Python

Podstawy

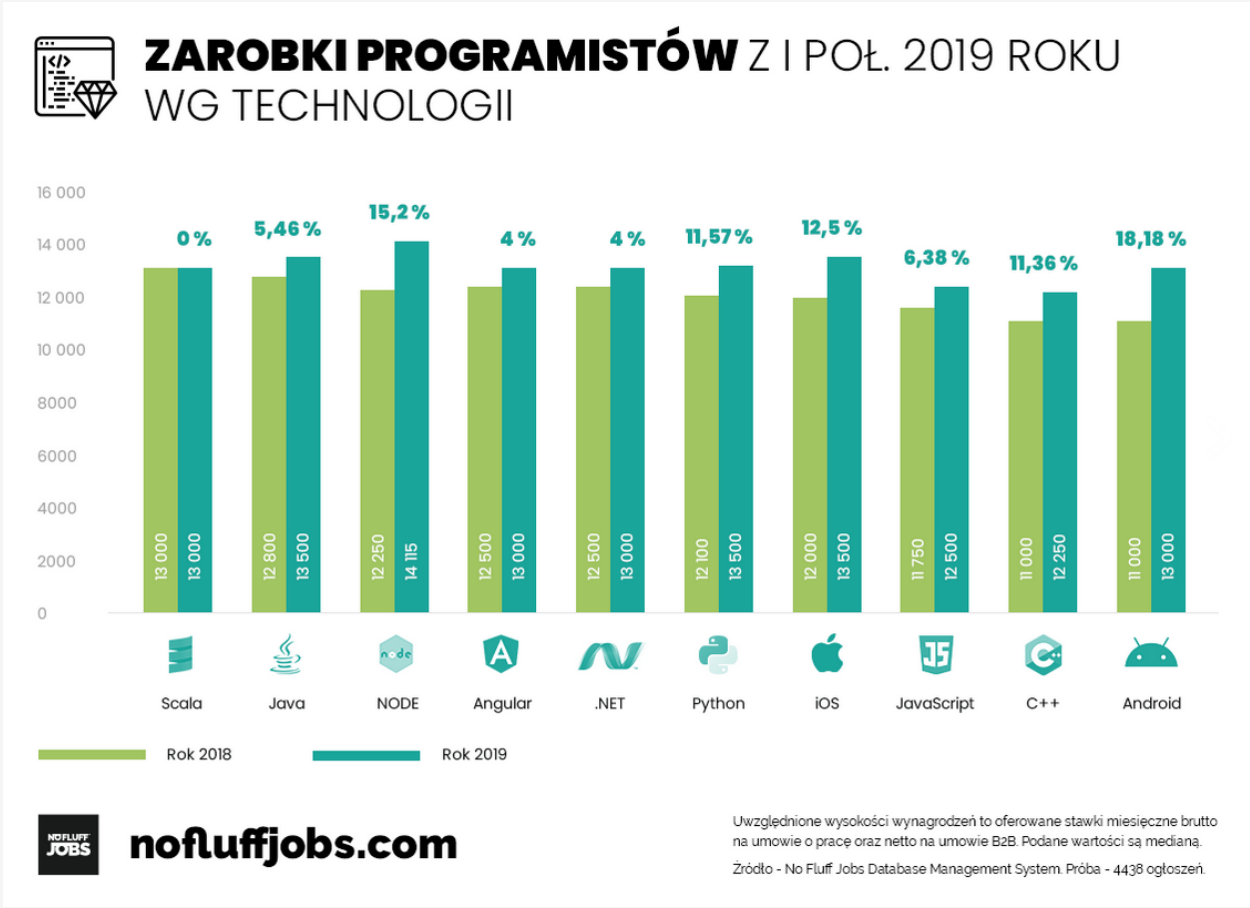
Łukasz Kordowski

Historia języka

- Napisany przez Guido van Rossum i wydany w 1990 (29 lat temu!) Amsterdam
- Najnowsza stabilna wersja to 3.7.4
- Wersja 2.7 przestanie być wspierana w 2020
- Nazwa nie pochodzi od węża a nazwy programu “Latający cyrk Monty Pythona.”
- Python rozwijany jest jako projekt Open Source zarządzany przez Python Software Foundation

Popularność języka

- Według Stack Overflow Developer Survey (najbardziej opiniotwórcze badania branży IT) Python jest najszybciej rosnącym w popularność językiem.
- Programiści Pythona to jedni z najlepiej zarabiających na rynku
- Ten kurs to dobry krok w Twojej karierze!



Najważniejsze zastosowania

- Technologie webowe
  - aplikacje webowe (Django, Flask)
  - REST API – interfejsy stron (Django, Flask, aiohttp)
  - Webscraping – indeksowanie stron (requests, BeatifulSoup)
- Obróbka i przetwarzanie danych (pandas, numpy, matplotlib)
  - Praca na plikach tekstowych i arkuszach danych (csv, xlsx)
  - Analiza danych i wizualizacja
- Uczenie maszynowe (Scikit-learn, TensorFlow)
  - Budowanie modeli typujących rozwiązania na podstawie wyuczonych wzorców
  - Sieci neuronowe
- Testy automatyczne (pytest, Selenium)
- Obsługa portów RaspberryPi

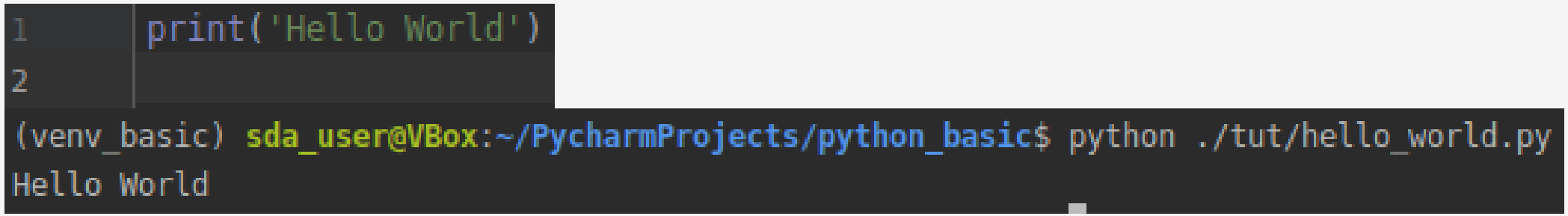
i wiele innych

Autor: Łukasz Kordowski

Podstawowe cechy

- **Wysokopoziomowy** – skupiasz się na rozwiązywanym problemie a nie na rejestrach procesora i alokowaniu pamięci
- **Interpretowalny** – w przeciwieństwie do języków kompilowanych, kod wykonywany jest przez dedykowany interpreter
- **Dynamicznie typowany** – typ zmiennej określany jest na podstawie przechowywanej wartości, nie ma potrzeby podawania typu zmiennej przy deklaracji
- Wspiera wiele praw (Paradygmatów) programowania:
  - Obiektowy (klasy, metody, dziedziczenie)
  - Funkcyjny (wyrażenia lambda)
  - Imperatywny (pętle, kolejność wykonania)
  - Refleksyjny (tworzenie obiektów w trakcie działania programu, dostęp do metod)

Pierwszy program



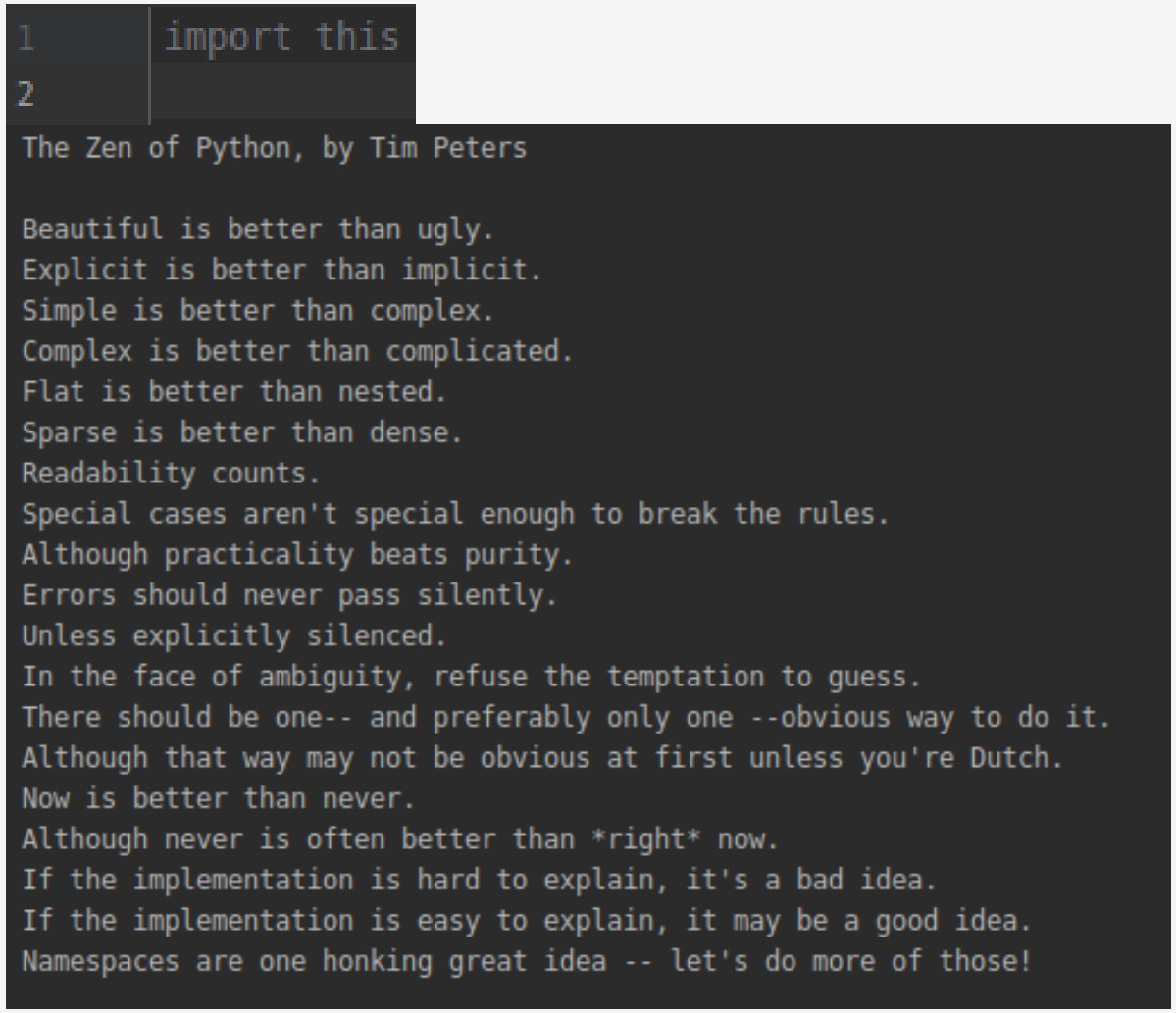
```
1       print('Hello World')
2
```

```
(venv_basic) sda_user@VBox:~/PycharmProjects/python_basic$ python ./tut/hello_world.py
Hello World
```

- Zawartość pliku to zaledwie funkcja wbudowana ***print*** z przekazanym ciągiem znaków
- Wykonanie skryptu następuje po wywołaniu w terminalu polecenia ***python*** z podaną ścieżką do skryptu

