

Spis treści

"Hello world" i pisanie na konsoli.....	5
Zmienne i typy danych.....	7
Zmienne.....	7
Typy danych.....	7
Typ tekstowy.....	8
Typy liczbowe.....	8
Instrukcje warunkowe.....	9
Jeden warunek.....	9
Else.....	10
Wiele warunków.....	10
Operatory logiczne w warunkach.....	11
Pętle.....	12
Pętla while.....	12
Pętla for.....	12
Zagnieżdżanie pętli.....	13
Instrukcja BREAK.....	14
Instrukcja CONTINUE.....	14
Łańcuchy znaków.....	16
Funkcje wbudowane.....	16
upper.....	16
lower.....	16
title.....	17
replace.....	17
len w kontekście ciągów tekstowych.....	17
count.....	18
strip.....	18
split i join - zamiana tekstu na listę i listy na tekst.....	19
Łańcuchy funkcji.....	19
Iterowanie po łańcuchach tekstowych.....	20
Mnożenie tekstu. Ale jak?.....	20
Wygodne sprawdzanie czy tekst zawiera frazę.....	21
Czy Python>Java?.....	21
Cięcia, cięcia - o cięciu łańcuchów tekstowych słów kilka.....	22
Listy.....	23
Tworzenie list.....	24
Pobieranie wartości z list.....	24
Iterowanie po listach.....	26
Sprawdzanie czy element znajduje się na liście.....	27
Modyfikowanie zawartości listy.....	27
Dodawanie nowych wartości i wstawianie w miejsce istniejących.....	27
Kasowanie elementów z listy.....	28
Funkcje wbudowane w listy.....	29
Sortowanie i odwracanie list.....	29
Inne ciekawe funkcje i możliwości.....	31

Krotki.....	33
Deklaracja i uzupełnianie krotek danymi.....	34
Pobieranie wartości z krotek.....	34
Słowniki.....	35
Tworzenie słowników.....	36
Pobieranie wartości ze słowników.....	36
Modyfikacja zawartości słowników.....	39
Zestawy.....	39
Tworzenie zestawów i konwersje z innych typów złożonych.....	39
Modyfikowanie zawartości zestawów.....	41
Wyjątki.....	42
Obsługa wyjątków.....	42
Funkcje.....	46
Deklarowanie funkcji.....	46
Parametry funkcji.....	47
Zwracanie wyników z funkcji.....	49
Wyrażenia Lambda.....	50
Funkcja jako argument.....	51
Funkcja w funkcji.....	51
Rekurencja.....	51
Dokumentowanie funkcji.....	52
Moduły.....	53
Definiowanie modułów.....	53
Dokumentowanie modułów i sprawdzanie dostępnych funkcji.....	55
Korzystanie z plików tekstowych.....	56
Czytanie z plików tekstowych.....	56
read().....	57
readlines().....	59
readline().....	60
Funkcja seek().....	60
Sprawdzanie ilości linii w pliku.....	61
Zapis w plikach tekstowych.....	62
Tryby otwarcia pliku.....	62
Wprowadzanie danych do pliku.....	63
Przetwarzanie JSON.....	64
Ładowanie danych JSON z pliku.....	65
Tworzenie i zapisywanie danych JSON do pliku.....	67
Przetwarzanie XML.....	69
Odczyt danych z pliku XML i sięganie do elementu po nazwie.....	70
Sięganie po podelementy.....	73
Sięganie do elementu po pozycji.....	73
Listy wartości w XML i odwoływanie się do "n-tego" wystąpienia tagu.....	74
Atrybuty.....	75
Użyteczne "sztuczki".....	77
Odczytywanie XML jako zwykły tekst.....	77
Sprawdzanie nazwy elementu.....	78
Modyfikowanie drzewa XML.....	79

