



Python

The Python logo consists of two interlocking snakes, rendered here in blue and yellow. The blue snake is positioned above the yellow one, and both have small black eyes.

Podstawy

Michał Nowotka

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development Academy

Python - Historia języka



- Napisany przez Guido van Rossum i wydany w 1990 (29 lat temu!)
- Trzy główne wersje, najnowsza stabilna to 3.7.3.
- Wersja 2.7 przestanie być wspierana w 2020.
- Wbrew pozorom nazwa nie pochodzi od węża a nazwy programu “Latający cyrk Monty Pythona.”
- Napisany jako język skryptowy dla rozproszonego systemu operacyjnego Amoeba, który nie zdobył wielkiej popularności.



Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development Academy

Python - Popularność języka



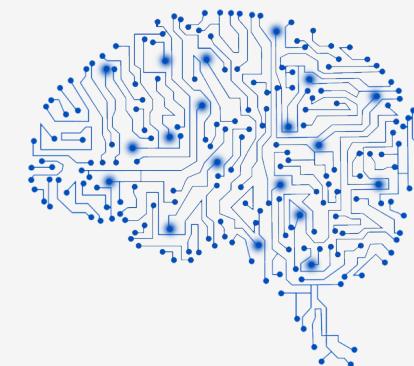
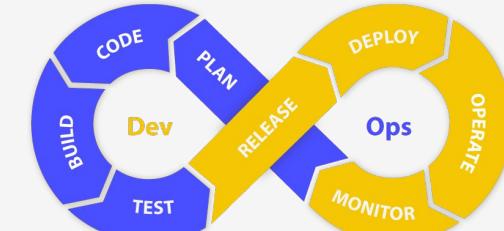
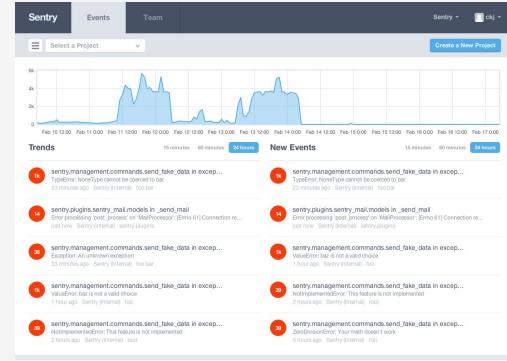
- Według [Stack Overflow Developer Survey](#) (najbardziej znaczącego badania opinii programistów) Python jest najszybciej rosnącym w popularność językiem.
- Jest [bardziej popularny](#) niż Java.
- Dynamiczny rozwój sztucznej inteligencji (AI) oraz technologii webowych przemawia za dalszymi wzrostami.
- **Jesteś na właściwym kursie!**



Python - Najważniejsze zastosowania



- Technologie webowe:
 - Aplikacje webowe (Django, Flask)
 - REST-owe API - silniki stron (Django, Flask)
 - Internetowe roboty indeksujące (requests, BeautifulSoup)
- Administracja systemów (DevOps)
- Obróbka i przetwarzanie danych (pandas, numpy):
 - Wczytywanie plików csv ixlsx
 - Uzupełnianie brakujących danych, czyszczenie danych
 - Analiza danych i wizualizacja
- Uczenie maszynowe (Scikit-learn, TensorFlow):
 - Wykrywanie obiektów na zdjęciach (twarze, samochody, itd)
 - Analiza ludzkiego języka np. automatyczne streszczenia tekstu



Python - Podstawowe cechy



- **Interpretowalny** - nie trzeba go komplilować oraz posiada własny dedykowany interpreter
- **Wysokopoziomowy** - operujesz na pojęciach z domeną problemu, który rozwiążujesz (np. dla banku będą to waluty, kredyty, konta, przelewy) a nie na bitach i rejestrach procesora.
- **Dynamicznie typowany** - deklarując zmienne nie musisz podawać ich typu oraz możesz w nich trzymać co zechcesz.
- Wspiera wiele stylów (**Paradygmatów**) programowania:
 - Obiektowy (klasy, metody, dziedziczenie)
 - Funkcyjny (wyrażenia lambda)
 - Imperatywny (pętle)
 - Refleksyjny (zaglądanie do zmiennych, modułów, klas)



Python - Pierwszy program



```
[ MacBook-Pro-Micha ] ~
[> ipython
Python 3.7.1 (default, Nov 28 2018, 11:51:47)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.2.0 -- An enhanced Interactive Python. Type '?' for help.

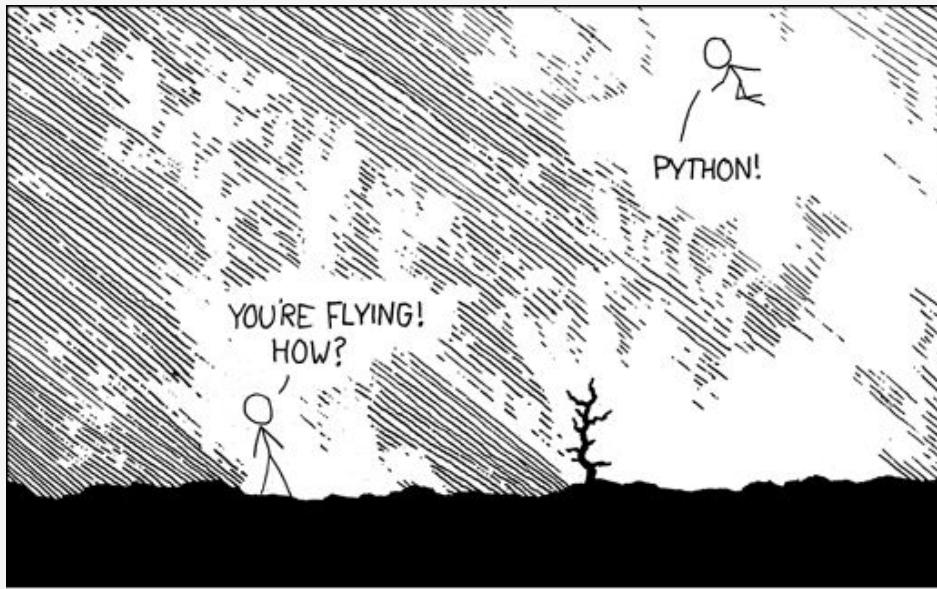
In [1]: print("Witaj Świecie")
Witaj Świecie

In [2]: save hello 0/1
The following commands were written to file `hello.py`:
print("Witaj Świecie")

In [3]:
Do you really want to exit ([y]/n)?

[ MacBook-Pro-Micha ] ~
[> python hello.py
Witaj Świecie
```

Python - Pierwszy program



Spróbuj sam wpisać w interpreterze
import antigravity
i zobacz co się stanie...

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development Academy

Python - Filozofia języka



```
MacBook-Pro-Micha ~
[> ipython
Python 3.7.1 (default, Nov 28 2018, 11:51:47)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.2.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import this
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!
```

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development Academy

Python - Charakterystyczne cechy na tle innych języków



- Używanie wcięć zamiast klamr.
- Pojedyncze wyrażenie nie musi kończyć się ś
- Zmienne nie mają deklaracji typu.