Autor: Łukasz Kordowski

Filozofia języka



```
1   import this
2
```

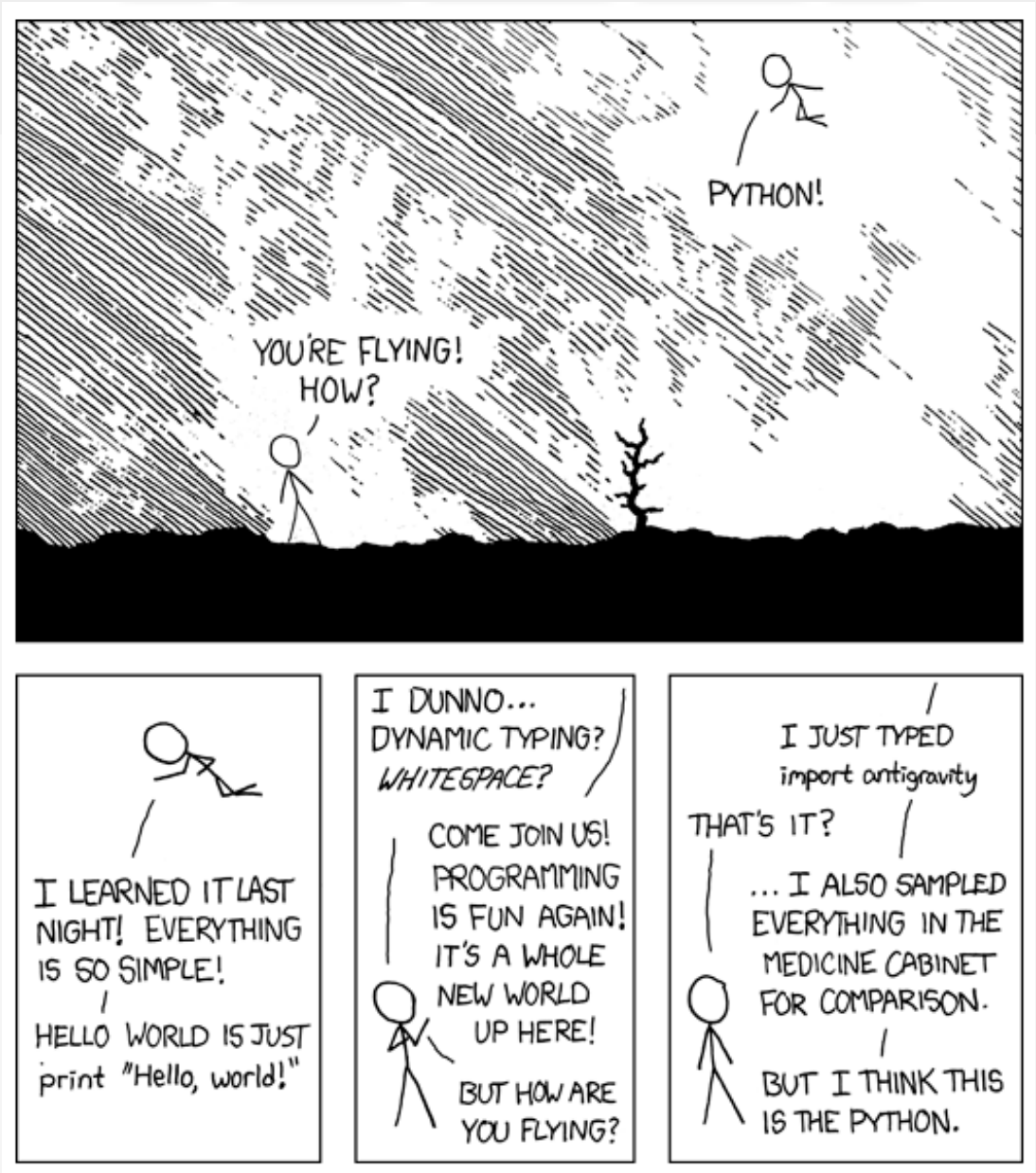
```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

- Filozofia języka Python opisana lirycznie

Autor: Łukasz Kordowski

Ciekawostka języka

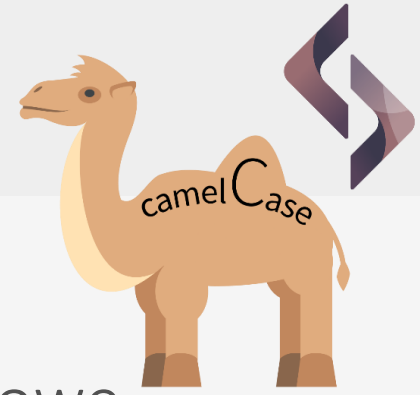
```
1  import antigravity
2
```



Działanie niektórych wbudowanych modułów może być dość zaskakujące

Autor: Łukasz Kordowski

Składnia języka



- język C podobny
- formatowanie poziomów programu za pomocą wcięć (unikatowe rozwiązanie), brak znacznika końca linii
- zmienne nie potrzebują deklaracji typu
- snake_case i camelCase

Python

```python
def silnia(x):
    if x == 0:
        return 1
    else:
        return x * silnia(x-1)

```

C

```c
int silnia(int x) {
    if (x == 0) return 1;
    else return x * silnia(x-1);
}

```

Autor: Łukasz Kordowski

Więcej o składni języka

- Zbiór dobrych praktyk formatowania kodu definiuje standard PEP8

Tak

```
x = 1
y = 2
long_variable = 3
```

Nie

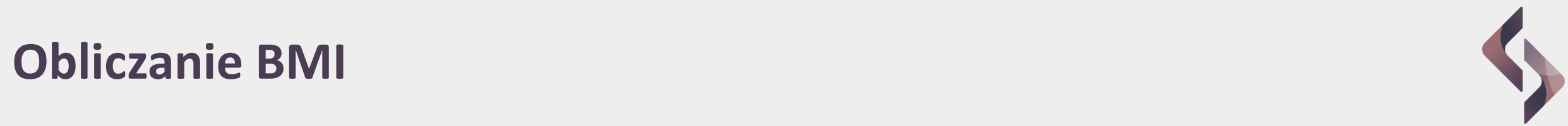
```
x =                 1
y =                 2
long_variable =     3
```

Cegiełki programowania

- Zmienne
- Pętle
- Funkcje
- Wyrażenia warunkowe
- Operatory
- Struktury danych
- Biblioteka standardowa/wbudowana

Autor: Łukasz Kordowski

Obliczanie BMI



```python
def calculate_bmi(weight, height):
    bmi = weight / height ** 2
    if bmi < 18.5:
        result = 'underweight'
    elif bmi > 25:
        result = 'overweight'
    else:
        result = 'normal'
    return result


if __name__ == '__main__':
    user_weight = float(input('Weight in [kg]: '))
    user_height = float(input('Height in [m]: '))
    user_result = calculate_bmi(user_weight, user_height)
    print(f'Your BMI says {user_result}')

```

p06_bmi.py

```
Weight in [kg]: 80
Height in [m]: 1.98
Your BMI says normal
```

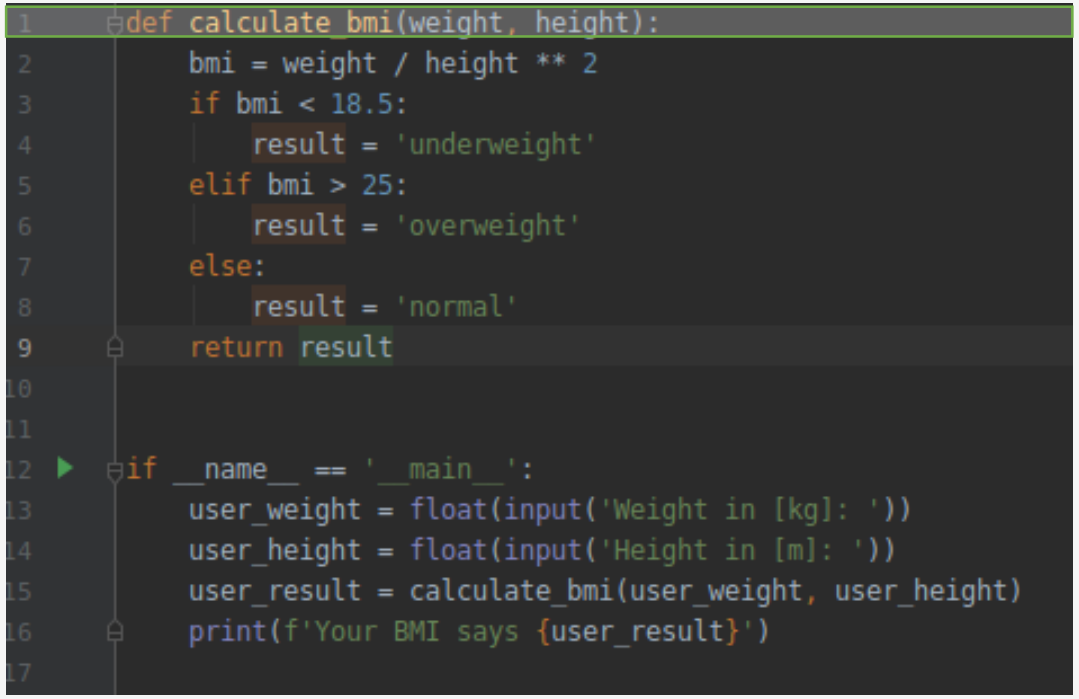
Wynik

```
Weight in [kg]: 80
Height in [m]: 1.98
Your BMI: 20.41 says normal
```

Wynik z bmi, p06a_bmi.py

Autor: Łukasz Kordowski

Obliczanie BMI - objaśnienie

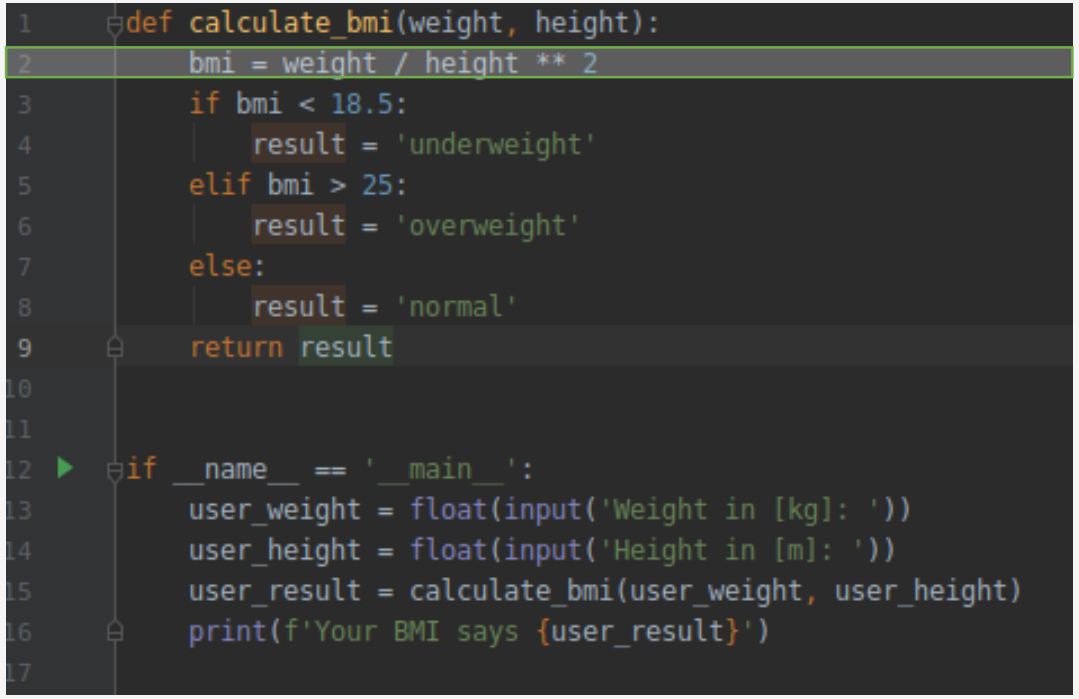