Modyfikowanie zawartości elementu.....	79
Dodawanie i modyfikowanie atrybutów elementu.....	79
Tworzenie nowych elementów.....	80
Usuwanie elementów.....	81
Zapis drzewa XML do pliku.....	82
Dane zdalne - wykorzystanie usług sieciowych.....	83
Pobieranie danych za pomocą GET.....	83
Przesyłanie danych za pomocą POST.....	84
Wykorzystanie baz danych.....	85
Łączenie z serwerem bazy danych.....	85
Łączenie z serwerem PostgreSQL.....	85
Łączenie z serwerem Oracle.....	85
Pobieranie danych z użyciem SELECT.....	86
Wstawianie, zmiana i kasowanie danych, oraz operacje DDL.....	88
Wyrażenia regularne.....	89
Podstawowe wyszukiwanie.....	89
Typy znaków.....	90
Kwantyfikatory ilościowe.....	90
Wykorzystanie symboli i kwantyfikatorów do wyszukiwania elementów według wzorca.....	91
Testy jednostkowe - framework py.test.....	93
Podstawowe testy.....	93
Uruchamianie wybranych testów.....	97
Parametryzacja testów.....	102
Fikstury.....	106
Problematyka.....	106
Funkcje setup_module i teardown_module.....	109
Dekorator @pytest.fixture.....	110
Makiety (Mocks).....	114
Dane testowe.....	115
Sprawdzanie pokrycia kodu testami.....	116
Klasy w Pythonie.....	118
Deklaracja klasy i pola.....	118
Funkcje w klasie.....	122
Funkcje prywatne.....	124
Flask.....	126
Tworzenie projektu i mapowanie pierwszego adresu.....	126
Konfiguracja portu nasłuchu serwera i automatyczna implementacja zmian.....	128
Kod i szablony kodu HTML.....	129
Przekazywanie danych do widoku i jinja2.....	131
Odczyt parametrów z paska.....	137
Pobieranie i umieszczanie danych w sesji.....	143
Obsługa formularzy.....	144
Usługi sieciowe we Flask.....	146
Usługi sieciowe zwracające dane.....	146
Usługi sieciowe przyjmujące dane.....	148
Parsowanie stron internetowych z użyciem BeautifulSoup 4.4.....	151
Instalacja pakietu.....	151

Obiekt klasy BeautifulSoup.....	151
Wyszukiwanie elementów i funkcja find.....	153
Wyszukiwanie pierwszego wystąpienia.....	153
Wyszukiwanie po id elementu.....	154
Wyszukiwanie po klasie css.....	155
Wyszukiwanie po atrybutach elementu.....	156
Zagnieżdżanie.....	157
Sięganie do sekcji strony.....	158
Atrybuty elementów.....	158
.....	158
name i string.....	160
Operowanie na listach elementów i funkcja find_all.....	161
Filtrowanie elementów z funkcją find_all.....	162
contents.....	164

"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 konkatenacja 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. Pomiedzy 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 deklaruję 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 musimy 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  
y=2, x=9  
y=2, x=10  
y=3, x=1
```

```
y=3, x=2
y=3, x=3
.....
.....
```

Zagnieżdzać możemy również pętle while. Nic nie stoi też na przeszkodzie by pętlę for zagnieździć 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="Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogoch"  
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 niedomkniętego z prawej strony zbioru jak w przypadku funkcji range.

W 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 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 wypełniając 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 wydrukuje 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) przesuwając 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]" skasuje 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","Steven Hawking"]
pdk.sort()
pdk.reverse()
print(pdk)
```

I oczywiście wynik:

['Ziemniak', 'Steven Hawking', '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', 'Steven Hawking', '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). Można by 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

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 zappełnić 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', 'SZAJSUNG999'])

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 zapelniając 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 obsłużony 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 obsłużony 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 domyslne_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".

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)
```

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)
```

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 zaimpotowane 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)
```

Zobaczmy wtedy to co zadeklarowaliśmy jako dokumentację danej funkcji:

hello()

Ta funkcja wypisuje na ekranie tekst **hello world!**

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 pojawią 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
śczęćóź
```

Ostatnią linię 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

ścześnie

<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 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', 'ścżąęóź']

<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', 'ścżąęóź']
```

```
<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" przesuwaa 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', 'ściąćóó']
```

```
[]
```

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

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 pisanie 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 widzieć 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" dostać 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 elementu

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 elementu

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 opakującym, 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 dostać do konkretnej wartości w danych, to dalej postępuję tak jak z każdymi innymi 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.js  
on")  
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 przesyłanych informacji. Podobnie jak funkcja "get" przyjmuje również parametr "auth" którego tu jednak nie użyłem:

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

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 jakiegoś znaku mogącego być potraktowane jako znak specjalny 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". Linię:

```
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 oblał się 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:

W tym

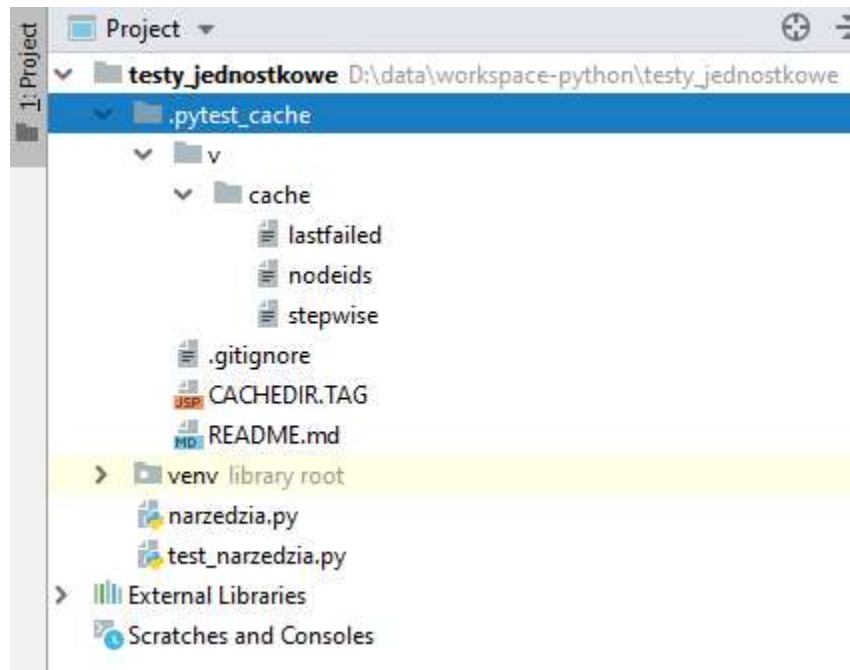
```
===== FAILURES =====
_____ test_sumuj _____

    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 =====
```

przypadku mamy tylko jedną funkcję testującą, jednak warto wiedzieć że w przypadku oblań 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 ....
```

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 .
```

Rzecz znacznie mniej oczywista to

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

możliwość uruchamiania testów które zawierają określony ciąg w nazwie:

Służy do
tego

```
(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>
```

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

```

"Python n

```

test_narzedzia.py::test_sumuj PASSED
test_narzedzia.py::test_dajCyfryLen PASSED

```

101/168

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:

Test

```
(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>
```

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
```

"Python na luzie" v.

105/168

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 przechwytuje 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:

Jak

```
(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 #####
```

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":

W

```
(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>
```

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:

Po

```
(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
=====
```

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 "dłfgdgdsgdsgdsg" :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 dorożenia 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

Klasy 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())
```

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 przeciwniejskie 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:

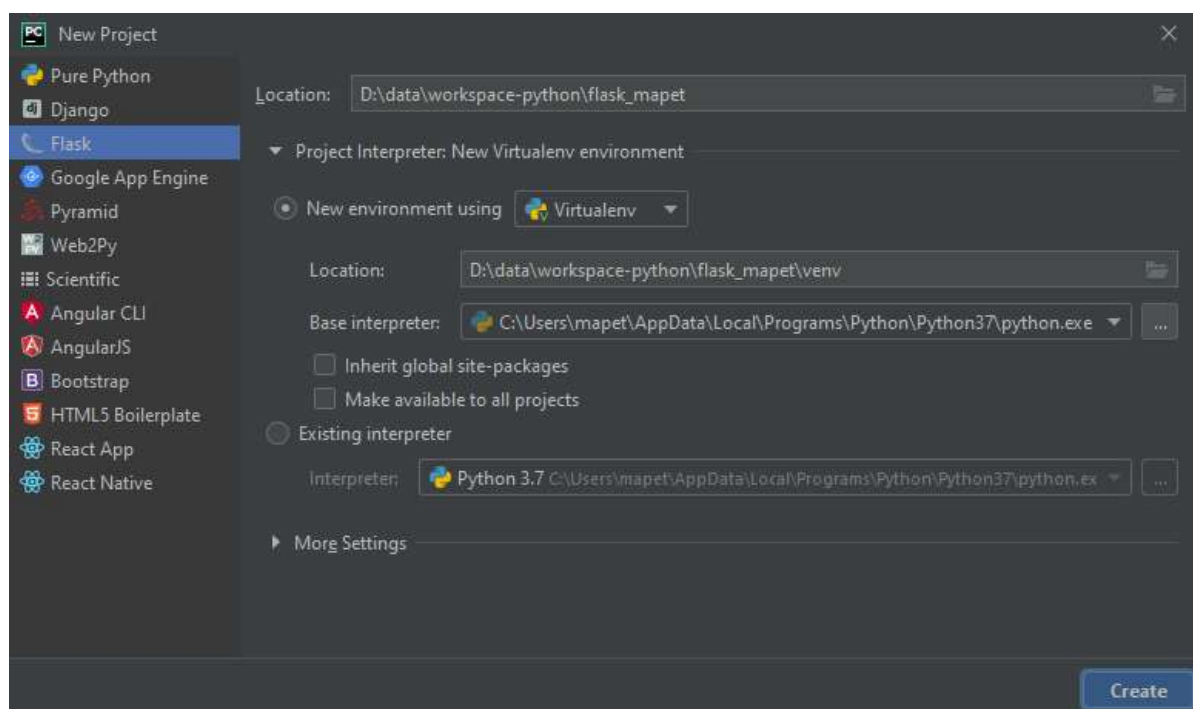
Traceback (most recent call last):
Wania Typowy Czy pindzisiont za godzine
File "D:/data/workspace-python/nauka/materialy/02.py", line 10, in <module>
print(pj.nazwisko,pj.imie,pj.__wyplata)
AttributeError: 'PracownikJanuszexu' object has no attribute '__wyplata'

Flask

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
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 szablony 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
```