Python

```
some_list = [1, "foo", False]  
  
for element in some_list:  
    print(element)
```



Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development Academy

Python - to czysta przyjemność



*Programming is fun
When the work is done
If you wanna make your work also fun:
use Python!*

(Wierszyk pochodzi z książki “Dive into Python”)



Cegiełki programowania



- Funkcje
- Pętle
- Wyrażenia warunkowe
- Operatory
- Struktury danych
- Biblioteka standardowa



Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development Academy

Pierwszy nietrywialny program, obliczanie BMI



bmi.py

```
bmi.py x
1 def compute_bmi(weight, height):
2     bmi = weight / height ** 2
3     if bmi < 18.5:
4         result = 'underweight'
5     elif bmi > 25:
6         result = 'overweight'
7     else:
8         result = 'normal'
9     return result
10
11
12 if __name__ == '__main__':
13     user_weight = float(input('Your weight [kg]: '))
14     user_height = float(input('Your height [m]: '))
15     user_result = compute_bmi(user_weight, user_height)
16     print(f'You are {user_result}')
17
```

wynik:

```
MacBook-Pro-Micha ~
> python bmi.py
Your weight [kg]: 63
Your height [m]: 1.68
You are normal
```



```
def compute_bmi(weight, height):
```

- Funkcja to jasno wydzielona część kodu, która ma jedno konkretne zadanie.
- Deklarację funkcji rozpoczyna słowo kluczowe **def**.
- Po nim następuje nazwa funkcji (u nas: **compute_bmi**)
- W nawiasie podajemy argumenty funkcji, rozdzielone przecinkami (u nas są dwa: **weight** oraz **height**).
- Nie każda funkcja musi przyjmować argumenty.

Definicja funkcji



```
1  def compute_bmi(weight, height):  
2      bmi = weight / height ** 2 # coś tutaj liczymy  
3      ... # dalsze przetwarzanie  
4      return result # zwracamy wynik  
5  
6  foo = 5 # tu jesteśmy już poza funkcją
```

- Po zadeklarowaniu funkcji, w kolejnych linijkach mówimy co ta funkcja ma robić, czyli dostarczamy jej definicji.
- Każda linijka takiej definicji jest wcięta w stosunku do deklaracji.
- Koniec wcięcia kończy definicję.

Wyrażenia warunkowe



```
if bmi < 18.5:  
    result = 'underweight'  
elif bmi > 25:  
    result = 'overweight'  
else:  
    result = 'normal'
```

- Warunek zawsze rozpoczynamy słowem kluczowym **if**.
- Po nim następuje logiczne wyrażenie oznaczające treść warunku (u nas **bmi < 18.5**)
- Warunek kończymy dwukropkiem. We wciętym bloku poniżej warunku piszemy co ma się stać jeśli będzie on spełniony.
- Jeśli warunek nie jest spełniony możemy sprawdzić dodatkowe warunki używając słowa kluczowego **elif** (skrót od słów **else if**).
- Na samym końcu blok **else** wykona się jeśli żaden z warunków nie jest spełniony.

Wartość zwracana funkcji



```
def czy_jestem_starym_koniem(wiek):  
    if wiek > 18:  
        return "Ty stary koniu"  
    return "Jesteś OK"
```

- Na ogólnym celu funkcji jest zwrócenie jakiegoś wyniku.
- Zwrócenie wyniku kończy działanie funkcji.
- Do zwracania wyniku z funkcji używam słowa kluczowego **return**.
- Pamiętajmy, że w prawdziwym kodzie **zawsze** używamy języka angielskiego.

Całość funkcji jeszcze raz:



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

Punkt wejściowy skryptu



```
12 > if __name__ == '__main__':
```

- Wcięty blok pod tą linijką wykona się tylko wtedy jeśli uruchomimy plik, w którym się ona znajduje bezpośrednio na przykład poleceniem **python bmi.py**.
- Blok nie uruchomi się, kiedy plik wywołujemy pośrednio, na przykład importujemy go w innym module poleceniem **import bmi**.
- PyCharm wykrywa taki punkt wejściowy i produkuje zieloną strzałkę na marginesie. Jej kliknięcie działa jak bezpośrednie uruchomienie pliku.
- <https://www.youtube.com/watch?v=sugvnHA7EIY>

Pobieranie danych od użytkownika



```
user_weight = float(input('Your weight [kg]: '))
```

- W tej linijce przypisujemy do zmiennej **user_weight** wagę ciała pobraną od użytkownika.
- Dane od użytkownika pobiera się za pomocą standardowej funkcji **input**, która jako argument pobiera komunikat, który zostanie wyświetlony użytkownikowi (u nas: **Your weight [kg]:**)
- Funkcja **input** zwraca po prostu tekst, nam potrzebna jest liczba, dlatego dokonujemy konwersji tekstu do liczby za pomocą standardowej funkcji **float** - to oczywiście może się nie udać jeśli użytkownik wpisze coś bez sensu.

Wywołanie funkcji



```
user_result = compute_bmi(user_weight, user_height)
```

- W tej linijce wywołujemy wcześniej zdefiniowaną funkcję **compute_bmi**.
- Na wejście funkcji podajemy wagę i wzrost pobrane przez użytkownika i zapisane w zmiennych **user_weight** oraz **user_height**.
- Wynik zwracany przez funkcję przypisujemy z kolei do zmiennej **user_result**.
- Znak **=** znaczy w tym przypadku (jak i we wszystkich pozostałych linijkach) przypisanie a więc można go czytać: przypisz do tego co stoi po lewej stronie (a więc do zmiennej **user_result**) wartość, która stoi po prawej stronie (a więc wynik działania funkcji **compute_bmi**).

Drukowanie wyniku



```
print(f'You are {user_result}')
```

- Jak wiemy z pierwszego programu (**hello.py**), standardowa funkcja **print** drukuje na konsolę tekst podany jej jako argument.
- Jednak w tym przypadku tekst nie może być interpretowany dosłownie.
- Daliśmy o tym znać, stawiając przed tekstem literkę **f**.
- To znaczy, że stworzyliśmy tzw. **f-string**.
- W f-stringu można używać nazwy zmiennych zdefiniowanych w kodzie powyżej. W miejsce nazwy zmiennej zostanie podstawiona jej wartość.

Kilka słów o nazywaniu zmiennych i funkcji.



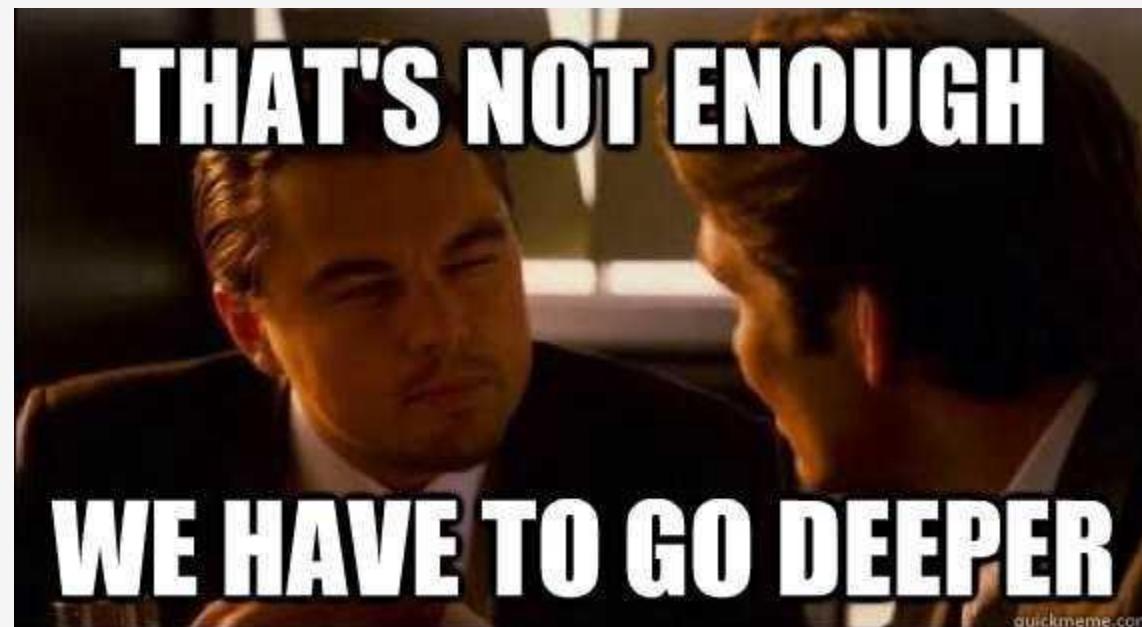
```
def compute_bmi(weight, height):  
    user_weight = float(input('Your weight [kg]: '))  
  
12 ► if __name__ == '__main__':
```

- Nazwy funkcji i zmiennych powinny zawsze zaczynać się małą literą.
- Python rozróżnia wielkość liter dlatego **foo** i **Foo** to dwie różne rzeczy.
- Jeśli nazwa składa się z kilku słów, należy rozdzielać je podkreślnikiem.
- Nie trzeba się bać długich nazw, za to należy unikać nazw krótkich, jednoliterowych.
- Podwójny podkreślnik nazywa się **dunder** (z angielskiego **double underscore**). Funkcje zaczynające i kończące się dunder'em (np. **__name__**) są specjalne.