```python
def calculate_bmi(weight, height):
    bmi = weight / height ** 2
    if bmi < 18.5:
        result = 'underweight'
    elif bmi > 25:
        result = 'overweight'
    else:
        result = 'normal'
    return result


if __name__ == '__main__':
    user_weight = float(input('Weight in [kg]: '))
    user_height = float(input('Height in [m]: '))
    user_result = calculate_bmi(user_weight, user_height)
    print(f'Your BMI says {user_result}')

```

- Definicja funkcji rozpoczęta jest znacznikiem ***def***
- Funkcje pomagają ograniczyć liczbę linii kodu poprzez swoje wywołanie
- Nazwa funkcji posługuje się konwencją snake_case
- W nawiasach okrągłych podajemy argumenty, czyli wartości potrzebne do wykonania funkcji, rozdzielone przecinkiem
- Linię definicji funkcji kończy znak ***:***
- Ciało funkcji znajduje się za wcięciem

Autor: Łukasz Kordowski

Obliczanie BMI - objaśnienie

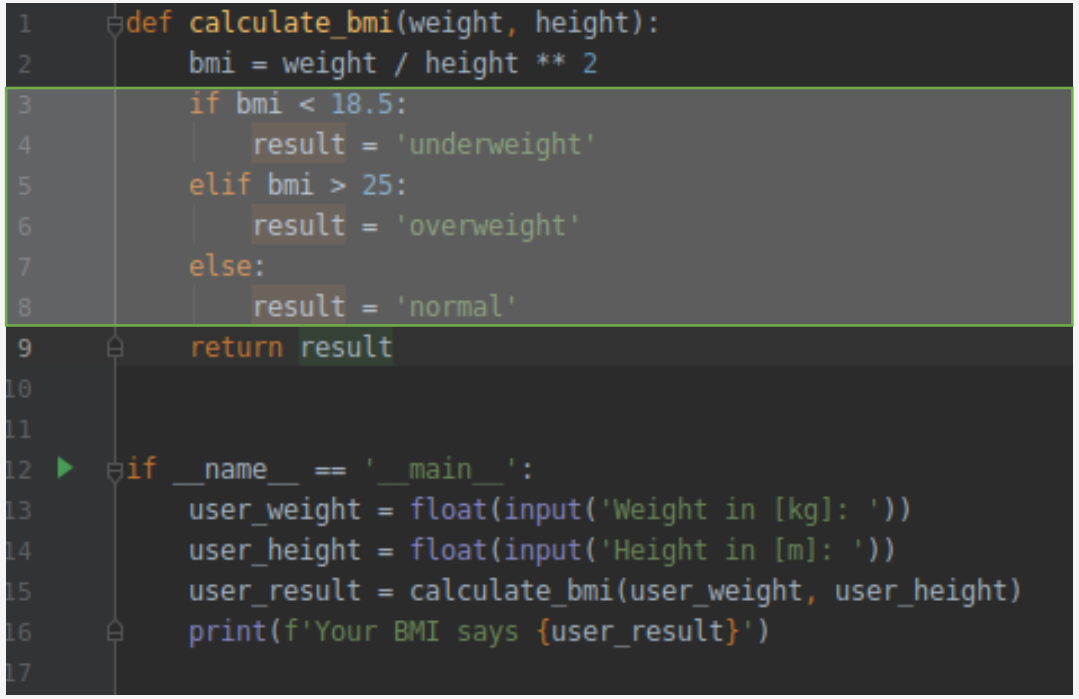


```python
def calculate_bmi(weight, height):
    bmi = weight / height ** 2
    if bmi < 18.5:
        result = 'underweight'
    elif bmi > 25:
        result = 'overweight'
    else:
        result = 'normal'
    return result


if __name__ == '__main__':
    user_weight = float(input('Weight in [kg]: '))
    user_height = float(input('Height in [m]: '))
    user_result = calculate_bmi(user_weight, user_height)
    print(f'Your BMI says {user_result}')
```

- W ciele funkcji wykonujemy potrzebne nam obliczenia
- W tym celu przy pomocy operatora przypisania **=**, zapisujemy wynik obliczeń do zmiennej ***bmi***
- BMI obliczamy jako wynik dzielenia naszych argumentów, waga przez drugą potęgę wzrostu

Autor: Łukasz Kordowski

Obliczanie BMI - objaśnienie

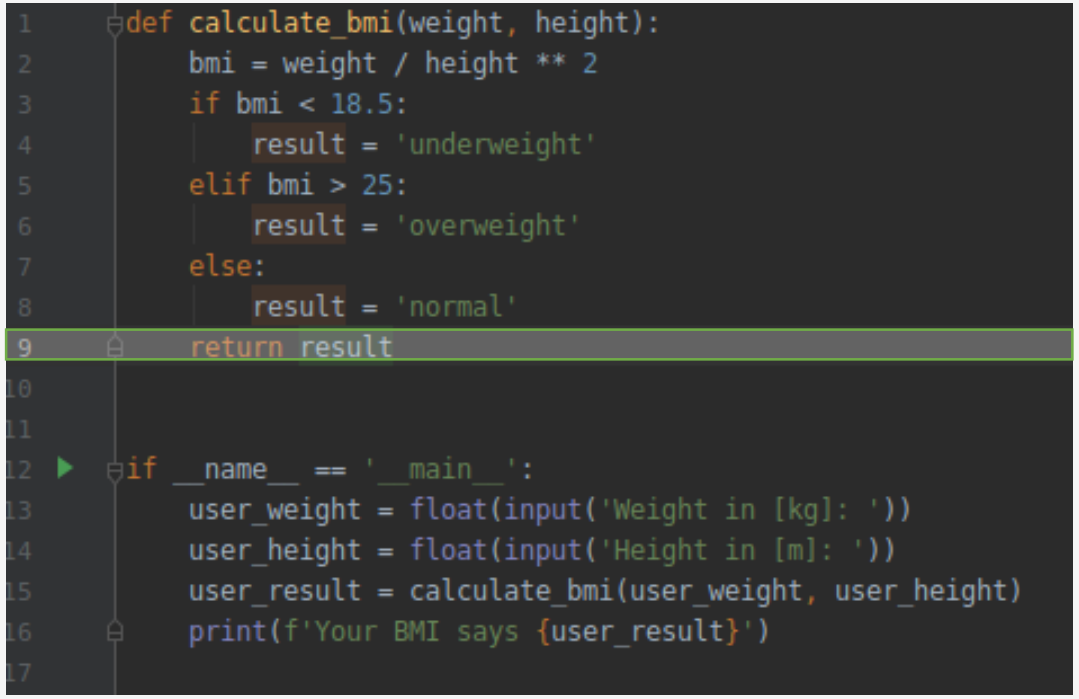


```python
def calculate_bmi(weight, height):
    bmi = weight / height ** 2
    if bmi < 18.5:
        result = 'underweight'
    elif bmi > 25:
        result = 'overweight'
    else:
        result = 'normal'
    return result


if __name__ == '__main__':
    user_weight = float(input('Weight in [kg]: '))
    user_height = float(input('Height in [m]: '))
    user_result = calculate_bmi(user_weight, user_height)
    print(f'Your BMI says {user_result}')
```

- Blok instrukcji warunkowej rozpoczynamy znacznikiem *if*
- Po nim następuje wyrażenie logiczne z operatorem porównania, którego sprawdzenie nas interesuje
- Warunek kończymy dwukropkiem. Wcięcie zawiera kod wykonywany w przypadku gdy sprawdzany warunek jest prawdziwy
- Pozostałe znaczniki ***elif*** i ***else*** są sprawdzane w przypadku gdy poprzedni warunek nie był spełniony

Autor: Łukasz Kordowski

Obliczanie BMI - objaśnienie

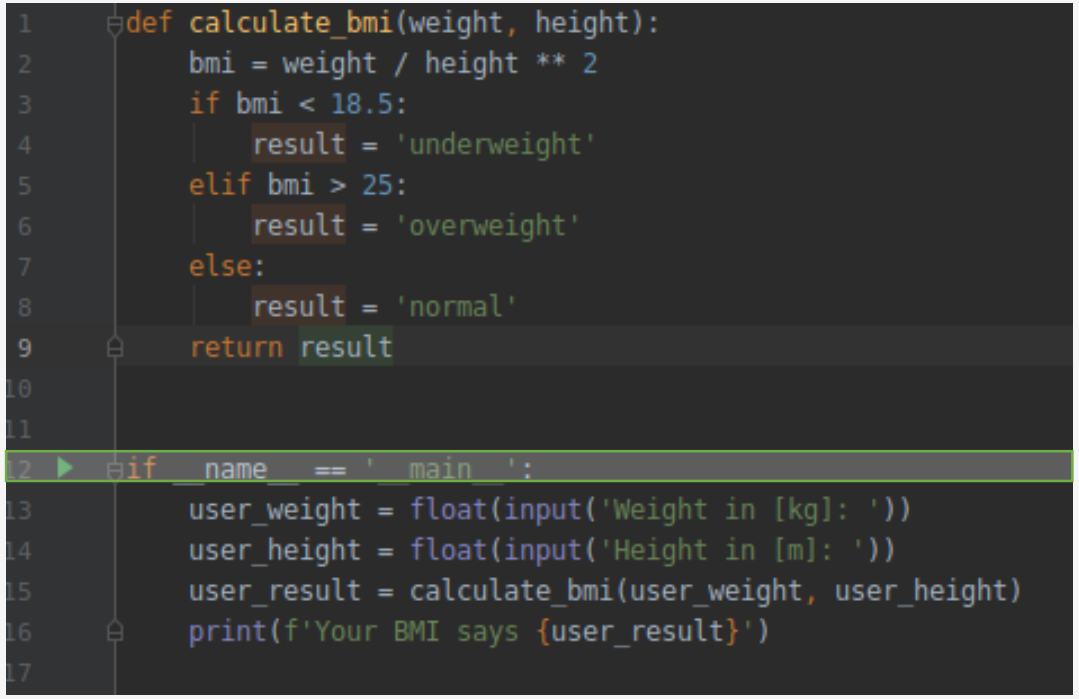


```python
def calculate_bmi(weight, height):
    bmi = weight / height ** 2
    if bmi < 18.5:
        result = 'underweight'
    elif bmi > 25:
        result = 'overweight'
    else:
        result = 'normal'
    return result


if __name__ == '__main__':
    user_weight = float(input('Weight in [kg]: '))
    user_height = float(input('Height in [m]: '))
    user_result = calculate_bmi(user_weight, user_height)
    print(f'Your BMI says {user_result}')

```

- Słowo kluczowe ***return*** zwraca wartość wyznaczoną w trakcie wykonywania funkcji i kończy jej działanie
- Nie wszystkie funkcje muszą zwracać wartość

Autor: Łukasz Kordowski

Obliczanie BMI - objaśnienie

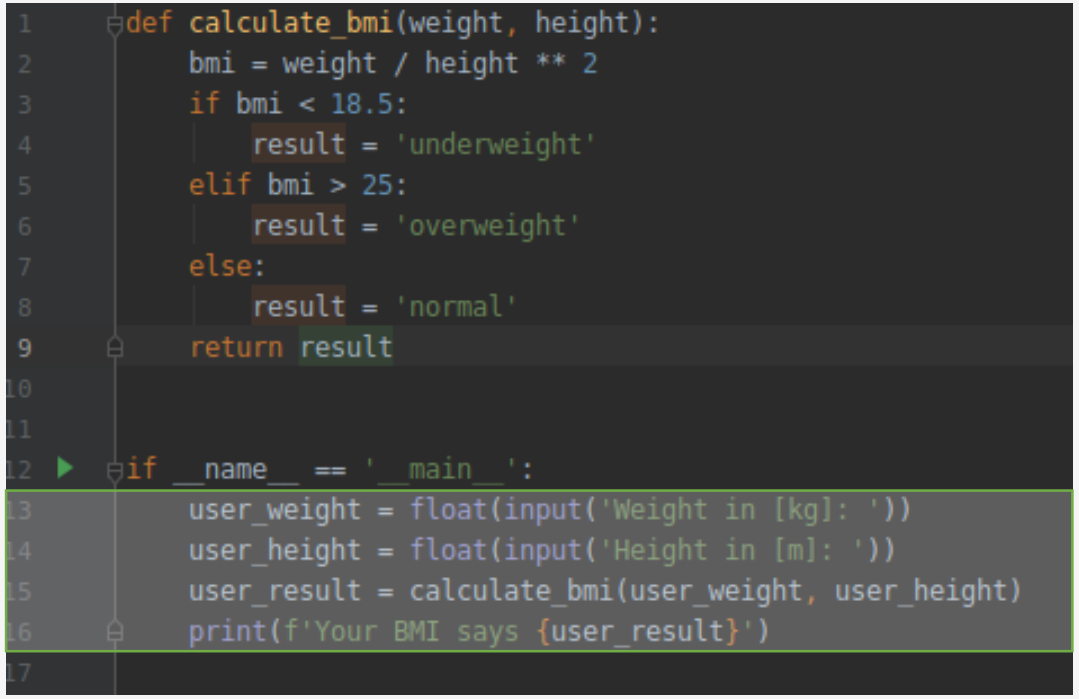