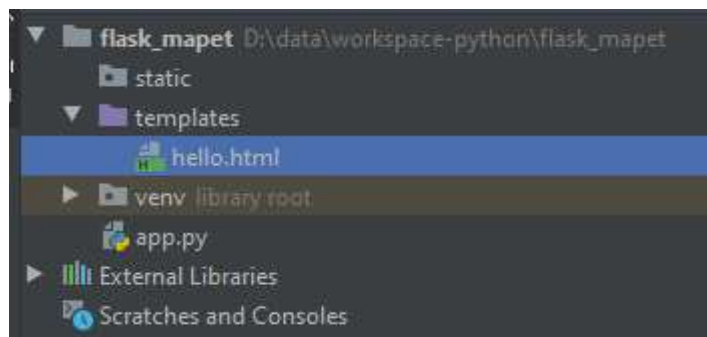
Efekt działania:



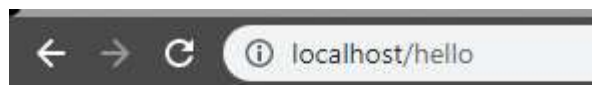
Na dłuższą metę, lub przy większej ilości kodu takie działanie będzie po prostu bardzo niewygodne. Wyobraź sobie 1000 linijkową stronę generowaną w ten sposób. Biorąc jeszcze pod uwagę że zwykle będą jakieś elementy wspólne na wielu podstronach, a podmieniana będzie tylko jakaś nieduża część widoku, to jest to całkowicie pozbawione sensu. Co więc możemy zrobić? Wykorzystać szablony HTML:

```
app.py x hello.html x
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
```

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

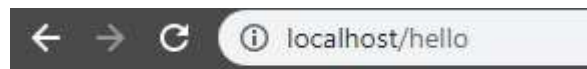
To są pojedyncze zmienne. Jak jednak przekazać listę, a następnie przeiterować po niej na poziomie widoku? W pierwszej kolejności modyfikuję funkcję by przekazać dane:

```
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....

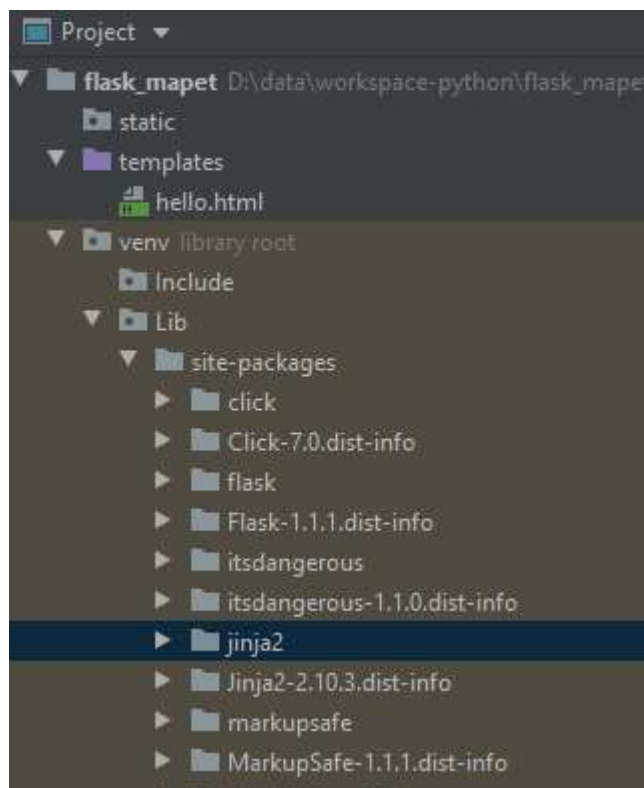
Imię: Andrzej

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
```

Odwołujemy się potem do tej listy w linii 7 naszego kodu HTML we fragmencie „for k in kraje”. Klamry i znacziki „%” 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:

← → ↺ ⓘ localhost/hello

To jest template....