UFFFF!

To było tylko 16 linijek kodu...



quickmeme.com

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development Academy

Przeżyjmy to jeszcze raz :)



```
def compute_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('Your weight [kg]: '))
    user_height = float(input('Your height [m]: '))
    user_result = compute_bmi(user_weight, user_height)
    print(f'You are {user_result}')
```

Typy danych w Pythonie



- **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)

```
foo = 1
foo = 1.2
foo = 'bar'
foo = False
foo = None
foo = [1,2,3]
foo = (1,2,3)
foo = {1,2,3}
foo = {'spam': 1, 'eggs': 2}
```

No dobrze, a co to jest to foo?



- 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.
- W takich przypadkach programiści tradycyjnie stosują ogólnie przyjęte nazwy zastępcze (egzemplifikaty).
- Najbardziej popularnymi zwyczajowymi nazwami są **foo**, **bar** oraz **baz**.
- W Pythonie przyjęło się stosować również **spam** i **eggs** ponieważ są to nazwy bezpośrednio zaczerpnięte z programu Latający Cyrk Monty Pythona.

W Pythonie wszystko jest obiektem



- Liczby
- Teksty
- Funkcje
- Moduły
- Nawet specjalny typ **None**

Wszystko to (i więcej) jest obiektem.

- Dzięki temu na każdym obiekcie możemy wywołać przydatne funkcje wbudowane:
 - **dir** - wypisz dostępne metody i atrybuty
 - **type** - wypisz typ obiektu
 - **help** - wypisze dokumentację

```
In [1]: isinstance(1, object)
Out[1]: True

In [2]: isinstance("ala ma kota", object)
Out[2]: True

In [3]: isinstance(isinstance, object)
Out[3]: True

In [4]: isinstance(None, object)
Out[4]: True

In [5]: isinstance(False, object)
Out[5]: True

In [6]: import math

In [7]: isinstance(math, object)
Out[7]: True
```

W Pythonie wszystko jest obiektem



- Dzięki temu na każdym obiekcie możemy wywołać przydatne funkcje:
 - **dir** - wypisz dostępne metody i atrybuty
 - **type** - wypisz typ obiektu
 - **help** - wypisze dokumentację

```
In [1]: type(None)
Out[1]: NoneType

In [2]: type(1)
Out[2]: int

In [3]: type(1.2)
Out[3]: float

In [4]: type(True)
Out[4]: bool

In [5]: dir("ala ma kota")
Out[5]:
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 ...
'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans',
 'partition',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 ...
...]
```

Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development Academy

Proste typy



- Pewne typy są proste, np. numeryczne typy mają wszystkie operacje, których uczyliśmy się w szkole.
- Jedyna uwaga to taka, że w Pythonie 3 operacja dzielenia dwóch liczb całkowitych może zwrócić liczbę zmiennoprzecinkową, tymczasem w Pythonie 2 zwróci liczbę całkowitą, a więc w Pythonie 2 wynikiem operacji $\frac{2}{3}$ będzie 0.
- Abytrzymać taką wartość w Pythonie 3 należy użyć operatora `//` oznaczającego część całkowitą z dzielenia.
- W informatyce ważną rolę pełni operator modulo (`%`) oznaczający resztę z dzielenia.

```
In [1]: 2 + 2
Out[1]: 4

In [2]: 2 + 2 * 2
Out[2]: 6

In [3]: 2 - 2
Out[3]: 0

In [4]: 2 ** 6
Out[4]: 64

In [5]: 2/3
Out[5]: 0.6666666666666666

In [6]: 2 // 3
Out[6]: 0

In [7]: 5 % 3
Out[7]: 2
```

Proste typy cd.



- Wyrażenia logiczne również podlegają prawom, które poznaliśmy w szkole.
- Należy przy tym pamiętać, że wartości logiczne łączy się ze sobą innymi operatorami niż liczby.
- Operatory te to:
 - Koniunkcja - **and**
 - Alternatywa - **or**
 - Negacja - **not**

```
In [1]: True and False
```

```
Out[1]: False
```

```
In [2]: True and True
```

```
Out[2]: True
```

```
In [3]: True or False
```

```
Out[3]: True
```

```
In [4]: False or False
```

```
Out[4]: False
```

```
In [5]: not True
```

```
Out[5]: False
```



Operatory porównania

- Operatory porównania przyjmują dwa obiekty (mogą ale nie muszą to być liczby) a zwracają wartość logiczną.
- Warto zauważyć, że operator równości to `==`, ponieważ `=` jest operatorem przypisania.
- Operator nierówności to `!=`.
- Ciągi znaków też można porównywać, będą one wtedy porównywane leksykograficznie.

```
In [1]: 2 > 3  
Out[1]: False  
  
In [2]: 3 > 2  
Out[2]: True  
  
In [3]: 2 == 3  
Out[3]: False  
  
In [4]: 2 == 2  
Out[4]: True  
  
In [5]: 2 >= 2  
Out[5]: True  
  
In [6]: 2 <= 2  
Out[6]: True  
  
In [7]: 2 != 2  
Out[7]: False
```

```
In [8]: 2 != 3  
Out[8]: True  
  
In [9]: 'ala' > 'kota'  
Out[9]: False  
  
In [10]: 'ala' < 'kota'  
Out[10]: True  
  
In [11]: 'ala' == 'kota'  
Out[11]: False  
  
In [12]: 'ala' != 'kota'  
Out[12]: True  
  
In [13]: 'ala' == 'ala'  
Out[13]: True
```

Operatory przypisania



- Operatory przypisania są pewnego rodzajem skrótowym zapisem.
- Jeśli mamy pewną zmienną, w której trzymamy liczbę, jak zwiększyć wartość zmiennej o ileś, np. o 2?
- Analogicznie do dodawania (operator `+=`) mamy również mnożenie, dzielenie, potęgowanie i modulo.
Odpowiednio: `*=`, `/=`, `**=`, `%=`.

