Metody filter i order_by można też stosować jednocześnie, należy jedynie pamiętać o właściwej kolejności:

```
def getProductsPriceOver(p):  
    return  
Product.query.filter(Product.productPrice>p).order_by(Product.productPrice).all()
```

Zmiana istniejących w bazie danych

Ciekawostka – jaką funkcją aktualizujemy dane? Otóż jest to ta sama funkcja której używaliśmy do ich dodawania...



**Stosowanie
eseknych funkcji
do aktualizacji i
dodawania nowych
danych**



**Stosowanie jednej
funkcji do
aktualizacji i
dodawania danych**

```
def changePrice(product,newPrice):  
    product.productPrice=newPrice  
    db.session.add(product)  
    db.session.commit()
```

Skąd zatem SQLAlchemy wie czy chcemy stworzyć nowy wpis w bazie czy zaktualizować istniejący? Na podstawie wypełnionego albo nie id obiektu. Oznaczyliśmy na etapie deklaracji klasy która kolumna jest kluczem głównym:

```
productId = db.Column(db.Integer, name="product_id", primary_key=True)
```

Zatem jeśli chcesz zaktualizować obiekt, uzupełnij jego ID. Jeśli chcesz dodać nowy, podajesz go bez ID. Jeśli zmienisz id w obiekcie i wywołasz na nim zapis, dokonana zostanie aktualizacja ID a nie dodanie nowego obiektu.

Kasowanie danych

Tu nie ma nic zaskakującego. Jak zapewne po zapoznaniu się z poprzednimi przykładami można się domyślić, istnieje metoda delete umożliwiająca nam takie działanie:

```
def deleteProduct(product):  
    db.session.delete(product)  
    db.session.commit()
```

SQLAlchemy rozpoznaje obiekt do skasowania na podstawie wartości w kluczu głównym, podobnie jak w przypadku aktualizacji.

Podglądanie generowanego SQLa

Zdarzają się sytuację że dostajemy od SQLAlchemy jakieś dziwne wyjątki i nie możemy zdiagnozować źródła problemu. W takich sytuacjach bardzo pomaga możliwość zweryfikowania generowanego przez SQLAlchemy SQLa. W tym celu zamiast od razu wywoływać metodę all(), odbierzemy najpierw obiekt zapytania. Jego reprezentacja po wydrukowaniu pokazuje właśnie wygenerowanego SQLa:

```
def showMeSQL(price):  
    q=Product.query.filter(Product.productPrice>price)  
    print(q)  
    return q.all()
```

```
showMeSQL(40)
```

Zapytanie które otrzymałem:

```
SELECT products.product_id AS products_product_id, products.product_name AS  
products_product_name, products.product_description AS products_product_description,  
products.product_price AS products_product_price  
FROM products  
WHERE products.product_price > %(product_price_1)s
```

Osadzanie aplikacji Flask na Dockerze

W pierwszej kolejności przygotujmy sobie kod aplikacji Flask:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

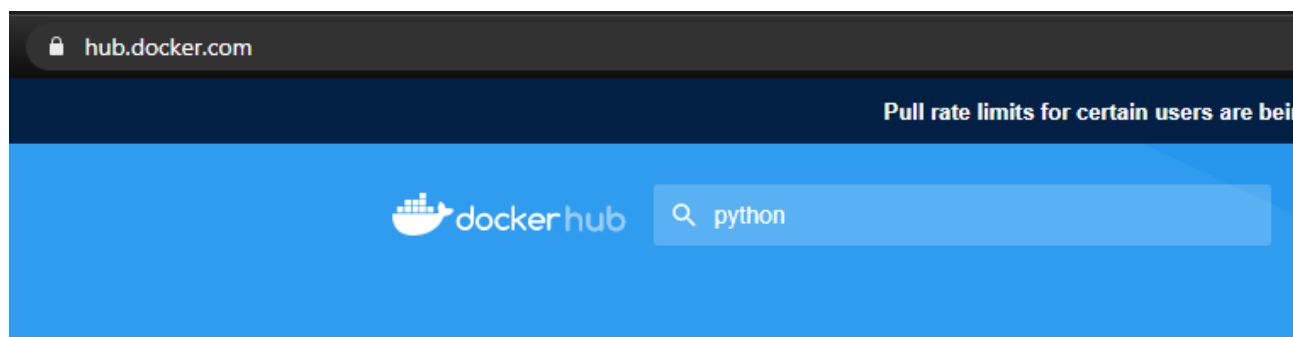
if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True, port=80)
```

Kod ten umieściłem w pliku app.py znajdującym się w głównym katalogu projektu. Jedynym zadaniem aplikacji jest słuchanie na porcie 80 i wyświetlenie "Hello World!" po wejściu na stronę główną aplikacji. Ważne jest by w wywołaniu app.run posłużyć się przełącznikiem host i podać adres, ponieważ bez tego aplikacja nie będzie widoczna spoza kontenera.

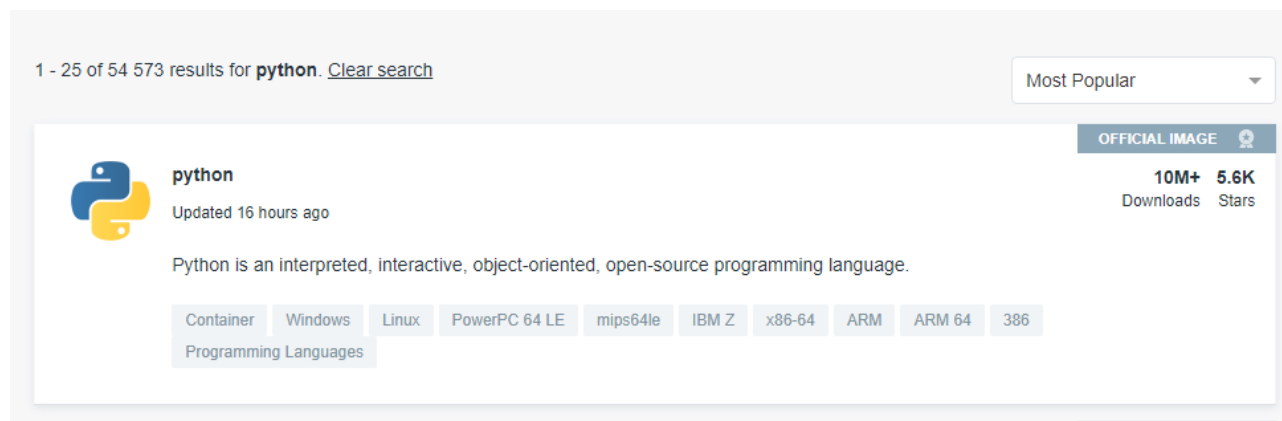
W kolejnym kroku musimy wygenerować plik requirements.txt który będzie nam potrzebny do instalacji zależności w kontenerze. Generujemy go wywołując z poziomu głównego katalogu aplikacji komendę:

```
pip freeze > requirements.txt
```

W dalszym kroku musimy utworzyć plik **Dockerfile** w głównym katalogu projektu. Do pliku tego trzeba będzie wprowadzić kilka informacji na temat sposobu budowania obrazu. Wchodzimy na <https://hub.docker.com/> i wyszukujemy obraz posiadający obsługę Pythona:



Wybieramy przykładowo taki obraz klikając na jego nazwę:



Pod spodem powinna znajdować się taka lista:

Simple Tags

- `3.10.0a2-buster` , `3.10-rc-buster` , `rc-buster`
- `3.10.0a2-slim-buster` , `3.10-rc-slim-buster` , `rc-slim-buster`
- `3.10.0a2-alpine3.12` , `3.10-rc-alpine3.12` , `rc-alpine3.12` ,
- `3.10.0a2-windowsservercore-ltsc2016` , `3.10-rc-windowsservercore-ltsc2016`
- `3.10.0a2-windowsservercore-1809` , `3.10-rc-windowsservercore-1809`

Wybieramy pierwszy wpis i wprowadzamy poniższą linię z wybranym obrazem go do pustego dotychczas pliku **Dockerfile**:

FROM python:3.10.0a1-buster

Linia ta informuje Dockera o obrazie który ma zostać wykorzystany do budowy naszego obrazu. Nie musisz przejmować się pobieraniem żadnego pliku związanego z tym obrazem. Docker pobierze go sam.

Musimy wskazać jeszcze czynności jakie mają zostać wykonane w celu uruchomienia aplikacji. Wprowadzamy do **Dockerfile** poniższe linie:

```
COPY requirements.txt .  
RUN pip install -r requirements.txt  
COPY app.py .  
CMD ["python","app.py"]
```

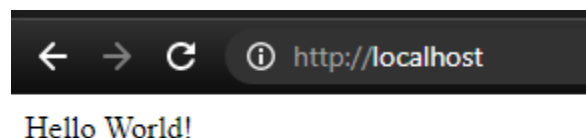
Zostaną one wykonane w chwili budowania obrazu. Instrukcja COPY powoduje skopiowanie wskazanego pliku (w tym przypadku pliku z listą wymaganych zależności) do obrazu. Kolejna instrukcja RUN powoduje instalację wymaganych zależności przez pip z pliku requirements.txt który stworzyliśmy przy pomocy instrukcji "pip freeze > requirements.txt" wykonany wcześniej. Kolejne wykonanie komendy COPY to skopiowanie pliku w którym znajduje się nasz program. Ostatnia instrukcja CMD uruchamia pythona nakazując mu wykonanie pliku "app.py". Musimy teraz utworzyć nasz obraz na podstawie pliku **Dockerfile**. Robimy to wywołując w linii poleceń:

```
docker build -t myrest .
```

"myrest" w powyższej komendzie to nazwa pod jaką później widoczny będzie nasz obraz. Kropka na końcu informuje go o położeniu pliku **Dockerfile**. Po uruchomieniu powinny pojawiać się linie informujące o zakończeniu tworzenia kolejnych etapów budowy obrazu. Przyszedł czas na uruchomienie kontenera:

```
docker run -p 80:80 myrest
```

Po uruchomieniu możemy wejść przez przeglądarkę na naszą aplikację:



Komendę tą możesz wykonać z dowolnego miejsca, ponieważ docker uruchamia obraz po nazwie - w tym przypadku "myrest". Jest to nazwa obrazu którą mu nadaliśmy podczas jego budowania. Pewnego wyjaśnienia może wymagać "-p 80:80". Pierwsza wartość liczbowa to port na którym apka ma być widoczna na zewnątrz, a druga to port na jakim chodzi apka wewnątrz kontenera.

Listę zbudowanych obrazów możesz zobaczyć wywołując komendę:

docker image ls

```
(venv) D:\data\workspace-python\sampleRest>docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
myrest	latest	3e72d7855127	17 minutes ago	895MB
samplerest	latest	cd77e4ac52b0	12 days ago	895MB
registry.heroku.com/docker-flask/web	latest	cd77e4ac52b0	12 days ago	895MB
<none>	<none>	44d59fa118c1	12 days ago	895MB
hello-world	latest	bf756fb1ae65	10 months ago	13.3kB

Listę uruchomionych kontenerów możesz zobaczyć wywołując komendę :

docker container ls

```
(venv) D:\data\workspace-python\sampleRest>docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
c38edbd09f81	myrest	"python app.py"	11 minutes ago	Up 11 minutes	0.0.0.0:80->80/tcp	quirky_leakey

Chcąc zatrzymać uruchomiony kontener potrzebujemy jego container_id z powyższej listy, a przynajmniej jego unikalnego początku:

docker stop c38

Parsowanie stron internetowych z użyciem BeautifulSoup

Beautiful Soup transformuje dokumenty HTML do postaci drzewa obiektów Python (z dokumentacji : <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>).

Instalacja pakietu

Pakiet BeautifulSoup 4 będziemy zwykle wykorzystywać w połączeniu z pakietem requests:

```
import requests as req
from bs4 import BeautifulSoup
```

Obiekt klasy BeautifulSoup

Na potrzeby tego kursu przygotowałem plik html dostępny pod adresem:

<http://jsystems.pl/static/data/pnl/dane.html>

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6
```