```python
def calculate_bmi(weight, height):
    bmi = weight / height ** 2
    if bmi < 18.5:
        result = 'underweight'
    elif bmi > 25:
        result = 'overweight'
    else:
        result = 'normal'
    return result


if __name__ == '__main__':
    user_weight = float(input('Weight in [kg]: '))
    user_height = float(input('Height in [m]: '))
    user_result = calculate_bmi(user_weight, user_height)
    print(f'Your BMI says {user_result}')
```

- Punkt wejściowy skryptu znajduje się w instrukcji warunkowej
- Kod znajdujący się za wcięciem będzie wykonany wyłącznie gdy plik go zawierający zostanie bezpośrednio wykonany
- Jest to zabezpieczenie przed wykonaniem kodu podczas importowania pliku w innym module/pliku
- Do sprawdzenie warunku używamy operatora porównania **==**

Autor: Łukasz Kordowski

Obliczanie BMI - objaśnienie

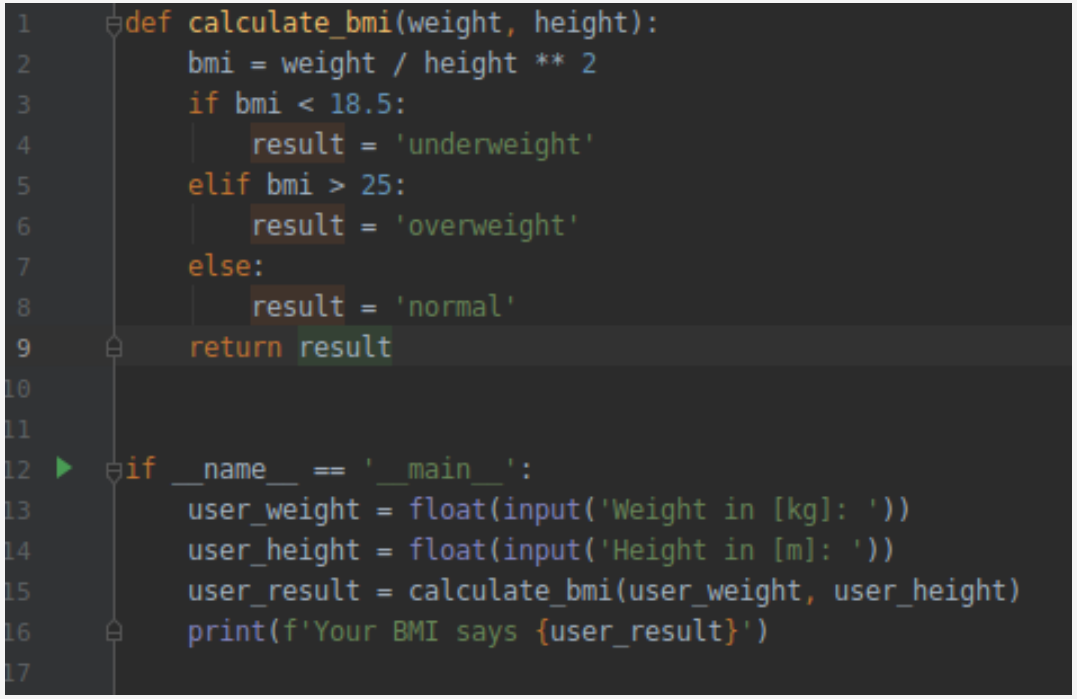


```python
def calculate_bmi(weight, height):
    bmi = weight / height ** 2
    if bmi < 18.5:
        result = 'underweight'
    elif bmi > 25:
        result = 'overweight'
    else:
        result = 'normal'
    return result


if __name__ == '__main__':
    user_weight = float(input('Weight in [kg]: '))
    user_height = float(input('Height in [m]: '))
    user_result = calculate_bmi(user_weight, user_height)
    print(f'Your BMI says {user_result}')
```

- Przypisujemy do zmiennych wartości wprowadzane przez użytkownika
- Na ich podstawie obliczamy żądaną wartość BMI
- Wypisujemy wynik dodając wynik do ciągu znaków przy pomocy ***f-string***

Autor: Łukasz Kordowski

Obliczanie BMI – składnia języka

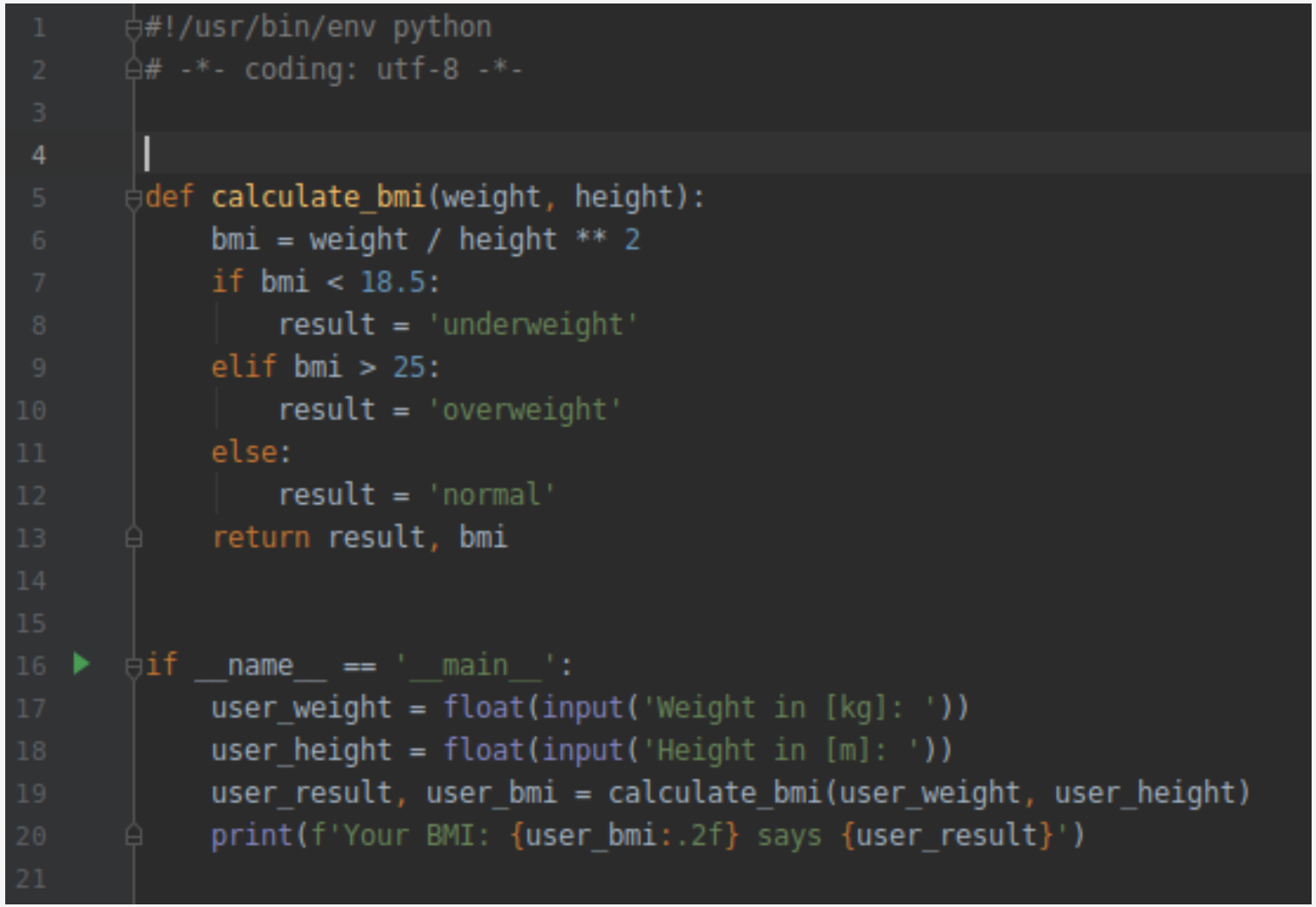


```python
def calculate_bmi(weight, height):
    bmi = weight / height ** 2
    if bmi < 18.5:
        result = 'underweight'
    elif bmi > 25:
        result = 'overweight'
    else:
        result = 'normal'
    return result


if __name__ == '__main__':
    user_weight = float(input('Weight in [kg]: '))
    user_height = float(input('Height in [m]: '))
    user_result = calculate_bmi(user_weight, user_height)
    print(f'Your BMI says {user_result}')

```

- Wielkość liter nazw ma znaczenie w pythonie, zmienna ***Result*** i ***result*** to dwie różne zmienne
- Należy unikać krótkich, małoznaczących nazw zmiennych i funkcji
- Podwójny podkreślnik to ***dunder*** (**d**ouble **under**score)
- Funkcje, które zawierają dunder nazywamy specjalnymi (***__name__***)

Autor: Łukasz Kordowski

Obliczanie BMI - shebang



```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-


def calculate_bmi(weight, height):
    bmi = weight / height ** 2
    if bmi < 18.5:
        result = 'underweight'
    elif bmi > 25:
        result = 'overweight'
    else:
        result = 'normal'
    return result, bmi


if __name__ == '__main__':
    user_weight = float(input('Weight in [kg]: '))
    user_height = float(input('Height in [m]: '))
    user_result, user_bmi = calculate_bmi(user_weight, user_height)
    print(f'Your BMI: {user_bmi:.2f} says {user_result}')

```

Autor: Łukasz Kordowski

Formatowanie tekstu

- Wyróżniamy kilka sposobów formatowania tekstu:
  - % - niezalecany
  - .format
  - f-string