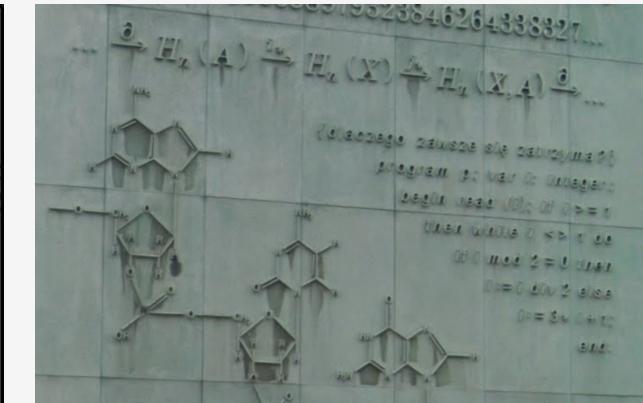
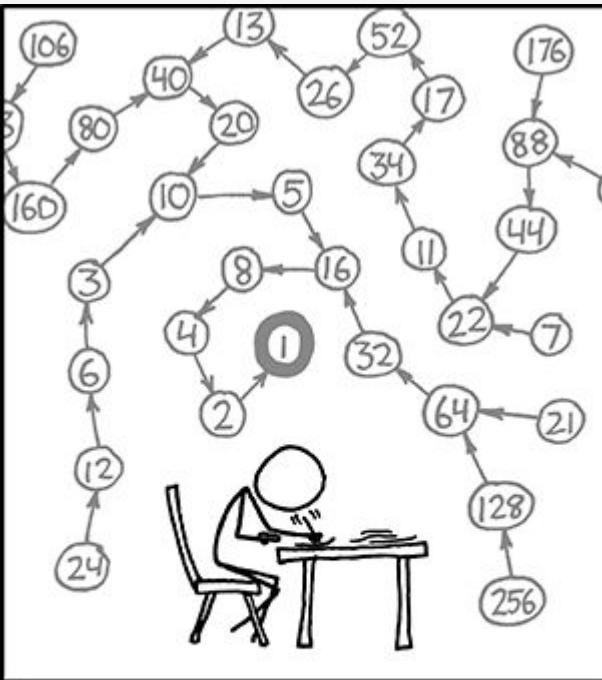
```
In [1]: my_variable = 5
In [2]: my_variable = my_variable + 2
In [3]: print(my_variable)
7

In [4]: my_variable = 5
In [5]: my_variable += 2
In [6]: print(my_variable)
7
```

Pora na kolejny program - 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.
- Implementację programu w języku Pascal można znaleźć na frontonie Biblioteki Uniwersytetu Warszawskiego.



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

Pora na kolejny program - problem Collatza



- Wypisz liczbę daną na wejściu funkcji.
- Jeśli ta liczba to jedynka to zakończ.
- W przeciwnym przypadku sprawdź czy liczba jest nieparzysta (modulo!)
- Jeśli tak, to wywołaj samą tę samą funkcję (samą siebie) z liczbą z wejścia pomnożoną przez 3 plus 1.
- W przeciwnym przypadku (czyli kiedy liczba jest parzysta) wywołaj tę samą funkcję z liczbą z wejścia podzieloną przez 2.

```
1  def collatz(number):  
2      print(number)  
3      if number == 1:  
4          return  
5      elif number % 2:  
6          return collatz(number * 3 + 1)  
7      else:  
8          return collatz(number // 2)  
9  
10  
11 ➤ if __name__ == '__main__':  
12     my_number = int(input('Give a number: '))  
13     print(f'Collatz sequence for number {my_number} is: ')  
14     collatz(my_number)
```

Pora na kolejny program - problem Collatza



- Dla 11 ciąg zbiegł do jedynki po drodze wypisując 13 innych liczb.
- Inne liczby szybciej lub wolniej zbiegną do jedynki ale zawsze się to na końcu uda.
- Poznaliśmy nową technikę programowania - **rekurencję!**
- Rekurencyjna funkcja to taka, która wywołuje samą siebie.
- Jeśli brzmi to skomplikowanie, nie martw się, żeby zrozumieć rekurencję musisz najpierw zrozumieć rekurencję :)
- Tak na poważnie, to będziemy do niej wracać!

```
MacBook-Pro-Micha:~/cli$ python collatz.py
Give a number: 11
Collatz sequence for number 11 is:
11
34
17
52
26
13
40
20
10
5
16
8
4
2
1
```

Typy złożone - struktury danych



- Proste typy są w miarę proste do zrozumienia bazując na intuicjach, które wynieśliśmy z podstawowej matematyki i logiki.
- Typy złożone posiadają wiele metod, z którymi trzeba się zapoznać, dlatego są nieco trudniejsze do zrozumienia ale niezbędne do efektywnego programowania.
- Skupimy się przede wszystkim na listach, zbiorach i słownikach.



Typy złożone - lista

- Lista jest uporządkowaną kolekcją obiektów.
- Wyobraź sobie kartę dań w restauracji, gdzie każdemu danemu jest przyporządkowana liczba porządkowa - to jest właśnie lista.
- Kiedy składasz zamówienie podajesz po prostu numery dań.
- Jeśli restauracja chce dodać nowe zamówienie do karty dań, najprościej je dopisać do końca listy i nadać mu kolejną liczbę porządkową.
- Sytuacja komplikuje się, kiedy restauracja chce dodać nowe danie na początek lub gdzieś w środku listy, ponieważ trzeba wtedy zmienić numerację pozostałych dań.
- To samo następuje kiedy usuwamy - łatwo skreślić ostatni element ale kiedy wykreślamy element ze środka to przydałoby się zmienić liczbę porządkową innych dań.
- Nie jest to oczywiście niemożliwe ale zajmie więcej czasu niż dodawanie / usuwanie ostatniego elementu.

APPETIZERS		
1. Egg Roll (vegetarian) (2)	1.99
2. Potsticker (Pork) (6)	4.85
3. Fried Won Ton (Pork) (8)	3.50
4. Paper Wrapped Chicken (6)	4.85
5. Cream Cheese Fried Won Ton (8)	3.50
6. Fried Shrimp (4)	5.15
Crunch Noodle (bag)	0.75
SALAD		
Chicken Salad	4.75
(Fresh garden salad with chicken in Chef special dressing topped with crunch noodles.)		
SOUPS		
Small (1-3)		Large (4-6)
7. Won Ton Soup	4.95
★ 8. Hot & Sour Soup	4.85
9. Egg Drop Soup	3.85
10. Vegetable Soup	3.85
CHICKEN		
(Any Dark Meat Change White Meat Add 1.00)		
★11. Kong Pao Chicken	6.75
(Diced chicken w/peanuts & green onions.)		
12. Moo Goo Gai Pan	6.75
(Sliced chicken breast w/mushrooms, snow peas, water chestnuts & bamboo shoots.)		
13. Chicken with Cashew Nuts	6.75
(Diced chicken with cashew nuts, water chestnuts and mushrooms in brown sauce.)		
14. Sweet & Sour Chicken	6.75
15. Chicken with Fresh Broccoli	6.95
(White meat chicken sautéed with fresh broccoli in white sauce.)		
16. Diced Chicken with Black Bean Sauce	6.75
(Diced chicken with green pepper and onion in Chinese bean sauce.)		
★17. Chicken with Garlic Sauce	6.75
(Diced chicken stir-fried with bamboo shoot, water chestnuts, green pepper and carrot in garlic sauce.)		
18. Curry Chicken	6.95
★19. Szechwan Chicken	6.95
20. Mongolian Chicken	6.95

Lista - podstawowe operacje



- Listę tworzymy podając kolejne jej elementy rozdzielone przecinkami i ujęte w kwadratowe klamry.
- Elementy listy mogą być różnego typu.
- Aby dodać element na koniec listy, używamy metody **append**, podając jej jako parametr element, który chcemy wstawić.
- Aby usunąć element używamy metody **pop**, która nie tylko usunie go z list ale też zwróci usunięty element jako wynik działania funkcji.

```
In [1]: my_list = [1, "ala", True]
```

```
In [2]: my_list.append(2)
```

```
In [3]: print(my_list)
[1, 'ala', True, 2]
```

```
In [4]: my_list.pop()
Out[4]: 2
```

```
In [5]: print(my_list)
[1, 'ala', True]
```

Lista - podstawowe operacje cd.