Tę właśnie stronę będziemy przetwarzać z użyciem biblioteki BeautifulSoup. Cała praca na stronie będzie się odbywać z użyciem obiektu klasy BeautifulSoup. Obiekt ten będzie reprezentował zawartość strony internetowej podczas jej przetwarzania. Przyjmuje on jako argument „konstruktora” tekst html strony. Podajemy mu ją jako tekst pobrany wcześniej za pomocą funkcji get biblioteki requests. Jako drugi argument „konstruktora” podajemy rodzaj parsera. Z użyciem tej biblioteki możemy parsować również pliki XML. W przedostatniej linii ustawiam jeszcze kodowanie. Na sąsiedniej stronie znajdziesz kod html który będziemy przetwarzać.

```
<html>
<head>
  <title>Tytuł strony!</title>
  <meta name="Author" content="Andrzej Klusiewicz"/>
</head>
<body>
<div id="calosc" class="bazowa">
  <div id="sekcja1" class="podsekcja">
    <p>
      Pierwszy akapit
    </p>
    <p>
      Drugi akapit
    </p>
    <p>Akapit z listą<br>
    <ul>
      <li>marchewka</li>
      <li>kijek</li>
      <li>przyspieszcz cząstek Bozonu Higgsa</li>
    </ul>
    </p>

  </div>
  <div id="sekcja2" class="podsekcja">
    Zawartość sekcji numer 2
  </div>
  <div id="sekcja3">
    <div>
      <p>Coś ciekawego przed
      <h3>OMG!</h3> i coś ciekawego po</p>
    </div>
  </div>
  <div id="sekcja4">
    <ul>
      <li class="punkty">punkt 1</li>
      <li class="punkty">punkt 2</li>
      <li class="punkty">punkt 3</li>
      <li class="punkty">punkt 4</li>
    </ul>
    <table id="tabelka" border="1" width="50%" class="windows95">
      <tr>
        <td>1</td>
        <td>Antarktyda</td>
      </tr>
      <tr>
        <td>2</td>
        <td>Syberia</td>
      </tr>
      <tr>
        <td>3</td>
```

```

        <td>Sosnowiec</td>
    </tr>
</table>
</div>
<div name="stopka">To jest stopka</div>
</div>
</body>
</html>

```

Wyszukiwanie elementów i funkcja find

Wyszukiwanie pierwszego wystąpienia

Obiekt klasy BeautifulSoup (na który odąd będę mówił zupa z racji że tak nazwałem zmienną) umożliwia wyszukiwanie elementów strony wg różnorodnych kryteriów. Korzystając z nazw typów elementów – jak H1, TABLE, UL – możemy odnajdywać pierwsze wystąpienie takich elementów:

```

1  import requests as req
2  from bs4 import BeautifulSoup
3  strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4  strona.encoding='utf-8'
5  zupa=BeautifulSoup(strona.text,'html.parser')
6  print(zupa.h3)
7  print(type(zupa.h3))

```

Wewnątrz div od id „sekcja3” znajduje się nagłówek H3 o treści „OMG!”. Powyższy kod pobiera pierwsze wystąpienie elementu h3 w całym dokumencie, a jest to właśnie wspomniany nagłówek. Przy okazji sprawdzam i drukuję również jego klasę:

```

↑ D:\data\workspace-python\beaut:
↓ <h3>OMG!</h3>
⋮ <class 'bs4.element.Tag'>
↑

```

Na elementach klasy Tag można wykonywać takie same operacje przeszukiwania jak na „zupie”. Toteż umożliwia to dalsze wchodzenie w głąb struktury. Równoznaczne byłoby zastosowanie funkcji find:

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 print(zupa.find('h3'))
7 print(type(zupa.find('h3')))
```

Jeśli jako argument funkcji find podamy rodzaj elementu, zostanie nam zwrócone pierwsze wystąpienie takiego elementu w pobranej stronie lub jej części.

Wyszukiwanie po id elementu

Jeśli element do którego chcemy sięgnąć posiada ID możemy odnaleźć go stosując argument id dla funkcji find:

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 print(zupa.find(id='sekcja2'))
```

Zwrócone dane:

```
<div class="podsekcja" id="sekcja2">
    Zawartość sekcji numer 2
</div>
```

Wyszukiwanie po klasie css

Funkcja find przyjmuje również argument class_ umożliwiający odniesienie się do klasy css wyszukiwanego elementu:

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 print(zupa.find(class_='podsekcja'))
```

W sytuacji gdyby kilka elementów posiadało tę samą klasę (co jest całkiem naturalne) funkcja zwróci pierwsze wystąpienie takiego elementu. W związku z tym, wynik działania skryptu wygląda tak:

```
D:\data\workspace-python\beautifullsouptutorial
<div class="podsekcja" id="sekcja1">
<p>
    Pierwszy akapit
</p>
<p>
    Drugi akapit
</p>
<p>Akapit z listą<br/>
<ul>
<li>marchewka</li>
<li>kijek</li>
<li>przyspieszacz cząstek Bozonu Higgsa</li>
</ul>
</p>
</div>
```

Wyszukiwanie po atrybutach elementu

Jeśli element nie posiada id ani nie używa klasy css a chcielibyśmy mieć do niego jakiś uchwyt, możemy posłużyć się którymś z jego atrybutów. Taki element znajduje się na końcu wspomnianego na początku rozdziału dokumentu HTML:

```
</table>
</div>
<div name="stopka">To jest stopka</div>
</div>
</body>
</html>
```

To może być zresztą jakikolwiek inny atrybut. Funkcja find posiada argument attrs. Podajemy do niego słownik z nazwami atrybutów i charakteryzującymi je wartościami:

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 print(zupa.find(attrs={'name':'stopka'}))
```

Wynik działania:

```
D:\data\workspace-python\beautifulsoup>
<div name="stopka">To jest stopka</div>

Process finished with exit code 0
```


Zagnieżdżanie

Każdy element zwracany przez funkcję `find`, ale również opisywane wcześniej uchwytty odnoszące się do nazw tagów html będą zwracały obiekt klasy `Tag`. Taki obiekt pozwala zagnieżdżać się dalej. W zasadzie nie ma tu ograniczeń ilościowych. Poniżej przykład takiego zagnieżdżania:

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 h3=zupa.find(id="sekcja3").div.p.h3
7 print(h3)
```

Wynik:

```
↑ D:\data\workspace-python\beautiful
↓ <h3>OMG!</h3>
:|:
:~: Process finished with exit code 0
```

Sięganie do sekcji strony

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 print(zupa.title)
7 print(zupa.head)
8 print(zupa.body)
```

Podobnie jak mogliśmy wyszukiwać elementy po nazwach tagów – H1, UL, DIV etc tak możemy sięgać do nagłówka , tytułu strony czy jego body. Wyniku działania tego skryptu z czystej przyzwoitości nie przytoczę 12.