```
>>> '%s %s' % ('one', 'two')
'one two'
>>> 'Value: {0}, rest: {1:.2f}'.format(1, 2)
'Value: 1, rest: 2.00'
>>> a = 1
>>> f'Value: {a}'
'Value: 1'
>>> b='a'
>>> f'Letter: {b.capitalize()}'
'Letter: A'
```

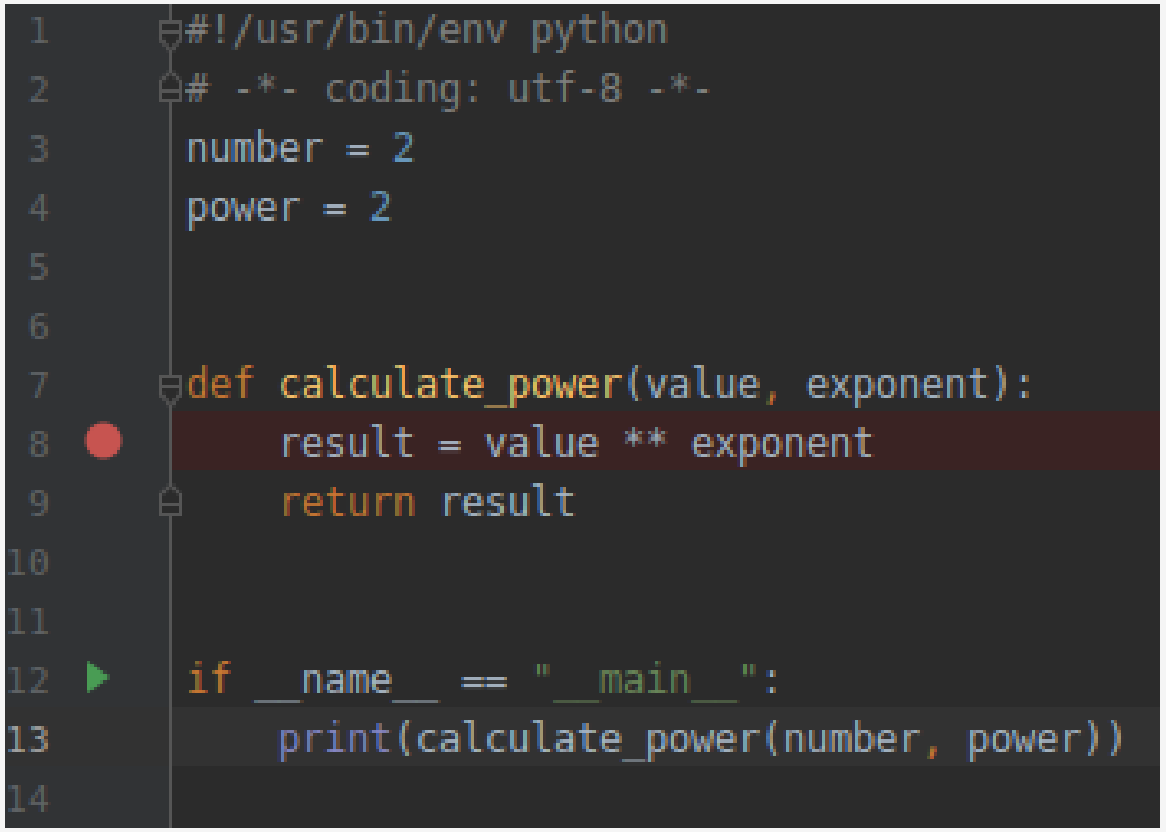
Autor: Łukasz Kordowski

Ciekawostka

- Jak to możliwe?

```
>>> 10
10
>>> _ + 1
11
```

Cykl życia zmiennej



```
1   #!/usr/bin/env python
2   # -*- coding: utf-8 -*-
3   number = 2
4   power = 2
5
6
7   def calculate_power(value, exponent):
8       result = value ** exponent
9       return result
10
11
12  if __name__ == "__main__":
13      print(calculate_power(number, power))
14
```

- Zmienne w zależności od miejsca deklaracji posiadają odpowiedni zakres występowania
- Zmienna ***number*** i ***power*** to zmienne globalne
- Zmienna ***result*** to zmienna lokalna
- PyCharm posiada rozbudowany tryb debuggera do pracy krokowej

Autor: Łukasz Kordowski

Podstawowe typy zmiennych

- **Integer** (liczby całkowite)
- **Float** (liczby zmiennoprzecinkowe)
- **String** (teksty – łańcuchy znaków)
- **Boolean** (wartości logiczne - True/False, prawda/fałsz)
- **None** (specjalny typ oznaczający brak wartości)
- **List** (uporządkowane listy)
- **Tuple** (krotki)
- **Set** (zbiory)
- **Dict** (słowniki)

```
1    foo = 1
2    foo = 1.2
3    foo = 'bar'
4    foo = False
5    foo = None
6    foo = [1, 2, 3]
7    foo = (1, 2, 3)
8    foo = {1, 2, 3}
9    foo = {'spam': 1, 'eggs': 2}
```

Autor: Łukasz Kordowski

Foo bar

- W programowaniu zmiennym trzeba nadać jakieś nazwy i często zastanawiamy się jaka nazwa jest sensowna
- Czasami zdarza się, że chcemy wytłumaczyć jakąś koncepcję, podać jakiś przykład w którym nazwa zmiennej jest nieistotna, ważna jest idea, którą chcemy wytłumaczyć
- Najbardziej popularnymi zwyczajowymi nazwami są **foo**, **bar** oraz **baz**
- **spam** i **eggs** są to nazwy bezpośrednio zaczerpnięte z programu Latający Cyrk Monty Pythona

Autor: Łukasz Kordowski

Funkcje wbudowane

- Funkcje które dostępne są domyślnie, nie ma potrzeby ich importowania
- Matematyczne:
  - abs() – zwraca wartość absolutną liczby
  - max() – zwraca wartość maksymalną zbioru
- Konwersji typów:
  - int() – zwraca wartość całkowitą liczby
  - hex() – zwraca wartość heksadecymalną liczby
- Wejścia / wyjścia:
  - input() – odczytuje wprowadzane dane z konsoli
  - print() – wypisuje łańcuch znaków
- Referencji:
  - dir() – listuje właściwości obiektu
  - type() – zwraca typ obiektu

```
>>> abs(-4)
4
>>> max([1, 2, 3])
3
>>> int(1.2)
1
>>> hex(10)
'0xa'
```

Autor: Łukasz Kordowski

Obiekty

- Wszystko w Pythonie jest obiektem
- Możemy korzystać z wbudowanych w obiekty metody

```
>>> isinstance('Text', object)
True
>>> isinstance(1, object)
True
>>> isinstance(None, object)
True
>>> isinstance(isinstance, object)
True
>>> help(object)
Help on class object in module builtins:

class object
 |  The most base type
```

```
1   foo = 1
2   foo = 1.2
3   foo = 'bar'.
4   foo = Fa
5   foo = No
6   foo = [1
7   foo = (1
8   foo = {1
9   foo = {'
10
```

```
capitalize(self)                          str
casefold(self)                            str
center(self, width, fillchar)             str
count(self, x, __start, __end)            str
encode(self, encoding, errors)            str
endswith(self, suffix, start, end)        str
expandtabs(self, tabsize)                 str
find(self, sub, __start, __end)           str
format(self, args, kwargs)                str
format_map(self, map)                     str
index(self, sub, __start, __end)          str
Press Enter to insert, Tab to replace
```

Autor: Łukasz Kordowski

Operatory

- Pozwalają manipulować wartościami zmiennych
- Python wspiera następujące typy operatorów:
  - Arytmetyczne
  - Porównania
  - Przypisania
  - Logiczne
  - Bitowe
  - Przynależności do zbioru
  - Stanu obiektu
- Kolejność wykonywania operatorów

Autor: Łukasz Kordowski

Operatory arytmetyczne

Operator	Opis
+	Dodawanie
-	Odejmowanie
*	Mnożenie
/	Dzielenie
%	Reszta z dzielenia
**	Potęgowanie
//	Dzielenie całkowite

```
>>> 2 + 3
5
>>> 5 - 2
3
>>> 3 * 4
12
>>> 2 / 3
0.6666666666666666
>>> 5 % 2
1
>>> 2 ** 3
8
>>> 2 // 3
0
```

Autor: Łukasz Kordowski

Operatory porównania

Operator	Opis
==	Równość
!=	Różne
>	Większy niż
<	Mniejszy niż
>=	Większy lub równy
<=	Mniejszy lub równy

```
>>> 2 == 2
True
>>> 2 == 3
False
>>> 2 != 3
True
>>> 2 > 3
False
>>> 2 < 3
True
```

Autor: Łukasz Kordowski

Operatory przypisania

Operator	Opis
=	Przypisanie
+=	Dodanie wartości i przypisanie
-=	Odjęcie wartości i przypisanie
*=	Pomnożenie wartości i przypisanie
/=	Podzielenie wartości i przypisanie
%=	Operacja modulo i przypisanie
**=	Potęgowanie wartości i przypisanie
//=	Dzielenie całkowite i przypisanie

```
>>> 2 += 3
  File "<input>", line 1
SyntaxError: can't assign to literal
>>> a = 2
>>> a += 2
>>> a
4
>>> a **= 2
>>> a
16
```

Autor: Łukasz Kordowski

Operatory bitowe

Operator	Opis
&	Koniunkcja bitowa
|	Alternatywa bitowa
^	Alternatywa bitowa rozłączna
~	Negacja bitowa
<<	Przesunięcie bitowe w lewo
>>	Przesunięcie bitowe w prawo

```
>>> a = 0b1010
>>> b = 0b0110
>>> bin(a & b)
'0b10'
>>> a & b
2
>>> bin(a | b)
'0b1110'
>>> bin(a ^ b)
'0b1100'
>>> bin(~b)
'-0b111'
>>> 0b0001 << 1
2
>>> bin(0b0001 << 1)
'0b10'
```

Autor: Łukasz Kordowski

Operatory logiczne

Operator	Opis
and	Koniunkcja
or	Alternatywa
not	Negacja

```
>>> True and True
True
>>> True and False
False
>>> True or False
True
>>> False or False
False
>>> not True
False
```

Operatory przynależności

Operator	Opis
in	Element zawiera się w sekwencji
not in	Element nie zawiera się w sekwencji

```
>>> 1 in [1, 2, 3]
True
>>> 4 in [1, 2, 3]
False
>>> 4 not in [1, 2, 3]
True
```

Autor: Łukasz Kordowski

Operatory stanu obiektu

Operator	Opis
is	Zmienne wskazują na ten sam obiekt (id)
is not	Zmienne nie wskazują na ten sam obiekt

```
>>> a = 1
>>> b = a
>>> c = 2
>>> a is b
True
>>> a is c
False
>>> b = 4
>>> a is b
False
>>> id(a)
10968800
```

Autor: Łukasz Kordowski

Operatory

- Możliwe jest także składanie operatorów w celu sprawdzenia bardziej zaawansowanych warunków

```
>>> (1.22 > 3) or (5 % 2 > 0) and 'e' in 'Text'
True
```

Obliczanie silni

- Iloczyn wszystkich liczb naturalnych dodatnich nie większych od podanej
- Oznaczane jako **n!**
- Oblicz silnię dowolnej liczby naturalnej

n!

Autor: Łukasz Kordowski

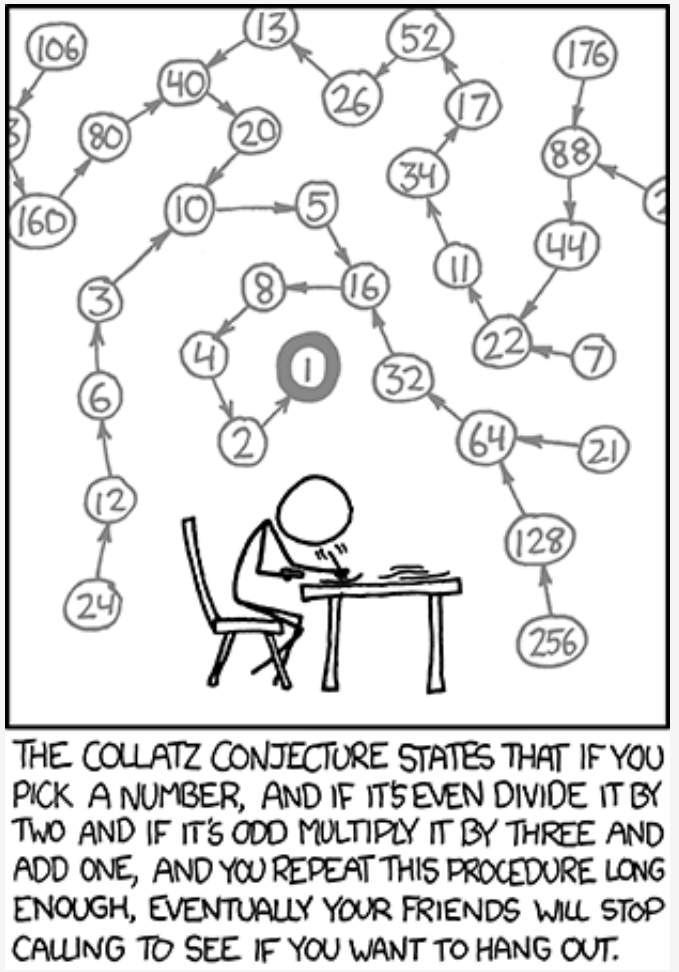
Obliczanie silni

- Pojawiła się tutaj funkcja, która wywołuje samą siebie – poznaliśmy rekurencję

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-


def factorial(x):
    if x == 0:
        return 1
    else:
        return x * factorial(x-1)


if __name__ == '__main__':
    value = 6
    result = factorial(value)
    print(f'Factorial of {value} is {result}')

```

Rekurencja raz jeszcze – problem Collatza

- Weź dowolną liczbę naturalną.
- Jeśli jest parzysta, podziel ją przez 2.
- Jeśli jest nieparzysta pomnóż ją przez 3 i dodaj 1.
- Jeśli jest jedynką to skończ.
- Powyższy program zawsze w końcu dojdzie do jedynki dla dowolnej liczby całkowitej ale matematycy do dziś nie są w stanie udowodnić dlaczego.