- Do każdego elementu listy można dostać się po jego indeksie (numerze porządkowym) za pomocą operatora `[]`.
- W Pythonie i większości języków programowania indeksy zaczynają się od zera, a więc `lista[0]` oznacza dostęp do pierwszego elementu zmiennej lista.
- Ujemny indeks oznacza dostęp od końca, np `-1` to indeks ostatniego elementu a `-2` przedostatniego.
- Można łatwo zmodyfikować zawartość listy pod danym indeksem poprzez przypisanie.

```
In [1]: list_1 = [1,2,3,4,5,6,7]
```

```
In [2]: print(list_1[0])  
1
```

```
In [3]: print(list_1[-1])  
7
```

```
In [4]: list_1[3] = "ala ma kota"
```

```
In [5]: print(list_1)  
[1, 2, 3, 'ala ma kota', 5, 6, 7]
```



Szatkownie listy - slicing.

- W kwadratowe nawiasy listy możemy wstawić nie tylko indeks.
- Dwie liczby oddzielone dwukropkiem oznaczają zakres zaczynający się od pierwszej liczby i ciągnący się aż do drugiej (ale bez niej).
- Trzy liczby oznaczają zakres i krok. Jeśli krok jest ujemny to zakres będzie odwrócony.
- Jeśli nie poda się jakieś z trzech liczb przyjmą one domyślną wartość. Dla pierwszej liczby jest to początek listy, dla drugiej jej koniec. Domyślny krok to 1.

```
In [1]: list_1 = [1,2,3,4,5,6,7]
In [2]: list_1[2:6]
Out[2]: [3, 4, 5, 6]

In [3]: list_1[2:6:2]
Out[3]: [3, 5]

In [4]: list_1[::-1]
Out[4]: [7, 6, 5, 4, 3, 2, 1]

In [5]: list_1[::]
Out[5]: [1, 2, 3, 4, 5, 6, 7]

In [6]: list_1[1:-1]
Out[6]: [2, 3, 4, 5, 6]

In [7]: list_1[1:]
Out[7]: [2, 3, 4, 5, 6, 7]

In [8]: list_1[:-1]
Out[8]: [1, 2, 3, 4, 5, 6]
```

Łączenie list.



- Jeśli chcemy uzyskać nową listę, która jest sumą elementów z już istniejących list, możemy po prostu użyć dodawania.
- Jeśli mamy już listę i chcemy do niej dopisać elementy z innej listy musimy użyć metody **extend**. Wywołuje się ją na docelowej liście a jako parametr podaje listę, którą chcemy dopisać.

```
In [1]: first_list = [1,2,3]
In [2]: second_list = ['a', 'b', 'c']
In [3]: list_sum = first_list + second_list
In [4]: print(list_sum)
[1, 2, 3, 'a', 'b', 'c']
In [5]: target_list = ['ala', 'ma', 'kota']
In [6]: source_list = ['ola', 'ma', 'psa']
In [7]: target_list.extend(source_list)
In [8]: print(target_list)
['ala', 'ma', 'kota', 'ola', 'ma', 'psa']
```

Modyfikacja elementów ze środka listy.



- Do tej pory poznaliśmy jedynie metody **append**, **extend** i **pop**, pozwalające dodać element bądź inną listę na koniec innej listy albo usunąć element z końca listy.
- Aby usunąć element ze środka listy należy użyć słowa kluczowego **del** i podając jego indeks w nawiasie kwadratowym.
- Aby dodać element do początku listy trzeba użyć operatora dodawania.
- Aby dodać element do środka listy trzeba użyć operatora zakresów.
- Te operacje są niewygodne ponieważ korzystanie z nich nie jest zalecane.

```
In [1]: a_list = [1,2,3,4,5,6,7]
In [2]: del a_list[5]
In [3]: print(a_list)
[1, 2, 3, 4, 5, 7]
In [4]: a_list = [0] + a_list
In [5]: print(a_list)
[0, 1, 2, 3, 4, 5, 7]
In [6]: a_list = a_list[:-1] + [6] + a_list[-1:]
In [7]: print(a_list)
[0, 1, 2, 3, 4, 5, 6, 7]
```

Wbudowane funkcje operujące na listach.



- Python dostarcza zestaw standardowych funkcji, z których niektóre operują na listach.
- Funkcja **min** zwraca najmniejszy element listy, **max** największy a **sum** sumę elementów (jeśli można je dodać).
- Funkcja **len** zwraca długość listy.
- Funkcja **reversed** pozwala odwrócić listę.
- Funkcja **sorted** umożliwia posortowanie listy ale podobnej funkcjonalności dostarcza również metoda **sort** listy. Różnica polega na tym, że **sort** sortuje listę w miejscu a **sorted** tworzy kopię listy, w której elementy są już posortowane (tak naprawdę zwraca generator ale o nich porozmawiamy później).

```
In [1]: a_list = [8, 3, 5, 7, 2, 1]
In [2]: max(a_list)
Out[2]: 8

In [3]: min(a_list)
Out[3]: 1

In [4]: sum(a_list)
Out[4]: 26

In [5]: len(a_list)
Out[5]: 6

In [6]: list(sorted(a_list))
Out[6]: [1, 2, 3, 5, 7, 8]

In [7]: list(reversed(a_list))
Out[7]: [8, 7, 5, 3, 2, 1]

In [8]: a_list.sort()

In [9]: print(a_list)
[1, 2, 3, 5, 7, 8]
```



zip: funkcja - suwak

- **zip** jest kolejną wbudowaną funkcją.
- Na wejście przyjmuje dwie lub więcej listy.
- Zwraca sekwencję krotek gdzie każdy element krotki pochodzi z jednej z list.
- Na przykład dla list ['a', 'b', 'c'] oraz [1, 2, 3] funkcja zip zwróci [('a', 1), ('b', 2), ('c', 3)].
- Jeśli funkcje mają różne długości wynik będzie miał długość najkrótszej z list.

```
In [1]: shopping_items = ['eggs', 'ham', 'cheese']

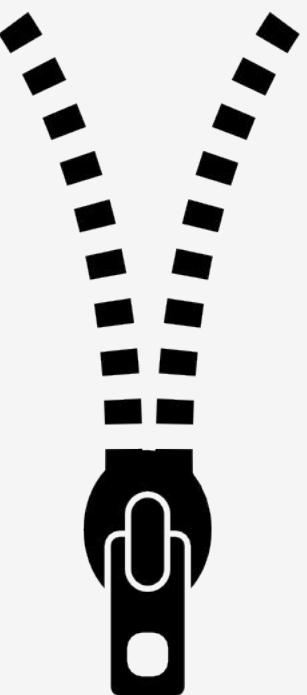
In [2]: quantities = [4, 2, 3]

In [3]: list(zip(shopping_items, quantities))
Out[3]: [('eggs', 4), ('ham', 2), ('cheese', 3)]

In [4]: longer_list = ['red', 'hot', 'chili', 'peppers']

In [5]: shorter_list = ['czewone', 'gorace', 'czili']

In [6]: list(zip(longer_list, shorter_list))
Out[6]: [('red', 'czewone'), ('hot', 'gorace'), ('chili', 'czili')]
```





Funkcja range - zakres

- 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.

```
In [1]: list(range(10))
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: list(range(1, 11))
Out[2]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [3]: list(range(0, 30, 5))
Out[3]: [0, 5, 10, 15, 20, 25]

In [4]: list(range(0, 10, 3))
Out[4]: [0, 3, 6, 9]

In [5]: list(range(0, -10, -1))
Out[5]: [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]

In [6]: list(range(0))
Out[6]: []
```