Atrybuty elementów

Obiekty klasy BeautifulSoup i Tag posiadają pole attrs zawierające w postaci słownika atrybuty elementu:

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 tab=zupa.find('table')
7 print(tab.attrs)
8 |
```

Analizowany przypadek to tabelka z takimi atrybutami:

```
</ul>
<table id="tabelka" border="1" width="50%" class="windows95">
  <tr>
```

Dane wyciągnięte z attrs:

```
D:\data\workspace-python\beautifulsoup4\tutorial\venv\Scripts\python.exe D:/
{'id': 'tabelka', 'border': '1', 'width': '50%', 'class': ['windows95']}

Process finished with exit code 0
```

Skoro to słownik to możemy go tak właśnie przetwarzać. Dopisuję jeszcze jedną linijkę kodu do skryptu by dostać się do wartości dla klucza „width”:

```
7 print(tab.attrs)
8 print(tab.attrs['width'])
9
```

Wynik:

```
D:\data\workspace-python\beautifuls
{'id': 'tabelka', 'border': '1', 'wi
50%

Process finished with exit code 0
```

name i string

Każdy element ma name, nie każdy ma string. Name służy do sprawdzania nazwy elementu. Może to być użyteczne do sprawdzania jakiego rodzaju jest element wyszukany np. po id albo klasie. String do zawartość elementu – tekst z pomiędzy tagów początkowego i końcowego. Uwaga – nie każdy element będzie miał tam jakąś wartość – nie będą jej miały elementy będące jedynie kontenerami.

Dla przykładu sięgnąłem do pierwszego elementu będącego tabelą i sprawdziłem jego typ. Następnie sięgnąłem do pierwszego akapitu i wyświetliłem jego zawartość:

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 tab=zupa.find('table')
7 print(tab.name)
8 p=zupa.find('p')
9 print(p.string)
```

Wynik:

```
D:\data\workspace-python\beautifullsc
table

Pierwszy akapit
```

Operowanie na listach elementów i funkcja find_all

Dotychczas operowaliśmy funkcją find, bądź odnosiliśmy się do pierwszych wystąpień elementów. Za każdym razem jednak pracowaliśmy z pojedynczym obiektem. Teraz przyszedł czas na przetwarzanie list elementów. Dla funkcji find_all działają te same filtry co dla funkcji find. Możesz wyszukiwać elementy w oparciu o ich id, klasę css czy atrybuty. Różnica polega jednak na tym, że tym razem nie dostaniemy w wyniku obiektu klasy Tag, a ResultSet. Jest to implementacja wzorca projektowego Iterator.

Na początek użyjemy funkcji find_all bez jakichkolwiek argumentów. W efekcie dostaniemy listę wszystkich elementów (pierwszego poziomu zagnieżdżenia ale i tych głębiej) występujących w dokumencie. Iteruję po zwróconej liście i wyświetlam nazwy typu elementu:

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 for d in zupa.find_all():
7     print(d.name)
```

Wynik działania skryptu:

```
D:\data\workspace-python\beautifull
html
head
title
meta
body
div
div
p
p
p
br
```

Uciąłem sporą część ze względu na objętość jaką zajmował cały wynik. Widzimy tu dosłownie wszystkie elementy – od strukturalnych jak head do akapitów.

Filtrowanie elementów z funkcją find_all

Podobnie jak funkcja find, funkcja find_all przyjmuje różnorakie argumenty. Wszystkie argumenty omawiane dla funkcji find mają zastosowanie również tutaj. Poniżej przykład wyszukiwania i wyświetlania tylko elementów będących punktami listy:

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 for li in zupa.find_all('li'):
7     print(li)
```

W sumie w dokumencie mamy dwie listy, ale w powyższym przykładzie nie ma to znaczenia. Wyciągneliśmy wszystkie elementy li występujące w dokumencie:

```
D:\data\workspace-python\beautifullsouptutorial
<li>marchewka</li>
<li>kijek</li>
<li>przyspieszacz cząstek Bozonu Higgsa</li>
<li class="punkty">punkt 1</li>
<li class="punkty">punkt 2</li>
<li class="punkty">punkt 3</li>
<li class="punkty">punkt 4</li>

Process finished with exit code 0
```

Jeśli chcielibyśmy wyciągnąć tylko punkty zawarte na liście znajdującej się w div o id „sekcja4” to zgodnie z zasadami zagnieżdżania omówionymi wcześniej robimy to w ten sposób:

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 for li in zupa.find(id='sekcja4').find_all('li'):
7     print(li)
```

Wynik:

```
↑ D:\data\workspace-python\beautifuls
↓ <li class="punkty">punkt 1</li>
  <li class="punkty">punkt 2</li>
  <li class="punkty">punkt 3</li>
  <li class="punkty">punkt 4</li>
  Process finished with exit code 0
```

Alternatywnie można ten sam efekt osiągnąć wyszukując elementy wg klasy css. Dla poniższego kodu wynik jest identyczny z powyższym.

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 for li in zupa.find_all(class_='punkty'):
7     print(li)
```

contents

Pole contents pozwala nam zaglądać do elementów zawartych w konterze. W naszym kodzie html mamy tabelkę:

```
<table id="tabelka" border="1" width="50%" class="windows95">
  <tr>
    <td>1</td>
    <td>Antarktyda</td>
  </tr>
  <tr>
    <td>2</td>
    <td>Syberia</td>
  </tr>
  <tr>
    <td>3</td>
    <td>Sosnowiec</td>
  </tr>
</table>
```

Chcemy docelowo dostać się do tekstu „Antarktyda”. Jest to druga kolumna drugiej linii tabeli. Zobaczmy więc co znajduje się w polu contents tabeli, ile jest tam elementów i jakiego typu dane znajdziemy w contents:

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 lista=zupa.find(id='tabelka').contents
7 print(len(lista))
8 print(type(lista))
9 print(lista)
```


Powyższy kod zwraca nam informację o 7 elementach. Jest to kolekcja typu lista, a zawartość tej listy to kolejne wiersze tabeli, oraz znaki „enter”:

```
D:\data\workspace-python\beauti
7
<class 'list'>
['\n', <tr>
<td>1</td>
<td>Antarktyda</td>
</tr>, '\n', <tr>
<td>2</td>
<td>Syberia</td>
</tr>, '\n', <tr>
<td>3</td>
<td>Sosnowiec</td>
</tr>, '\n']
```

Szukamy gdzie ta nasza Antarktyda. Znajduje się ona w elemencie o indeksie 1. Wydrukujmy więc ten element:

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 lista=zupa.find(id='tabelka').contents
7 print(lista[1])
8
```

Dobraliśmy się do wiersza:

```
D:\data\workspace-python\be
<tr>
<td>1</td>
<td>Antarktyda</td>
</tr>
```

Taki wiersz również ma składowe – kolumny. Zobaczmy jak BS4 to rozbije:

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 lista=zupa.find(id='tabelka').contents
7 print(lista[1].contents)
8
```

Wynik działania:

```
↑ D:\data\workspace-python\beautifullsouptutorial\venv\Script
↓ ['\n', <td>1</td>, '\n', <td>Antarktyda</td>, '\n']
```

Nasz wiersz składa się zatem z kolumn oraz znaków „enter” będących pozostałością po formatowaniu dokumentu. Fraza „Antarktyda” znajduje się w elemencie o indeksie 3. Zajrzyjmy więc do niego:

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 lista=zupa.find(id='tabelka').contents
7 print(lista[1].contents[3])
8
```

Wynik:

```
↑ D:\data\workspace-python\beautifulls
↓ <td>Antarktyda</td>
⏏ Process finished with exit code 0
⇓
```

Teraz możemy odnieść się do „string” aby dostać się do poszukiwanej frazy:

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 lista=zupa.find(id='tabelka').contents
7 print(lista[1].contents[3].string)
8
```

Wynik:

```
↑ D:\data\workspace-python\beautiful
↓ Antarktyda
: Process finished with exit code 0
⇅
```

Zamiast rozbijać wyszukiwanie na atomy, możemy skorzystać z zagnieżdżenia:

```
1 import requests as req
2 from bs4 import BeautifulSoup
3 strona=req.get('http://jsystems.pl/static/data/pnl/dane.html')
4 strona.encoding='utf-8'
5 zupa=BeautifulSoup(strona.text,'html.parser')
6 print(zupa.find(id='tabelka').contents[1].contents[3].string)
7
8
```

Dającego nam dokładnie ten sam wynik. Tłumacząc wynik wyszukiwania z linii 6 kodu: Znajdź element o id równym „tabelka”, sięgnij do drugiego (o indeksie 1) z zawartych w nim elementów, z tego elementu wyciągnij czwarty (o indeksie 3) jego podelement, a z niego wypuść zawartość.

Moduł click – obsługa parametrów z linii poleceń

Stosowanie parametrów

Moduł „click” pozwala odbierać od użytkownika parametry przekazywane do skryptu. Dawniej był stosowany moduł „argparse” do tego samego celu, jednak był niewygodny i mało intuicyjny toteż obecnie używany jest raczej „click”. Zaczniemy od prostego skryptu przyjmującego jeden parametr.

```
import click

@click.command()
@click.option('-i')
def witacz(i):
    print(f'Witaj {i}')

witacz()
```

To jest cała zawartość mojego pliku app.py. Funkcja witacz raczej nie należy do złożonych – przyjmuje przez argument imię osoby którą ma powitać. Magia dzieje się dzięki dekoratorom @click.command() i @click.option(). Pierwsza informuje moduł click że funkcja witacz jest przez niego obsługiwana. Drugi określa nazwę parametru którego użyjemy przy wywoływaniu skryptu – nazwa tego parametru musi się pokrywać z nazwą parametru wywoływanej funkcji. Wywołanie skryptu:

```
python app.py -i Andrzej
```

Efekt na konsoli:

```
Witaj Andrzej
```

Jeśli podkusiła Cię ciekawość i sprawdziłeś co się stanie jeśli nie podamy parametru, to zobaczyłeś:

```
Witaj None
```

Oczywiście nie jest to efekt jaki chcielibyśmy uzyskać. Jak więc zmusić click do żądania wartości tego parametru w przypadku jego niepodania? Do dekoratora `@click.option` dodajemy jeszcze argument „prompt”. W przypadku nie podania wartości dla parametru zostaniemy o to poproszeni:

```
import click

@click.command()
@click.option('-i', prompt=True)
def witacz(i):
    print(f'Witaj {i}')

witacz()
```

Alternatywnie możemy zastosować argument „required”:

```
import click

@click.command()
@click.option('-i', required=True)
def witacz(i):
    print(f'Witaj {i}')

witacz()
```

Różnica sprowadza się do tego, że w pierwszym przypadku zostaniemy poproszeni o podanie wartości dla argumentu, w drugim dostaniemy komunikat tego typu:

```
Usage: app.py [OPTIONS]
Try 'app.py --help' for help.
Error: Missing option '-i'.
```

A skrypt nie jest uruchamiany.

Stosowanie wielu parametrów

Aby zastosować wiele argumentów na raz wystarczy dodać kilka dekoratorów `@click.option`:

```
import click

@click.command()
@click.option('-i')
@click.option('-n')
def witacz(i,n):
    print(f'Witaj {i} {n}')

witacz()
```

Skrypt wywołujemy tak:

```
python app.py -i Andrzej -n Klusiewicz
```

Efekt na konsoli:

```
Witaj Andrzej Klusiewicz
```

Kolejność podawania parametrów nie ma znaczenia.

Automatyczne generowanie pomocy

Click automatycznie dodaje parametr „—help”, po zastosowaniu którego wyświetla się pomoc. Treść pomocy deklarujemy przy użyciu argumentu dekoratora option w ten sposób:

```
import click

@click.command()
@click.option('-i', help='Imię witanej osoby')
@click.option('-n', help='Nazwisko witanej osoby')
def witacz(i,n):
    print(f'Witaj {i} {n}')

witacz()
```

Wywołanie skryptu w ten sposób:

```
python app.py --help
```

Spowoduje wyświetlenie pomocy:

Usage: app.py [OPTIONS]

Options:

- i TEXT** Imię witanej osoby
- n TEXT** Nazwisko witanej osoby
- help** Show this message and exit.

Wprowadzanie haseł

Przykład znaleziony w dokumentacji, ale jest tak fajny że postanowiłem go tu umieścić ku uciesze administratorów 😊. Często pojawiający się w pracy (mojej) problem to tajność haseł pojawiających się w kodzie źródłowym. Można ten problem rozwiązać na kilka sposobów, ale jednym z nich jest wykorzystanie funkcjonalności modułu click. Wyobraźmy sobie że w skrypcie jest jawnie zawarte hasło do bazy danych. Zamiast je podawać jawnie, możemy nie tylko przyjąć je przez parametr, ale jednocześnie ukryć wprowadzane znaki. Ukrywanie wprowadzanych znaków zapewnia nam argument „hide_input”. Możesz też (opcjonalnie) wykorzystać argument „confirmation_prompt” który powoduje konieczność potwierdzenia wprowadzonej wartości. Bardzo wygodne w sytuacjach gdy nasz skrypt ma na przykład ustawiać jakieś hasło.