Autor: Łukasz Kordowski

Rekurencja raz jeszcze – problem Collatza

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-


def collatz(number):
    print(number)
    if number == 1:
        return
    elif number % 2:
        return collatz(number * 3 + 1)
    else:
        return collatz(number // 2)


if __name__ == '__main__':
    user_number = int(input('Input a number: '))
    print(f'Collatz sequence for number {user_number} is: ')
    collatz(user_number)

```

```
Input a number: 10
Collatz sequence for number 10 is:
10
5
16
8
4
2
1
```

Autor: Łukasz Kordowski

Funkcje raz jeszcze

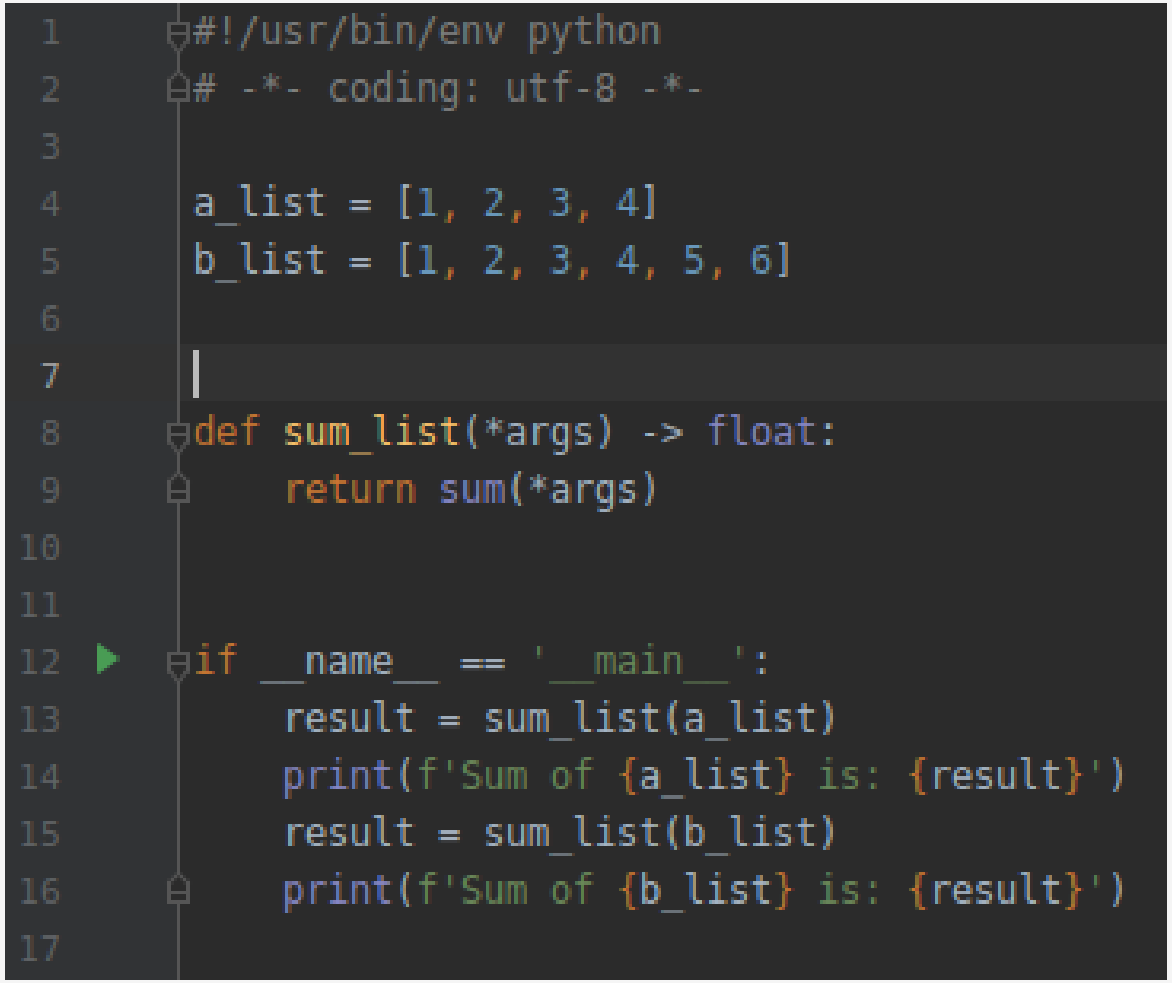
- Typowanie parametrów
- Typowanie wartości zwracanej
- Wywołanie z nazwą parametru
- Domyślna wartość parametru
- Możliwość podania parametru opcjonalnego przez podanie wartości domyślnej *None*

```
Factorial of 6 is 720
Factorial of default is 6
```

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-


def factorial(x: int = 3) -> int:
    if x == 0:
        return 1
    else:
        return x * factorial(x-1)


if __name__ == '__main__':
    value = 6
    result = factorial(x=value)
    print(f'Factorial of {value} is {result}')
    result = factorial()
    print(f'Factorial of default is {result}')

```

Autor: Łukasz Kordowski

Funkcje raz jeszcze



```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

a_list = [1, 2, 3, 4]
b_list = [1, 2, 3, 4, 5, 6]


def sum_list(*args) -> float:
    return sum(*args)


if __name__ == '__main__':
    result = sum_list(a_list)
    print(f'Sum of {a_list} is: {result}')
    result = sum_list(b_list)
    print(f'Sum of {b_list} is: {result}')
```

- Zmienna liczba argumentów
- Założeniem jest otrzymanie na wejściu listy (a właściwie elementu iterowalnego)

```
Sum of [1, 2, 3, 4] is: 10
Sum of [1, 2, 3, 4, 5, 6] is: 21
```

Autor: Łukasz Kordowski

Struktury danych - listy

- Najbardziej podstawową strukturą danych jest lista
- Każdy element listy posiada przypisany adres / indeks
- Indeksy rozpoczynamy od 0
- Indeksy mogą być ujemne, wybieramy wtedy elementy od końca listy
- Elementem listy może być obiekt
- Elementy listy mogą być różnego typu

```
>>> a
[1, 2]
>>> type(a)
<class 'list'>
```

```
>>> a_list = [1, 2, 3, 4, 5, 6]
>>> a_list[-1]
6
>>> a_list[-2]
5
```

Autor: Łukasz Kordowski

Listy

- Tworzenie listy następuje poprzez podanie elementów rozdzielonych przecinkami w nawiasach kwadratowych
- Aktualizacja dowolnego pola realizowana jest przez przypisanie wartości pod dany indeks
- Usunięcie elementu wykonujemy za pomocą znacznika *del*

```
>>> a_list = [1, 'abc', True, 2]
>>> a_list[0]
1
>>> a_list[1:3]
['abc', True]
>>> a_list[2] = 0.5
>>> a_list
[1, 'abc', 0.5, 2]
>>> del a_list[0]
>>> a_list
['abc', 0.5, 2]
```

Autor: Łukasz Kordowski

Listy – podstawowe operacje

Wyrażenie	Opis
len([1, 2, 3])	Długość
[1, 2, 3] + [4, 5, 6]	Łączenie list
[„A”] * 4	Powtarzanie list
3 in [1, 2, 3]	Przynależność do listy

```
>>> len([1, 2, 3])
3
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> ["A"] * 4
['A', 'A', 'A', 'A']
>>> 3 in [1, 2, 3]
True
```

Autor: Łukasz Kordowski

Listy – szatkowanie

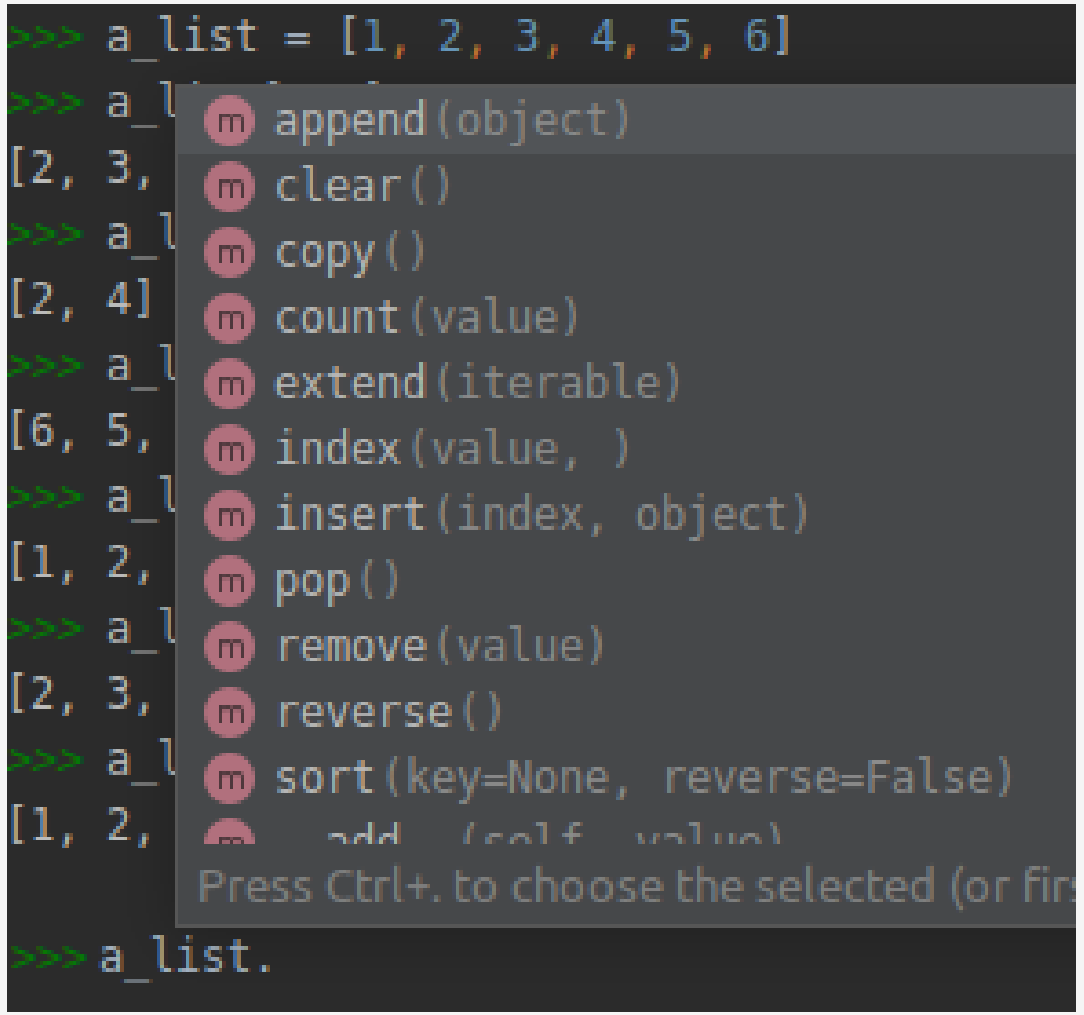
- Listy umożliwiają wydzielenie ich fragmentów
- Odbywa się to przez podanie w nawiasach liczb oddzielonych dwukropkiem, czyli podanie zakresu
- Kolejne liczby oznaczają początek i koniec wycinka oraz krok postępu, domyślnie o wartości 1

```
>>> a_list = [1, 2, 3, 4, 5, 6]
>>> a_list[1:4]
[2, 3, 4]
>>> a_list[1:5:2]
[2, 4]
>>> a_list[::-1]
[6, 5, 4, 3, 2, 1]
>>> a_list[::]
[1, 2, 3, 4, 5, 6]
>>> a_list[1:]
[2, 3, 4, 5, 6]
>>> a_list[:-1]
[1, 2, 3, 4, 5]
```