Pętla for



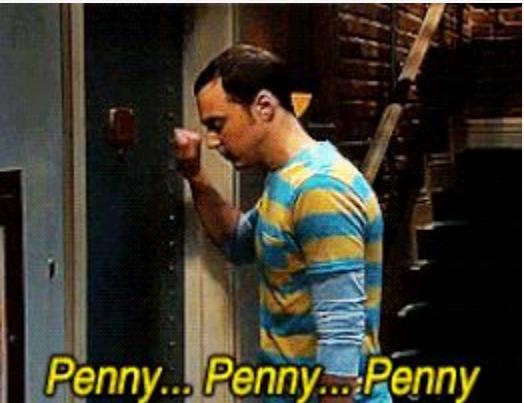
- Pętla **for** służy między innymi do wykonania jakiejś operacji na każdym elemencie listy.
- Za pomocą pętli for możemy np. zaimplementować funkcję znajdująca największy element z listy dodatnich liczb całkowitych (czyli taką naszą prymitywną wersję funkcji **max**).
- Na początku przypisujemy do zmiennej **largest** najmniejszy możliwy wynik czyli 0.
- Dla każdego elementu listy sprawdzamy czy nie jest on większy niż obecny największy element i jeśli tak to aktualizujemy wartość zmiennej **largest**.

```
1  def my_max(sequence):  
2      largest = 0  
3      for element in sequence:  
4          if element > largest:  
5              largest = element  
6      return largest
```



Pętla for

- Pętla **for** może również służyć do powtórzenia jakiejś czynności określona liczbę razy.
- Możemy w ten sposób np. zasymulować zachowanie Sheldona Coopera z Teorii Wielkiego Wybuchu, który zawsze puka trzy razy.



```
In [1]: def sheldon_knock(name):  
    ...:  
    ...:  
    ...:  
    ...:
```

```
In [2]: sheldon_knock('Penny')  
Penny!  
Penny!  
Penny!
```

Pętla while



- Pętla **while** działa tak długo jak długo spełniony jest logiczny warunek podany w jej definicji.
- Zaimplementujmy odliczanie oparte na pętli **while**.
- Odliczamy w dół od zadanej liczby aż do zera.



```
In [1]: def count_down(number):
...:     while number:
...:         print(number)
...:         number -= 1
...:         print('Lift off!')
...:

In [2]: count_down(10)
10
9
8
7
6
5
4
3
2
1
Lift off!
```

Pętla while



- Za pomocą pętli while możemy zaimplementować wspomniany wcześniej problem Collatza, tym razem bez rekurencji.
- Dopóki liczba nie jest jedynką, podziel ją przez dwa lub pomnóż przez 3 i dodawaj jeden w zależności od parzystości.
- Niektórzy uważają, że programy napisane bez użycia rekurencji są bardziej czytelne i łatwiej je zrozumieć.

```
1  def collatz(number):
2      while number != 1:
3          print(number)
4          if number % 2:
5              number = number * 3 + 1
6          else:
7              number = number // 2
8
9  ► if __name__ == '__main__':
10     my_number = int(input('Give a number: '))
11     print(f'Collatz sequence for number {my_number} is:')
12     collatz(my_number)
```

```
[ MacBook-Pro-Micha: ~ /cli ]$ python collatz.py
Give a number: 11
Collatz sequence for number 11 is:
11
34
17
52
26
13
40
20
10
5
16
8
4
2
```

Pętle: słowa kluczowe break i continue.



- Dwa słowa kluczowe w istotny sposób wpływają na zachowanie pętli.
- Słowo kluczowe **break** przerywa wykonanie pętli, nawet jeśli warunek pętli jest spełniony.
- Słowo kluczowe **continue** przerywa bieżącą iterację i przechodzi do następnego obiegu pętli.

```
In [1]: def premature_lift_off(number):
...:     while number:
...:         print(number)
...:         number -= 1
...:         if number == 5:
...:             break
...:     print("Lift off!")

In [2]: premature_lift_off(10)
10
9
8
7
6
Lift off!
```

```
In [1]: def even_count_down(number):
...:     while number:
...:         number -= 1
...:         if number % 2:
...:             continue
...:         print(number)

In [2]: even_count_down(10)
8
6
4
2
0
```

Czas na program: ustalanie ojcostwa



- 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!**



Czas na program: ustalanie ojcostwa



- 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.

```
1st suspect DNA: GAGCCTACTAACGGGAT
2nd suspect DNA: GAGCCTACTAACAAAAT
Child DNA: CATCGTAATGACGGCCT
Suspect #1 is a father.
```

```
hamming.py x
1 def hamming(strand_a, strand_b):
2     result = 0
3     zipped_strands = zip(strand_a, strand_b)
4     for pair in zipped_strands:
5         if pair[0] != pair[1]:
6             result+=1
7     return result
8
9
10 if __name__ == '__main__':
11     suspect_1 = input('1st suspect DNA: ')
12     suspect_2 = input('2nd suspect DNA: ')
13     child = input('Child DNA: ')
14     dist_1 = hamming(suspect_1, child)
15     dist_2 = hamming(suspect_2, child)
16     if dist_1 < dist_2:
17         print('Suspect #1 is a father.')
18     else:
19         print('Suspect #2 is a father.'
```

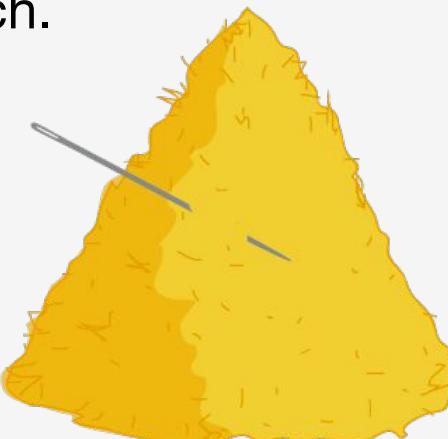


Autor: Michał Nowotka

Wyszukiwanie elementu w liście



- Aby sprawdzić czy dany element znajduje się na liście można użyć operatora **in**.
- Czasami odpowiedź na pytanie “czy element należy do listy” do za mało i potrzebujemy znać dokładną pozycję (indeks) elementu na liście. W takim wypadku należy wywołać na liście metodę **index**, która jako jedyny argument przyjmuje element, którego indeks na liście chcemy poznać.
- Jeśli na liście znajduje się kilka poszukiwanych elementów, metoda **index** zwróci pozycję pierwszego z nich.



```
In [1]: haystack = [1,2,3,4]
In [2]: needle = 2
In [3]: needle in haystack
Out[3]: True

In [4]: haystack.index(needle)
Out[4]: 1

In [5]: needle = 0
In [6]: needle in haystack
Out[6]: False

In [7]: haystack = [1,0,0,0]
In [8]: needle in haystack
Out[8]: True

In [9]: haystack.index(needle)
Out[9]: 1
```

Zbiór - set



- Kolejnym po liście złożonym typem danych w języku Python jest zbiór (**set**).
- Zbiór można sobie wyobrażać jako worek. Kiedy wkładasz rękę do worka nigdy nie wiesz co z niego wyciągniesz. Tak samo jest ze zbiorami - w przeciwieństwie do list kolejność elementów w zbiorze nie jest ustalona (ani istotna).
- Kolejną różnicą jest to, że elementy zbioru nigdy się nie powtarzają - wszystkie elementy są unikalne.
- Zbór tworzymy podając jego elementy rozdzielone przecinkami w nawiasach klamrowych.
- Można też przekształcić listę w zbiór i ponownie zbiór na listę. Jest to najprostszy sposób na usunięcie duplikatów z listy.



```
In [1]: my_set = {1,2,3,4}
In [2]: my_set
Out[2]: {1, 2, 3, 4}

In [3]: my_set = {1,2,3,4,4,4}
In [4]: my_set
Out[4]: {1, 2, 3, 4}

In [5]: list_with_duplicates = [1, 2, 3, 1, 2, 3]
In [6]: list_without_duplicates = list(set(list_with_duplicates))

In [7]: list_without_duplicates
Out[7]: [1, 2, 3]
```