Imię: Andrzej

Nazwisko: Klusiewicz

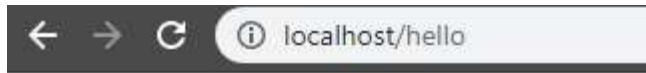
- Polska
- Niemcy
- Rosja
- Ukraina
- Czechy

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 takimi „” 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:

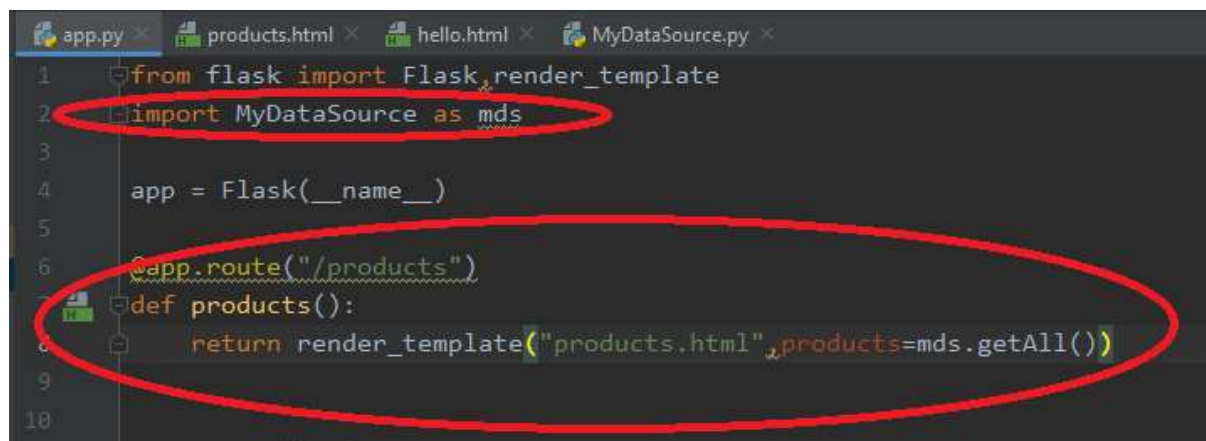
- **Polska**
- Niemcy
- Rosja
- Ukraina
- Czechy

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:

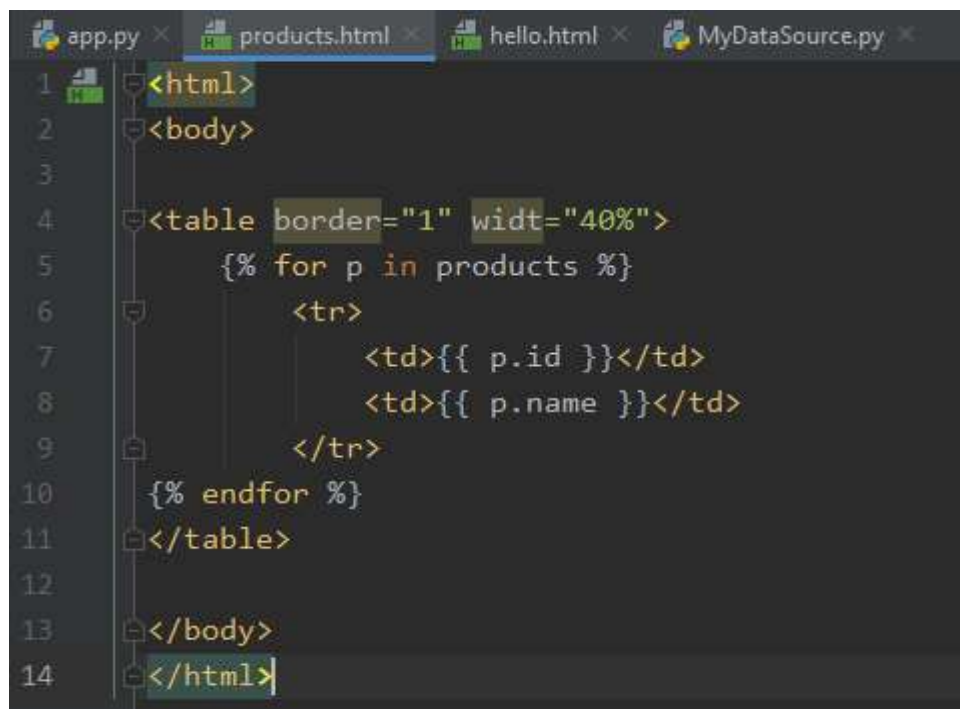
```
1 class Product:
2     def __init__(self, id, name, description):
3         self.id=id
4         self.name=name
5         self.description=description
6
7     dane=[
8         Product(1, 'Bulbulator', 'Robi bul bul'),
9         Product(2, 'Wihajster', 'Nie wiadomo co robi'),
10        Product(3, '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”.



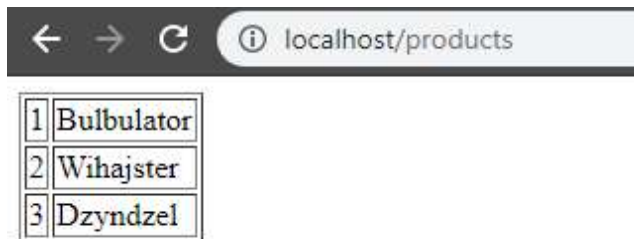
```
1 from flask import Flask, render_template
2 import MyDataSource as mds
3
4 app = Flask(__name__)
5
6 @app.route("/products")
7 def products():
8     return render_template("products.html", products=mds.getAll())
9
10
```

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>
```