Autor: Łukasz Kordowski

Listy – metody

Metoda	Opis
.append(obj)	Dopisanie elementu **obj** na końcu listy
.count(obj)	Zliczenie wystąpień elementu **obj**
.extend(seq)	Dopisanie sekwencji do listy
.index(obj)	Najniższy indeks elementu **obj**
.pop()	Usuwa i zwraca ostatni element z listy
.remove(obj)	Usuwa element **obj** z listy
.reverse()	Zwraca listę w odwrotnej kolejności
.sort([func])	Sortuje listę z możliwością podania metody
sorted(seq)	Zwraca posortowaną listę



Autor: Łukasz Kordowski

Listy – rozpakowanie

- Elementy listy można przypisać do zmiennych nie tylko odnosząc się do indeksów elementów
- Python posiada ciekawy mechanizm rozpakowywania listy na zmienne

```
>>> a_list = [1, 2, 3]
>>> a, b, c = a_list
>>> a
1
>>> b
2
>>> c
3
>>> x, *y = a_list
>>> x
1
>>> y
[2, 3]
```

Autor: Łukasz Kordowski

Funkcja wbudowana zip

- Jako parametry funkcja przyjmuje dwie lub więcej list
- Wynikiem działania jest lista kolejnych elementów, sparowanych według indeksu z każdej z list na wejściu funkcji
- Jeśli listy mają różne długości, to wynik będzie miał długość najkrótszej z nich

```
>>> a_list = [1, 2, 3, 4]
>>> b_list = ['a', 'b', 'c', 'd']
>>> zip(a_list, b_list)
<zip object at 0x7f8a312a9e08>
>>> list(zip(a_list, b_list))
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

Autor: Łukasz Kordowski

Funkcja wbudowana range

- Jedną z ostatnich ważnych dla nas funkcji jest **range** (ściśle biorąc nie jest to funkcja ale my możemy ją tak traktować).
- **range** zwraca sekwencję liczb całkowitych.
- **range**(n) zwraca sekwencję liczb od 0 do n-1.
- **range**(a, b) zwraca sekwencję liczb od a do b-1.
- **range**(a, b, c) zwraca sekwencję liczb [a, a+c, a+c+c, ...] aż do b-1.
- Używając funkcji range można szybko tworzyć listy.

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
```

Autor: Łukasz Kordowski

Pętla for

- Pozwala wykonać zaplanowaną operację określoną liczbę razy

```
>>> def for_hello(name):
...     for i in range(3):
...         print(f'Hello {name}!')
...
>>> for_hello('Python')
Hello Python!
Hello Python!
Hello Python!
```

Autor: Łukasz Kordowski

Pętla for

- W Pythonie często wykorzystujemy pętlę ***for*** do iterowania po elementach list

```
>>> for x in 'Python':
...     print(f'Current x value: {x}')
...
Current x value: P
Current x value: y
Current x value: t
Current x value: h
Current x value: o
Current x value: n
```

Autor: Łukasz Kordowski

Pętla for z indeksem

- Oprócz wartości elementu możemy pobrać także jego indeks w danej liście

```
>>> for index, value in enumerate('Python'):
...     print(f'Current value: {value} on index: {index}')
...
Current value: P on index: 0
Current value: y on index: 1
Current value: t on index: 2
Current value: h on index: 3
Current value: o on index: 4
Current value: n on index: 5
```

Autor: Łukasz Kordowski

Pętla for z else?

- Po wyczerpaniu elementów lub braku elementów sekwencji, po której iterujemy możemy wykonać osobny blok kodu

```
>>> data = range(3)
>>> for x in data:
...     print(f'Current x value: {x}')
... else:
...     print('End of string')
...
Current x value: 0
Current x value: 1
Current x value: 2
End of string
>>> data = []
>>> for x in data:
...     print(f'Current x value: {x}')
... else:
...     print('End of string')
```

Autor: Łukasz Kordowski

Pętla while

- Pętla **while** działa tak długo jak długo spełniony jest logiczny warunek podany w jej wywołaniu

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-


def count_down(number):
    while number:
        print(number)
        number -= 1
    print('Now we know while loop!')


if __name__ == '__main__':
    count_down(10)
```

```
10
9
8
7
6
5
4
3
2
1
Now we know while loop!
```

Autor: Łukasz Kordowski

Pętle – słowa kluczowe break i continue

- Słowo kluczowe **break** przerywa wykonanie pętli, nawet jeśli warunek pętli jest spełniony.
- Słowo kluczowe **continue** przerywa bieżącą iterację i przechodzi do następnego obiegu pętli.

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-


def count_down(number):
    while number:
        print(number)
        number -= 1
        if number == 5:
            print('Aborted!')
            break
    print('Now we know while loop!')


if __name__ == '__main__':
    count_down(10)
```

```
10
9
8
7
6
Aborted!
Now we know while loop!
```

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-


def count_down(number):
    while number:
        number -= 1
        if number % 2:
            print(number)
            continue
    print('Now we know while loop!')


if __name__ == '__main__':
    count_down(10)
```

```
9
7
5
3
1
Now we know while loop!
```

Autor: Łukasz Kordowski

Kolejny program – testy DNA

- Ojcostwo ustala się przy pomocy badań genetycznych.
- Materiał genetyczny potencjalnych ojców jest porównywany z materiałem dziecka.
- Za ojca uznaje się osobę, której materiał genetyczny jest najbardziej podobny do dziecka.
- Materiał genetyczny to nić DNA - ciąg nukleotydów (znaków składających się z czterech liter: A, C, T oraz G).
- Aby obliczyć podobieństwo dwóch nici DNA używa się tzw. odległości Hamminga.
- Polega to na tym, że patrzy się na kolejne litery dwóch nici - za każdą niezgodność liter przyznaje się jeden punkt. Odległość jest sumą punktów po całej nici.
- DNA ojca ma najmniejszą odległość Hamminga spośród wszystkich kandydatów.
- Oczywiście to bardzo uproszczony model i **tak się tego nie robi**!

Autor: Łukasz Kordowski

Kolejny program – testy DNA

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

suspect_1 = 'GAGCCTACTAACGGGAT'
suspect_2 = 'GAGCCTACTAACAAAAT'
child = 'CATCGTAATGACGGCCT'


def hamming(strand_a, strand_b):
    result = 0
    zipped_strands = zip(strand_a, strand_b)
    for item_a, item_b in zipped_strands:
        if item_a != item_b:
            result += 1
    return result


if __name__ == '__main__':
    print(f'Suspect #1: {suspect_1}')
    print(f'Suspect #2: {suspect_2}')
    print(f'Child: {child}')
    distance_1 = hamming(suspect_1, child)
    distance_2 = hamming(suspect_2, child)
    if distance_1 < distance_2:
        print('Suspect #1 is a father')
    else:
        print('Suspect #2 is a father')
```

- Na początku odległość wynosi 0.
- Łączymy w pary litery na obu niciach.
- Dla każdej pary sprawdzamy czy jej elementy są różne.
- Jeśli tak to zwiększamy dystans o jeden.
- W ten sposób liczymy odległość pomiędzy DNA dziecka i obu potencjalnych ojców.
- Za ojca uznajemy tego, którego odległość od DNA dziecka jest mniejsza.

```
Suspect #1: GAGCCTACTAACGGGAT
Suspect #2: GAGCCTACTAACAAAAT
Child: CATCGTAATGACGGCCT
Suspect #1 is a father
```

Autor: Łukasz Kordowski

Zbiór - set

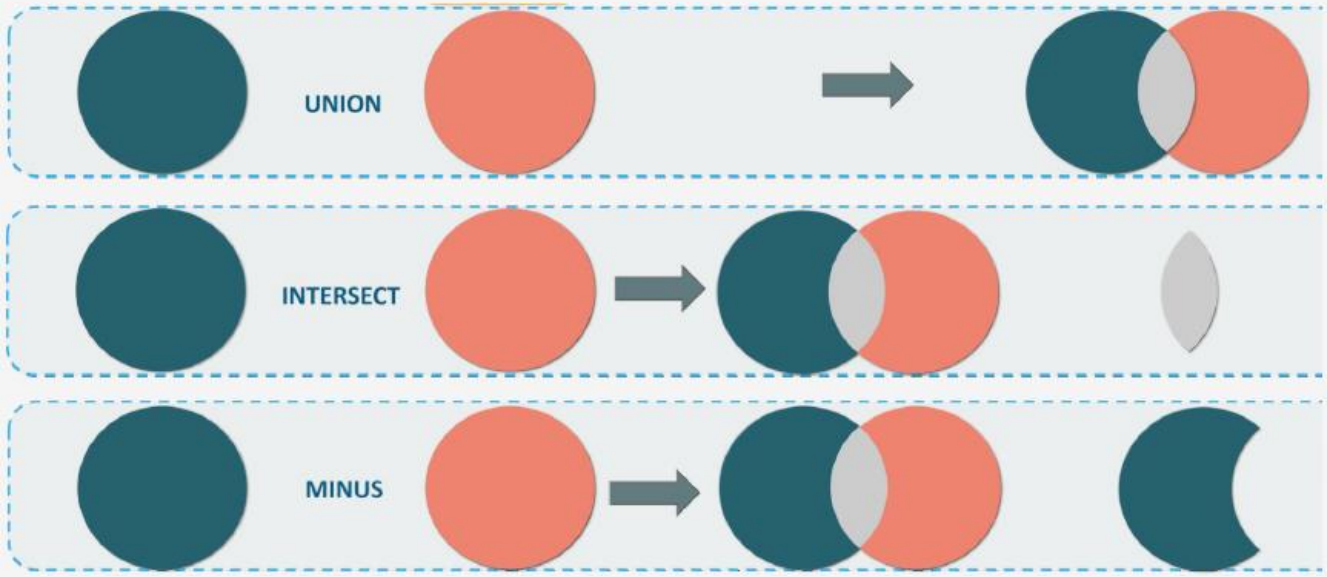
- Zbiór tworzymy podając wartości, rozdzielone przecinkami, w nawiasach klamrowych
- Zawiera unikalne elementy, jego wartości nie powtarzają się
- Kolejność elementów nie jest ważna

```
>>> {1, 2, 3}
{1, 2, 3}
>>> {1, 2, 3}[0]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'set' object does not support indexing
>>> {1, 2, 3, 1, 2, 1}
{1, 2, 3}
>>> {2, 1, 3}
{1, 2, 3}
```