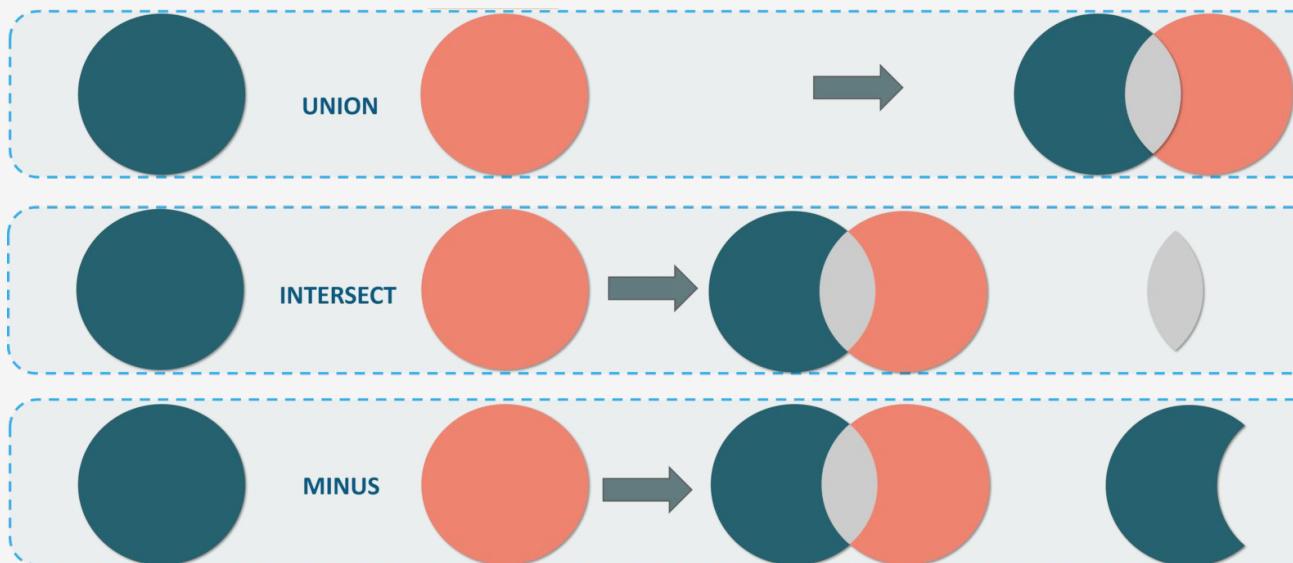
Autor: Michał Nowotka

Prawa do korzystania z materiałów posiada Software Development Academy

Podstawowe operacje na zbiorach



- W szkole poznaliśmy podstawowe operacje na zbiorach:
 - suma (unia)
 - iloczyn (przecięcie, część wspólna)
 - różnica (również różnica symetryczna)
- Python umożliwia przeprowadzenie na zbiorach tych samych operacji (a nawet więcej).



```
In [1]: set_one = {1, 2, 3, 4, 5}  
In [2]: set_two = {4, 5, 6, 7, 8}  
In [3]: set_one | set_two  
Out[3]: {1, 2, 3, 4, 5, 6, 7, 8}  
In [4]: set_one & set_two  
Out[4]: {4, 5}  
In [5]: set_one - set_two  
Out[5]: {1, 2, 3}  
In [6]: set_two - set_one  
Out[6]: {6, 7, 8}  
In [7]: set_one ^ set_two  
Out[7]: {1, 2, 3, 6, 7, 8}
```

Modyfikacje zbioru, funkcje wbudowane



- Metoda **add** umożliwia dodanie nowego elementu do zbioru.
- Metoda **update** pozwala na dodanie do zbioru elementów z innego zbioru.
- Metoda **remove** usuwa element ze zbioru.
- Metoda **clear** usuwa wszystkie elementy ze zbioru.
- Wbudowane funkcje **len**, **min**, **max**, **sum**, które poznaliśmy na przykładzie listy działają również w kontekście zbiorów.
- Pustego zbioru nie da się stworzyć za pomocą literatuła **{ }** ponieważ ten jest zarezerwowany dla pustego słownika. Aby stworzyć pusty zbiór trzeba użyć funkcji **set()**.

```
In [1]: a_set = set()
In [2]: a_set.add(5)
In [3]: a_set.add(5)
In [4]: a_set
Out[4]: {5}

In [5]: a_set.update({1,2,3})
In [6]: a_set
Out[6]: {1, 2, 3, 5}

In [7]: len(a_set)
Out[7]: 4

In [8]: max(a_set)
Out[8]: 5

In [9]: min(a_set)
Out[9]: 1

In [10]: sum(a_set)
Out[10]: 11
```

Przynależność do zbioru, podzbiór, nadzbiór



- Podobnie jak w przypadku listy operator **in** służy do sprawdzenia czy element należy do zbioru.
- Jednak w przypadku zbioru wynik operacji jest znacznie szybszy ponieważ zbiór jest zoptymalizowany pod kątem operacji przynależności.
- Ponieważ zbiór nie jest uporządkowany i nie można odnosić się do jego elementów po indeksie nie ma metody **index**.
- Metoda **issubset** sprawdza czy zbiór, na którym wywoływana jest metoda, jest podzbiorem zbioru podanego jako argument.
- Metoda **issuperset** sprawdza czy zbiór, na którym wywoywana jest metoda, jest nadzbiorem zbioru podanego jako argument.

```
In [1]: a_set = {1, 2, 3, 4}

In [2]: 5 in a_set
Out[2]: False

In [3]: 2 in a_set
Out[3]: True

In [4]: a_set.issubset({1, 2, 3, 4, 5})
Out[4]: True

In [5]: a_set.issuperset({1, 2})
Out[5]: True

In [6]: a_set.issuperset({1, 2, 3, 4, 5})
Out[6]: False

In [7]: a_set.issubset({1, 2})
Out[7]: False
```

Pora na program - izogramy



- Izogram jest słowem w którym żadna litera nie powtarza się.
- Przykładem jest np. słowo “skrzynia”.
- Należy napisać program który poprosi użytkownika o podanie słowa i napisze czy dane słowo jest izogramem czy nie.

```
1  def is_isogram(word):
2      letters = set()
3      for letter in word.lower():
4          if letter in letters:
5              return False
6          letters.add(letter)
7      return True
8
9
10 if __name__ == '__main__':
11     while True:
12         my_word = input('Give a word: ').strip()
13         answer = 'is' if is_isogram(my_word) else 'is not'
14         print(f'Word {my_word} {answer} an isogram\n')
15         shall_continue = input('Do you want to continue ([y]/n)? ')
16         if shall_continue.lower() != 'y':
17             break
```

```
Give a word: skrzynia
Word skrzynia is an isogram

Do you want to continue ([y]/n)?: y
Give a word: Ala
Word Ala is not an isogram

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

Process finished with exit code 0
```

Pora na program - izogramy



- Tworzymy zbiór liter o nazwie **letters**.
- Na początku zbiór jest pusty.
- Bierzemy nasze słowo i przekształcamy każdą literkę na małą.
- W pętli **for** przechodzimy po każdej literce w słowie.
- Dla każdej litery sprawdzamy czy jest już w naszym zbiorze.
- Jeśli jest to słowo na pewno nie jest izogramem, więc możemy natychmiast zakończyć działanie funkcji w wynikiem **False**.
- W przeciwnym wypadku litery jeszcze nie było więc dodajemy ją do naszego zbioru.
- Można napisać to prościej!

```
1  def is_isogram(word):  
2      letters = set()  
3      for letter in word.lower():  
4          if letter in letters:  
5              return False  
6          letters.add(letter)  
7      return True
```

Albo prościej:

```
1  def is_isogram(word):  
2      return len(word) == len(set(word))
```

Pora na program - izogramy



```
5 ► if __name__ == '__main__':
6     while True:
7         my_word = input('Give a word: ').strip()
8         answer = 'is' if is_isogram(my_word) else 'is not'
9         print(f'Word {my_word} {answer} an isogram\n')
10        shall_continue = input('Do you want to continue ([y]/n)? ')
11        if shall_continue.lower() != 'y':
12            break
```

- Rozpocznij nieskończoną pętlę, z której można wyrwać się jedynie słowem kluczowym **break**.
- Pobierz od użytkownika słowo, zignoruj białe znaki z początku lub końca.
- Twórz odpowiedź na podstawie wyniku funkcji **is_isogram**.
- Wypisz sformatowaną odpowiedź.
- Zapytaj użytkownika czy chce dalej się bawić.
- Jeśli nie, przerwij pętlę.