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.

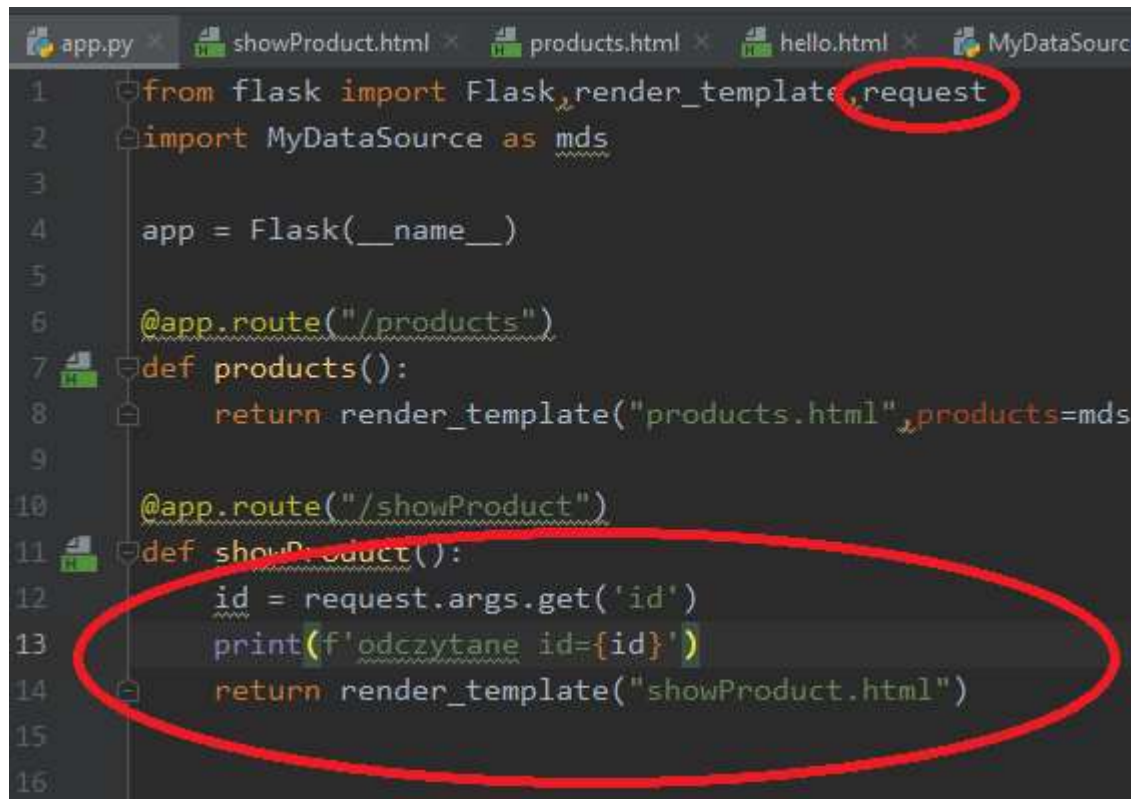
W kolejnym kroku dokonuję takiej przeróbki by wcześniej napisy, teraz stały się adresami pod które odsyłają linki:

```
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:

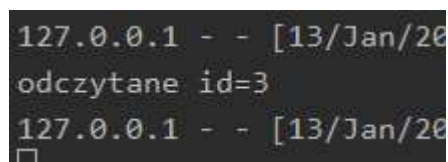