Autor: Łukasz Kordowski

Podstawowe operacje na zbiorach

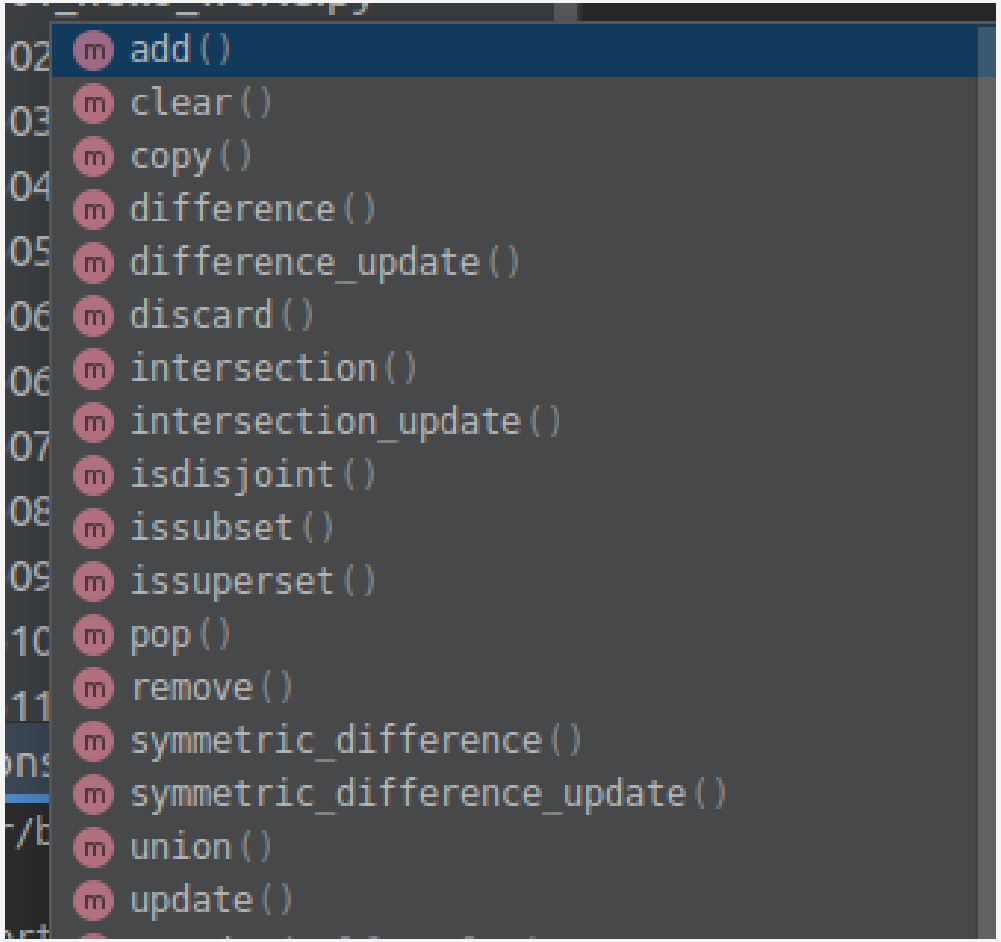
- Na zbiorach możemy wykonywać podstawowe działania:
  - Sumowanie
  - Iloczyn
  - Różnica




```
>>> set_a = {1, 2, 3, 4, 5}
>>> set_b = {4, 5, 6, 7, 8}
>>> set_a | set_b
{1, 2, 3, 4, 5, 6, 7, 8}
>>> set_a.union(set_b)
{1, 2, 3, 4, 5, 6, 7, 8}
>>> set_a & set_b
{4, 5}
>>> set_a.intersection(set_b)
{4, 5}
>>> set_a - set_b
{1, 2, 3}
>>> set_b - set_a
{8, 6, 7}
>>> set_a ^ set_b
{1, 2, 3, 6, 7, 8}
```

Podstawowe operacje na zbiorach

Metoda	Opis
.add()	Dodaje element do zbioru
.clear()	Usuwa wszystkie elementy ze zbioru
.difference()	Zwraca różnicę zbiorów
.difference_update()	Aplikuje różnicę zbiorów na pierwszy zbiór
.isdisjoint()	Sprawdza czy zbiory są rozłączne
.issubset()	Sprawdza czy zbiór zawiera się w podanym zbiorze
.issuperset()	Sprawdza czy zbiór zawiera w sobie podany zbiór
.remove()	Usuwa podany element ze zbioru
.union()	Zwraca sumę zbiorów



Autor: Łukasz Kordowski

Izogramy

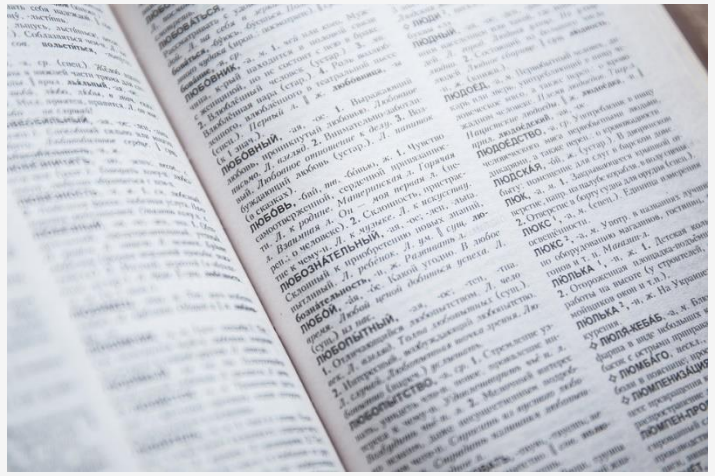
```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-


def is_isogram(word):
    letters = set()
    for letter in word.lower():
        if letter in letters:
            return False
        letters.add(letter)
    return True


if __name__ == '__main__':
    while True:
        my_word = input('Input a word: ').strip()
        answer = 'is' if is_isogram(my_word) else 'is not'
        print(f'Word {my_word} {answer} an isogram\n')
        shall_continue = input('Do you want to continue (y/n)?: ')
        if shall_continue != 'y':
            break

```

- Izogram jest słowem, w którym żadna litera nie powtarza się

Autor: Łukasz Kordowski

Słownik - dict

- Ta struktura danych jest podobna do książki telefonicznej lub encyklopedii.
- Zaglądamy pod pewien klucz (nazwisko abonenta, hasło encyklopedyczne) a w zamian dostajemy pewną użyteczną wartość (numer telefonu, definicję pojęcia)
- Słownik przypomina zbiór w tym sensie, że jego klucze muszą być unikalne.
- Jednak zbiór nie wiąże klucza z żadną wartością a słownik to robi.



Autor: Łukasz Kordowski

Słownik - dict

- Słownik tworzymy wypisując pary klucz:wartość, oddzielając dwukropkiem klucz od wartości. Pary oddzielamy od siebie przecinkami, całość jest ujęta w nawiasy klamrowe.
- Do wartości ze słownika możemy odnieść się poprzez korespondujący z nim klucz używając operatora **[ ]**.
- Tego samego operatora można użyć aby zmienić wartość spod danego klucza.
- Aby usunąć klucz (i jego wartość) ze słownika należy użyć poznanego już słowa kluczowego **del**.
- Wartości w słowniku mogą być różnych typów. Podobnie klucze ale one muszą spełniać pewne minimalne wymagania, które omówimy później.

```
>>> shopping = {'apple': 2, 'pear': 3, 'tomato': 1}
>>> shopping['apple']
2
>>> shopping['banana'] = 5
>>> shopping
{'apple': 2, 'pear': 3, 'tomato': 1, 'banana': 5}
>>> del shopping['apple']
>>> shopping
{'pear': 3, 'tomato': 1, 'banana': 5}
>>> shopping['banana'] = 1000
>>> shopping
{'pear': 3, 'tomato': 1, 'banana': 1000}
```

Autor: Łukasz Kordowski

Słownik - dict

- Podobnie jak zbiór, słownik również jest zoptymalizowany pod kątem sprawdzania czy dany klucz w nim istnieje oraz wyciągania wartości spod danego klucza. Dlatego też nie ma metody **index**.
- Jeśli użyjemy operatora **[ ]** w celu dostania się pod klucz, który nie istnieje w słowniku to dostaniemy błąd.
- Aby ustrzec się przed błędem w takiej sytuacji należy użyć metody **get**, która zwróci **None** jeśli klucza nie ma lub wartość domyślną jeśli ją podamy.
- Pusty słownik możemy stworzyć przy użyciu literału **{ }**.

```
>>> shopping['carrot']
Traceback (most recent call last):
  File "/usr/lib/python3.6/code.py", line 91, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
KeyError: 'carrot'
>>> shopping.get('carrot', 0)
0
```

Słownik - dict

- Słownik udostępnia trzy ważne metody:
  - **keys** - zwraca zbiór wszystkich kluczy
  - **values** - zwraca zbiór wszystkich wartości
  - **items** - zwraca zbiór wszystkich par (klucz, wartość).
- Należy pamiętać, że kiedy iterujemy po słowniku w pętli **for** to każdy kolejny element jest kluczem a nie parą (klucz, wartość).
- Operator **in** umożliwia sprawdzenie czy dany klucz (ale nie wartość) jest w słowniku.

```
>>> shopping
{'apple': 3, 'orange': 4, 'banana': 3}
>>> list(shopping.keys())
['apple', 'orange', 'banana']
>>> list(shopping.values())
[3, 4, 3]
>>> list(shopping.items())
[('apple', 3), ('orange', 4), ('banana', 3)]
```

Autor: Łukasz Kordowski

Scrabble

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
scores = {
    "a": 1, "b": 3, "c": 3, "d": 2, "e": 1, "f": 4, "g": 2, "h": 4,
    "i": 1, "j": 8, "k": 5, "l": 1, "m": 3, "n": 1, "o": 1, "p": 3,
    "q": 10, "r": 1, "s": 1, "t": 1, "u": 1, "v": 4, "w": 4, "x": 8,
    "y": 4, "z": 10
}


def scrabble_score(word):
    total_score = 0
    for letter in word.lower():
        total_score += scores[letter]
    return total_score


if __name__ == '__main__':
    while True:
        my_word = input('Input a word: ').strip()
        score = scrabble_score(my_word)
        print(f'Word {my_word} is worth {score} points\n')
        shall_continue = input('Do you want to continue (y/n)?: ')
        if shall_continue != 'y':
            break

```

- W grze Scrabble każda litera ma swoją wartość punktową.
- Należy napisać program, który poprosi użytkownika o słowo i wyliczy jego wartość punktową na podstawie wartości każdej z liter (pomijamy premie).

```
Input a word: metafora
Word metafora is worth 13 points

Do you want to continue (y/n)?: y
Input a word: python
Word python is worth 14 points

Do you want to continue (y/n)?: n
```

Autor: Łukasz Kordowski

List comprehension – wyrażenie listowe

- Wyrażenia listowe są innym sposobem na tworzenie listy, zamiast podawania jej elementów.
- W wyrażeniu listowym nową listę tworzymy ze starej. Podczas tej operacji, na każdy element starej list możemy nałożyć jakieś przekształcenie.
- Ponadto elementy starej list możemy odfiltrować.
- Poniżej używamy wyrażenia listowego do stworzenia nowej listy jedynie z parzystych elementów starej listy podnosząc do kwadratu każdy z elementów, który przeszedł przez filtr.

```
>>> list_a
[0, 1, 2, 3, 4, 5, 6]
>>> list_b = [x ** 2 for x in list_a if not x % 2]
>>> list_b
[0, 4, 16, 36]
```

Autor: Łukasz Kordowski

Dict i set comprehension

- Analogiczne działania możemy wykonać na słownikach i zbiorach

```
>>> shopping = {'apple': 3, 'orange': 4, 'banana': 3}
>>> to_shop = {v: k for k, v in shopping.items()}
>>> to_shop
{3: 'banana', 4: 'orange'}
>>> to_shop = {k: v for k, v in shopping.items() if v < 4}
>>> to_shop
{'apple': 3, 'banana': 3}
```

```
>>> word_list = ['ala', 'ola', 'alex', 'anna', 'al']
>>> word_lengths = {len(word) for word in word_list}
>>> word_lengths
{2, 3, 4}
```

Autor: Łukasz Kordowski