```
import click
@click.command()
@click.option('--password', prompt=True, confirmation_prompt=True, hide_input=True)
def connectDatabase(password):
    print(f'connecting to database. Url: user/{password}@jsystems.pl/databaseName')

connectDatabase()
```

Efekt działania powyższego kodu:

```
python app.py
Password:
Repeat for confirmation:
connecting to database. Url: user/moje_haslo@jsystems.pl/databaseName
```


Moduł ipaddress

Moduł ipaddress pozwala na sprawdzanie czy dany ciąg jest adresem IP, czy jest w sieci publicznej czy prywatnej etc. Aby rozpocząć operacje na adresie IP musimy najpierw stworzyć reprezentujący go obiekt:

```
import ipaddress
a=ipaddress.ip_address("10.0.0.1")
```

Na tym etapie wykonywana jest walidacja adresu. Gdybys podał coś co adresem IP nie jest, jak na przykład tu:

```
import ipaddress
a=ipaddress.ip_address("there's no place like localhost...")
```

otrzymamy wyjątek z komunikatem:

ValueError: "there's no place like localhost..." does not appear to be an IPv4 or IPv6 address

Ipaddress dla poprawnie zwalidowanego adresu udostępnia szereg ciekawych funkcjonalności. Możemy np. weryfikować czy dany adres leży w sieci prywatnej czy publicznej:

```
import ipaddress
a=ipaddress.ip_address('46.41.128.110')
a2=ipaddress.ip_address('192.168.1.1')
print(a.is_private)
print(a2.is_private)
```

Efekt:

False
True

Zamiast pola is_private możemy też użyć is_global:

```
import ipaddress
a=ipaddress.ip_address('46.41.128.110')
a2=ipaddress.ip_address('192.168.1.1')
print(a.is_global)
print(a2.is_global)
```

z oczywiście odwrotnym skutkiem:

True
False

Możemy też porównywać adresy IP:

```
import ipaddress
a=ipaddress.ip_address('46.41.128.110')
a2=ipaddress.ip_address('46.41.128.111')
print(a>a2)
```

Efekt:

False

Jeszcze jedna ciekawostka (tu puszczenie oczka w stronę administratorów) na temat ipaddress. Możemy wylistować wszystkie adresy z danej sieci. Tym razem będziemy musieli utworzyć obiekt innej klasy, dlatego też zamiast funkcji ip_adress potrzebujemy funkcji ip_network:

```
import ipaddress
for h in ipaddress.ip_network('192.168.2.0/24').hosts():
    print(h)
```

Skutek działania:

```
192.168.2.1
192.168.2.2
192.168.2.3
192.168.2.4
192.168.2.5
192.168.2.6
.....
.....
```

Nie są to wszystkie możliwości tego modułu, wybrałem te które moim zdaniem są najbardziej przydatne. Zainteresowanych rozszerzeniem zapraszam do zapoznania się z dokumentacją.

Wirtualne środowisko - VENV

Nie zawsze będziemy pracować z IDE typu Pycharm które wytworzy za nas "automagicznie" środowisko uruchomieniowe. Nawet jeśli tak by było, warto dowiedzieć się jakie mechanizmy za tym stoją.

Aby stworzyć wirtualne środowisko dla naszego projektu musimy wywołać komendę:

python -m venv env

gdzie "env" jest nazwą podkatalogu środowiska wirtualnego. Po wywołaniu tej komendy, w katalogu powinien się pojawić katalog "env" zawierający nasze wirtualne środowisko.

```
D:\venvs>python -m venv env

D:\venvs>dir env
Volume in drive D is Nowy
Volume Serial Number is 80C7-FEC0

Directory of D:\venvs\env

09.11.2020  15:05    <DIR>          .
09.11.2020  15:05    <DIR>          ..
09.11.2020  15:05    <DIR>          Include
09.11.2020  15:05    <DIR>          Lib
09.11.2020  15:05                117 pyvenv.cfg
09.11.2020  15:05    <DIR>          Scripts
                   1 File(s)            117 bytes
                   5 Dir(s)  354 192 990 208 bytes free

D:\venvs>_
```

Aby wejść w nasze wirtualne środowisko aktywujemy je poprzez wywołanie skryptu "activate" (w Windows):

```
D:\venvs>env\Scripts\activate

(env) D:\venvs>_
```

Nasz prompt powinien zmienić zawartość na zasadzie jak powyżej.

Z wirtualnego środowiska możemy wyjść w dowolnym momencie wywołując komendę:

deactivate

Do katalogu głównego dodaję plik "get.py" o następującej treści:

import requests as re

req=re.get('http://onet.pl')

Nie oczekujemy od tego modułu zbyt wiele, ważne jest wykorzystanie biblioteki requests. Jak widać moduł ten wykorzystuje pakiet requests. Spróbujemy go wywołać z konsoli:

```
(env) D:\venvs>python get.py
Traceback (most recent call last):
  File "D:\venvs\get.py", line 1, in <module>
    import requests as re
ModuleNotFoundError: No module named 'requests'

(env) D:\venvs>
```

Dostajemy komunikat o braku modułu "requests" którego używaliśmy w naszym module. Doinstalujemy brakującą bibliotekę wraz z wszystkimi zależnościami wywołując:

pip install requests

Efekt:

```
(env) D:\venvs>pip install requests
Collecting requests
  Using cached requests-2.24.0-py2.py3-none-any.whl (61 kB)
Collecting urllib3!=1.25.0,!1.25.1,<1.26,>=1.21.1
  Using cached urllib3-1.25.11-py2.py3-none-any.whl (127 kB)
Collecting idna<3,>=2.5
  Using cached idna-2.10-py2.py3-none-any.whl (58 kB)
Collecting chardet<4,>=3.0.2
  Using cached chardet-3.0.4-py2.py3-none-any.whl (133 kB)
Collecting certifi>=2017.4.17
  Using cached certifi-2020.11.8-py2.py3-none-any.whl (155 kB)
Installing collected packages: urllib3, idna, chardet, certifi, requests
Successfully installed certifi-2020.11.8 chardet-3.0.4 idna-2.10 requests-2.24.0 urllib3-1.25.11
WARNING: You are using pip version 20.2.3; however, version 20.2.4 is available.
You should consider upgrading via the 'd:\venvs\env\scripts\python.exe -m pip install --upgrade pip' command.
```

W wyniku widzimy że poza requests zostały też dodane biblioteki:

- urllib3
- idna
- chardet
- certifi

Sprawdzić aktualnie zainstalowane biblioteki możemy wywołując :

pip freeze