```
1 from flask import Flask, render_template, request
2 import MyDataSource as mds
3
4 app = Flask(__name__)
5
6 @app.route("/products")
7 def products():
8     return render_template("products.html", products=mds.products)
9
10 @app.route("/showProduct")
11 def showProduct():
12     id = request.args.get('id')
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
□
```

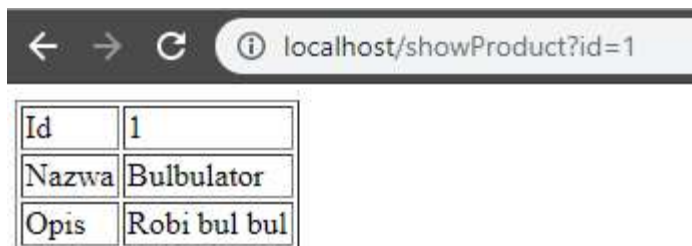
Pozostaje nam teraz odczytanie obiektu którego szczegóły chcemy przeglądać, przekazanie go do widoku i wyświetlenie. Korzystając z mojego pseudo DAO pobieram pojedynczy obiekt klasy Product (linia 13). Produkt ten przekazuję jak zawsze do widoku (linia 14):

```
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 data:

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 uzytkownika w sesji")
10    session['loggedUser'] = "ThePaniHalynaKsiegowa"
11    print("uzytkownik zalogowany to " + session['loggedUser'])
12    return ""
13
14 @app.route("/product")
```

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 uzytkownika w sesji
uzytkownik 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:

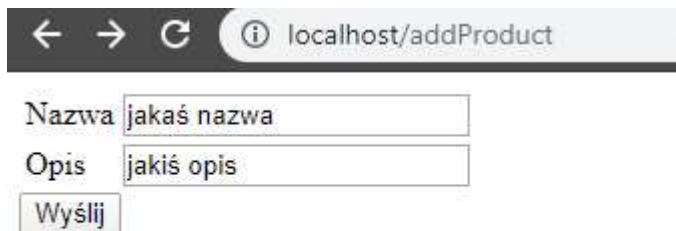
```
3 <form method="POST">
4     <table>
5         <tr>
6             <td>Nazwa</td>
7             <td><input type="text" name="name"/></td>
8         </tr>
9         <tr>
10            <td>Opis</td>
11            <td><input type="text" name="desc"/></td>
12        </tr>
13    </table>
14    <input type="submit" 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)
```

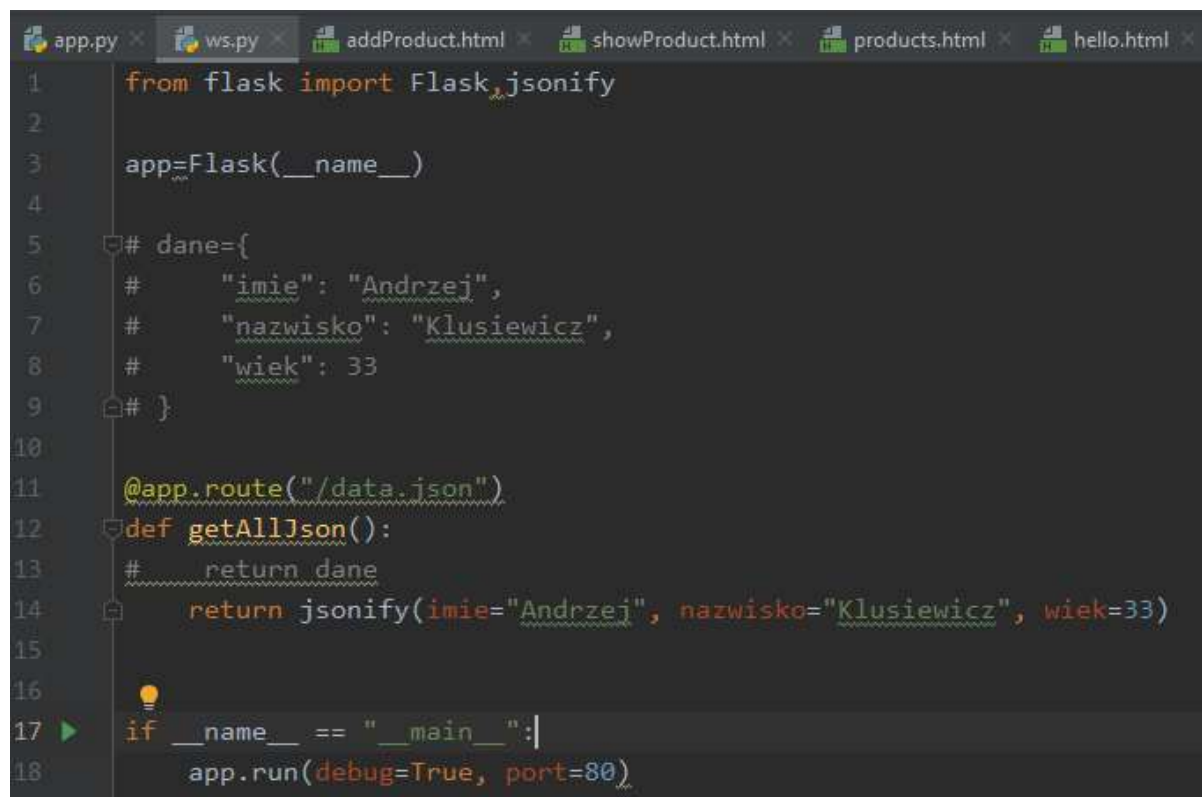
I efekt działania:



```
{
  "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.

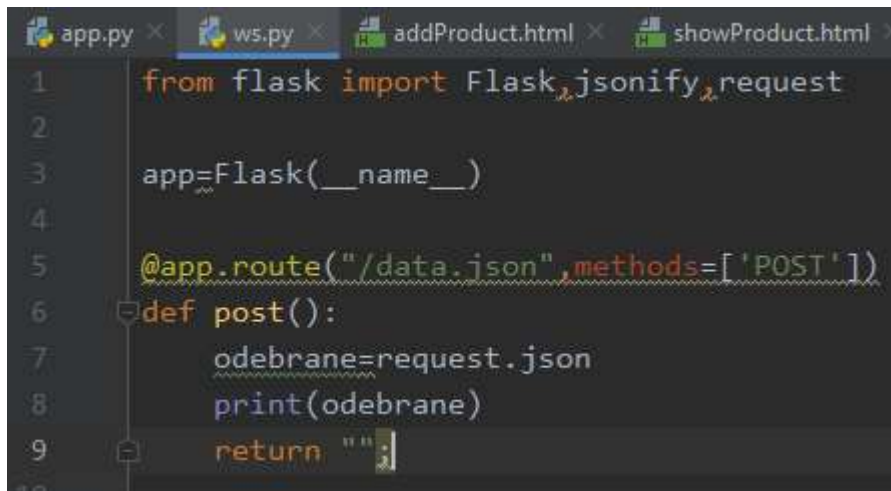
Zamiast deklarować słownik, mogę się też posłużyć funkcją jsonify serializującą dane. Muszę ją jednak uprzednio zaimportować. Korzystając z niej mogę osiągnąć taki sam efekt, ale bez deklaracji słownika:



```
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”:



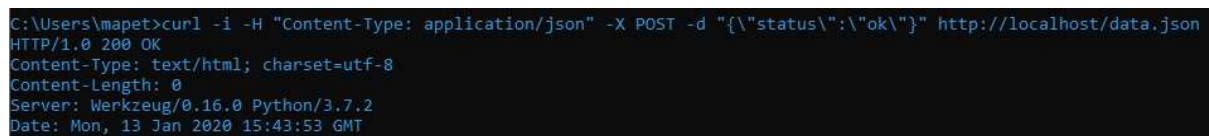
```
1 from flask import Flask, jsonify, request
2
3 app=Flask(__name__)
4
5 @app.route("/data.json", methods=['POST'])
6 def post():
7     odebrane=request.json
8     print(odebrane)
9     return ""
```

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. Przełącznik „-i” służy temu by w odpowiedzi dostać nagłówek 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  
}
```

Parsowanie stron internetowych z użyciem BeautifulSoup 4.4

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')
```

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>przyspieszacz 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 odtąd 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 jakiegokolwiek 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\beautifullsouptu
<div name="stopka">To jest stopka</div>

Process finished with exit code 0
```

Zagnieżdżanie

Każdy element zwracany przez funkcję `find`, ale również opisywane wcześniej uchwytów 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>
:|l
:~ 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\beautifulsoupstutorial\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\beautifulsoup
<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)
8
```

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 rozbija:

```
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\beautifulsouptutorial\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\beautifuls
↓ <td>Antarktyda</td>
ID:
⇅ Process finished with exit code 0
```

Teraz możemy odnieść się do „string” aby dobrać 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 wypłaj zawartość.