```
(env) D:\venvs>pip freeze
certifi==2020.11.8
chardet==3.0.4
idna==2.10
requests==2.24.0
urllib3==1.25.11
(env) D:\venvs>
```

Korzystając z "pip freeze" możemy też utworzyć plik służący do instalacji wymaganych zależności:

pip freeze > requirements.txt

Zawartość pliku "requirements.txt" po wywołaniu powyższej komendy:

```
certifi==2020.11.8
chardet==3.0.4
idna==2.10
requests==2.24.0
urllib3==1.25.11
```

Możemy teraz przekazać nasz moduł "get.py" i plik "requirements.txt" innemu programiście w celu uruchomienia. Aby mógł to zrobić musi utworzyć sobie lokalne środowisko venv dla tego projektu, a następnie wywołać:

```
pip install -r requirements.txt
```

Co spowoduje taki efekt:

```
(env) D:\venvs2>pip install -r requirements.txt
Collecting certifi==2020.11.8
  Using cached certifi-2020.11.8-py2.py3-none-any.whl (155 kB)
Collecting chardet==3.0.4
  Using cached chardet-3.0.4-py2.py3-none-any.whl (133 kB)
Collecting idna==2.10
  Using cached idna-2.10-py2.py3-none-any.whl (58 kB)
Collecting requests==2.24.0
  Using cached requests-2.24.0-py2.py3-none-any.whl (61 kB)
Collecting urllib3==1.25.11
  Using cached urllib3-1.25.11-py2.py3-none-any.whl (127 kB)
Installing collected packages: certifi, chardet, idna, urllib3, requests
Successfully installed certifi-2020.11.8 chardet-3.0.4 idna-2.10 requests-2.24.0 urllib3-1.25.11
WARNING: You are using pip version 20.2.3; however, version 20.2.4 is available.
You should consider upgrading via the 'd:\venvs2\env\scripts\python.exe -m pip install --upgrade pip'

(env) D:\venvs2>python get.py

(env) D:\venvs2>
```

Jak widać, po tej operacji mamy już możliwość uruchomienia skryptu "get.py" bez błędu.

Wzorce projektowe

Wzorce projektowe są sprawdzonymi nazwanymi rozwiązaniami powszechnie występujących w programowaniu problemów. Ten opis rozszerzę w kolejnej edycji tej książki. Zostaną też dodane pozostałe wzorce projektowe...

Wzorce kreacyjne

Wzorzec "Singleton"

Singleton to wzorzec kreacyjny. Zapewnia istnienie co najwyżej jednej instancji danej klasy. Instancja ta musi być dostępna z dowolnego miejsca.

Przykład zacznę od zupełnie zwyczajnej klasy, stworzenia 2 jej obiektów i sprawdzenia czy to ten sam obiekt:

```
class Singleton:
    pass

x = Singleton()
y = Singleton()
print(id(x))
print(id(y))
print(x is y)
```

W efekcie dostałem na konsoli wynik:

```
2639965680864
2639965681032
False
```

Id obiektów się różnią, co świadczy o tym że są to różne obiekty. Potwierdza to też sprawdzenie wyrażenia "x is y" w ostatniej linii.

Przerobiłem nieco naszą klasę przesłaniając metodę "__new__" i ponowiłem test:

```
class Singleton:
    __instancja=None
    def __new__(cls, *args, **kwargs):
        if cls.__instancja is None:
            cls.__instancja=super().__new__(cls,*args,**kwargs)
        return cls.__instancja

a=Singleton()
b=Singleton()
print(id(a))
print(id(b))
print(a is b)
```

Wynik na konsoli:

```
2597567231368
2597567231368
True
```

Tym razem otrzymaliśmy te same id i potwierdzenie że jest to ten sam obiekt. Na czym jednak polegał ten fikołek? Co tu się stało? Metoda "__new__" jest wywoływana w chwili tworzenia obiektu. W naszej klasie mamy prywatne pole "__instancja" które będzie zawierało obiekt klasy Singleton. Obiekt naszej klasy "Singleton" będzie zawierał obiekt klasy "Singleton". Teraz zasada działania kodu wewnątrz metody "__new__" jest taka: jeśli nie mam jeszcze zainicjalizowanego obiektu to go tworzymy, a niezależnie czy właśnie go stworzyliśmy czy był już wcześniej zwracamy go. To jest mechanizm który pilnuje istnienia tylko jednego obiektu klasy Singleton.

Znając już mechanizm powinniśmy dotrzeć do prostego wniosku. Jeśli istnieje tylko jeden obiekt klasy Singleton, to w ramach tej klasy mogę stworzyć pole które również będzie przechowywane w jednej instancji. Przeanalizujmy poniższy przykład:

```
class Singleton:

    pole=None
    __instancja=None

    def __new__(cls, *args, **kwargs):
        if cls.__instancja is None:
            cls.__instancja=super().__new__(cls,*args,**kwargs)
        return cls.__instancja

a=Singleton()
b=Singleton()
a.pole='Zawartość pola'
print(b.pole)
```


W związku z zastosowaniem wzorca "Singleton" obiekty "a" i "b" to w rzeczywistości ten sam obiekt. Z tego powodu mimo przypisania "zawartość pola" do pola "pole" w obiekcie "a", wartość ta jest widoczna również w obiekcie "b". Jest to spowodowane wskazywaniem na ten sam adres w pamięci dla obu obiektów.

Zasymulujmy wykorzystanie singletona do utrzymywania tylko jednego połączenia z bazą danych.

```
class DatabaseConnector:
    polaczenie=None
    instancja=None

    def __new__(cls, *args, **kwargs):
        if cls.instancja is None:
            print('nawiązywanie połączenia')
            cls.polaczenie='połączenie do bazy Oracle'
            cls.instancja=super().__new__(cls,*args,**kwargs)
        return cls.instancja

a=DatabaseConnector()
b=DatabaseConnector()
print(b.polaczenie)
```

wynik działania na konsoli:

nawiązywanie połączenia
połączenie do bazy Oracle

W powyższym przykładzie "połączenie" jest zwykłym polem tekstowym, ale nic nie stoi na przeszkodzie byśmy tam umieścili cokolwiek innego. Stworzyłem dwa obiekty. W chwili tworzenia obiektu "a" pole "instancja" było w obiekcie klasy "DatabaseConnector" w związku z czym inicjalizujemy to pole uzupełniając także pole "polaczenie". W chwili tworzenia obiektu "b" pole instancja jest już wypełnione w związku z czym "__new__" oddaje już wcześniej zainicjalizowany obiekt bez jego powtórnego inicjalizowania (czyli również wypełniania pola "polaczenie". Zarówno obiekt "a" jak i "b" referują do tego samego miejsca w przestrzeni pamięci - tak więc "a" i "b" to w rzeczywistości ten sam obiekt.

Wzorzec "Fabryka Abstrakcyjna"

Wzorzec kreatywny. Pozwala tworzyć rodziny obiektów dziedziczących po tej samej klasie bez określania ich konkretnych klas. Wzorzec ten powinniśmy stosować wtedy gdy zawsze chcemy użyć tego samego zestawu metod na dowolnym wybranym obiekcie z danej rodziny (klas mających wspólną klasę bazową po której dziedziczą).

Przyjmijmy że jest kilka fabryk samochodów różnych rodzajów. Jedna fabryka produkuje samochody sportowe, druga limuzyny, trzecia samochody miejskie. Wszystkie samochody, niezależnie od tego jaki jest to samochód i z jakiej fabryki pochodzi powinny posiadać metodę "jedz". Chciałbym móc tworzyć obiekt dowolnej z fabryk i odbierać obiekt dowolnego rodzaju samochodu a następnie wywoływać na obiekcie tego samochodu metodę jedz. Aby zapewnić posiadanie przez wszystkie rodzaje samochodów metody "jedz" tworzę bazową klasę abstrakcyjną ze zdefiniowaną taką metodą:

```
from abc import ABC, abstractmethod
class Samochod(ABC):
    @abstractmethod
    def jedz(self):
        pass
```

W kolejnym kroku tworzę różne klasy reprezentujące różne rodzaje samochodów. Ważne by wszystkie dziedziczyły po klasie "Samochod":

```
class SportowySamochod(Samochod):
    def jedz(self):
        print('sportowy wydech robi wroom!')

class Limuzyna(Samochod):
    def jedz(self):
        print('nawet nie usłyszysz dźwięku silnika...')

class SamochodMiejski(Samochod):
    def jedz(self):
        print('no niby jedzie...')
```

Każda z klas w inny sposób implementuje działanie metody "jedz" ale każda taką metodę posiada, co wynika z konieczności implementacji metody abstrakcyjnej klasy abstrakcyjnej po której dziedziczymy.

Przyszła pora na fabryki. Podobnie jak dowolny z samochodów ma posiadać metodę "jedz", tak każda fabryka powinna posiadać metodę "produkuj_samochod" po której oczekujemy oddania nam obiektu samochodu tworzego przez tę właśnie fabrykę. Dla realizacji tego oczekiwania posłużymy się tą samą metodą co w przypadku klas samochodów. Utworzymy bazową klasę abstrakcyjną dla fabryk i wymusimy implementację metody "produkuj_samochod" we wszystkich dziedziczących po tej klasie klas fabryk:

```
class FabrykaSamochodow(ABC):
    @abstractmethod
    def produkuj_samochod(self):
        pass
```

Implementacja poszczególnych fabryk:

```
class FabrykaSportowych(FabrykaSamochodow):
    def produkuj_samochod(self):
        return SportowySamochod()
```

```
class FabrykaLimuzyn(FabrykaSamochodow):
    def produkuj_samochod(self):
        return Limuzyna()
```

```
class FabrykaMiejskich(FabrykaSamochodow):
    def produkuj_samochod(self):
        return SamochodMiejski()
```

Pozostaje nam już tylko kod wykonawczy który skorzysta z dobrodziejstw stworzonej przez nas struktury:

```
rodzaj='miejski'
if rodzaj=='sportowy':
    fabryka=FabrykaSportowych()
elif rodzaj=='limuzyna':
    fabryka=FabrykaLimuzyn()
elif rodzaj=='miejski':
    fabryka=FabrykaMiejskich()
else:
    raise NotImplementedError(f"nie ma fabryki dla typu {rodzaj}");
samochod=fabryka.produkuj_samochod()
samochod.jedz()
```

W zależności od ustawionej wartości w zmiennej "rodzaj" wykorzystuję obiekt właściwej dla danego rodzaju fabryki do stworzenia obiektu klasy dziedziczącej po klasie samochód. Z racji dziedziczenia wszystkich fabryk po klasie abstrakcyjnej "FabrykaSamochodow" mogę po obiekcie dowolnej fabryki

spodziewać się że posiada ona metodę "produkuj_samochod". Z racji dziedziczenia wszystkich rodzajów samochodów bo abstrakcyjnej klasie "Samochod" mogą w obiekcie dowolnego samochodu spodziewać się metody "jedz". Cały kod:

```
from abc import ABC, abstractmethod
class Samochod(ABC):
    @abstractmethod
    def jedz(self):
        pass

class SportowySamochod(Samochod):
    def jedz(self):
        print('sportowy wydech robi wroom!')

class Limuzyna(Samochod):
    def jedz(self):
        print('nawet nie usłyszysz dźwięku silnika...')

class SamochodMiejski(Samochod):
    def jedz(self):
        print('no niby jedzie...')

class FabrykaSamochodow(ABC):
    @abstractmethod
    def produkuj_samochod(self):
        pass

class FabrykaSportowych(FabrykaSamochodow):
    def produkuj_samochod(self):
        return SportowySamochod()

class FabrykaLimuzyn(FabrykaSamochodow):
    def produkuj_samochod(self):
        return Limuzyna()

class FabrykaMiejskich(FabrykaSamochodow):
    def produkuj_samochod(self):
        return SamochodMiejski()

rodzaj = 'miejski'
if rodzaj == 'sportowy':
    fabryka = FabrykaSportowych()
elif rodzaj == 'limuzyna':
    fabryka = FabrykaLimuzyn()
elif rodzaj == 'miejski':
    fabryka = FabrykaMiejskich()
else:
```

```
raise NotImplementedError(f"nie ma fabryki dla typu {rodzaj}");  
samochod=fabryka.produkuje_samochod()  
samochod.jedz()
```

Wzorzec "Budowniczy"

Stosujemy go gdy obiekt będzie konstruowany z wielu mniejszych, gdzie chcemy mieć kontrolę nad procesem tworzenia obiektu. Skupia się bardziej na sposobie konstrukcji niż na obiektach wchodzących w jego skład - jak to jest np. we wzorcu "Fabryka Abstrakcyjna". Tu będzie więcej w niedalekiej przyszłości....