Słownik - dict



- Ta struktura danych jest podobna do książki telefonicznej lub encyklopedii.
- Zaglądamy pod pewien klucz (nazwisko abonenta, hasło encykopedyczne) 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 a żadną wartością a słownik to robi.

Phone Book	
273-02399	Taylor, Ed R
623-1262	Taylorville, 2 Littleton Rd Chel.
511-2254	Taylorville, 2 Littleton Rd Chel.
3-8650	Taylorville, 2 Littleton Rd Chel.
3-8625	Taylorville, 2 Littleton Rd Chel.
3-8203	Taylorville, 2 Littleton Rd Chel.
-040	Taylorville, 2 Littleton Rd Chel.
-0550	Taylorville, 2 Littleton Rd Chel.
7-2215	Tewkesbury Dale & Chase 12 Somers Ave Wsd.
0-0501	TFC Studio 142 Somers Ave Wsd.
3856	TFD Management Co 148 Somers Ave Wsd.
8082	148 Somers Ave Wsd.
9200	Thai Jasmine 313 Littleton Rd Chel.
6229	Thai K 12 Castle Ln Westford.
200	Thairu Hannah 24 Castle Rd Chel.
015	Thakkar Paresh & Parul 39 Arlington Ave.
200	Thakrar Bhavna & Kiran 47 Russell Way Wsd.
57	Vijay & Gwyn 10 Lady Slipper Ln Atn.
14	Thaller Kurt & Carol 10 Knowiton Dr Atn.
84	Thandapani Suresh 8 Meyer Hill Dr Atn.
58	Tharp Lisa Kauffman 10 Dunham Ln Acton.
56	Thatcher Bruce M 5 Till Dr Atn.....
19	Patricia 23 Turtle Dr Acton.....
7	Thatcher Photography 5 Till Dr Atn.....
2	Thatcher R 102 Willow W Atn.....
1	Thaure Frederic & Lisa 8 Heather Hill Rd Atn.....
1	Thayer Kent & Astrid 45 Parkerville Rd Chel.....
1	Wm H Jr 47 North Rd Chel.....
1	Wm & Teresa 46 Sylvan Av Chel.....
	The Courtyard Condominium
	360 Littleton Rd Chel.....
	The Family Eye Care Center 133 Littleton Rd Wsd..
	The Gill Group 1 Courthouse Ln Chel.....
	The Gutierrez Co 2 Technology Pk Dr Wsd.....
	The Mill 481 Great Rd Atn.....
	The Movement Center Dance Studio
	423 Great Rd Atn.....
	The Pizza Place 210 Boston Rd Chel.....
	The Stone Yard 2 Spectacle Pond Rd Lttn.....
	The Total You 326 Great Rd Lttn.....
	The Whole Body Studio 187 Littleton Rd Wsd.....
	Theall Frank 20 Sheila Av Chel.....
	Theatre III Box Office 250 Central Atn.....
	Theide D 89 West St Wsd.....
	David 89 West St Wsd.....

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.

```
In [1]: my_phonebook = {'police': 997, 'pizza': 467, 'emergency': 112}

In [2]: my_phonebook['pizza']
Out[2]: 467

In [3]: my_phonebook['girlfriend'] = 'Forever alone :P'

In [4]: my_phonebook
Out[4]:
{'police': 997,
 'pizza': 467,
 'emergency': 112,
 'girlfriend': 'Forever alone :P'}

In [5]: del my_phonebook['girlfriend']

In [6]: my_phonebook
Out[6]: {'police': 997, 'pizza': 467, 'emergency': 112}

In [7]: my_phonebook['pizza'] = 12321

In [8]: my_phonebook
Out[8]: {'police': 997, 'pizza': 12321, 'emergency': 112}
```

Słownik - zaglądanie pod klucz



- 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 **{ }**.

```
In [1]: my_dict = {}

In [2]: my_dict['invalid_key']
-----
KeyError
<ipython-input-2-ecfb297c56ea> in <module>
----> 1 my_dict['invalid_key']

KeyError: 'invalid_key'

In [3]: some_val = my_dict.get('invalid_key')

In [4]: print(some_val)
None

In [5]: some_val = my_dict.get('invalid_key', 'default value')

In [6]: print(some_val)
default value

In [7]: my_dict['some_key'] = 'foo'

In [8]: some_val = my_dict['some_key']

In [9]: print(some_val)
foo

In [10]: some_val = my_dict.get('some_key')

In [11]: print(some_val)
foo
```

Słownik - zbiory kluczy, wartości, par



- 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.

```
In [1]: my_phonebook = {'police': 997, 'pizza': 467, 'emergency': 112}

In [2]: list(my_phonebook.keys())
Out[2]: ['police', 'pizza', 'emergency']

In [3]: list(my_phonebook.values())
Out[3]: [997, 467, 112]

In [4]: list(my_phonebook.items())
Out[4]: [('police', 997), ('pizza', 467), ('emergency', 112)]

In [5]: for key in my_phonebook:
...:     print(key)
...:
police
pizza
emergency

In [6]: 'police' in my_phonebook
Out[6]: True

In [7]: 'girlfriend' in my_phonebook
Out[7]: False
```

Pora na zadanie - wyniki w Scrabble



- 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).

```
Give a word: cabbage
Word cabbage is worth 14 in Scrabble
Do you want to continue ([y]/[n])?: y
Give a word: garncarski
Word garncarski is worth 17 in Scrabble
Do you want to continue ([y]/[n])?: n
```

```
1  scores = {'a': 1, 'e': 1, 'i': 1, 'o': 1, 'u': 1, 'l': 1, 'n': 1, 'r': 1,
2      's': 1, 't': 1, 'd': 2, 'g': 2, 'b': 3, 'c': 3, 'm': 3, 'p': 3,
3      'f': 4, 'h': 4, 'v': 4, 'w': 4, 'y': 4, 'k': 5, 'j': 8, 'x': 8,
4      'q': 10, 'z': 10}
5
6
7  def scrabble_score(word):
8      total_score = 0
9      for letter in word.lower():
10          total_score += scores[letter]
11
12
13
14  if __name__ == '__main__':
15      while True:
16          my_word = input('Give a word: ').strip()
17          score = scrabble_score(my_word)
18          print(f'Word {my_word} is worth {score} in Scrabble')
19          shall_continue = input('Do you want to continue ([y]/[n])?: ')
20
21          if shall_continue.lower() != 'y':
22              break
```

List comprehension - wyrażenia 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.

```
In [1]: old_list = [1, 2, 3, 4, 5, 6, 7]
```

```
In [2]: new_list = [x ** 2 for x in old_list if not x % 2]
```

```
In [3]: print(new_list)
[4, 16, 36]
```

List comprehension - wyrażenia listowe



- Używając wyrażenia listowego możemy usunąć pętlę for z funkcji liczącej wynik w Scrabble dla danego słowa.
- Teraz cała nasza funkcja składa się tylko z jednej linijki kodu i jest bardzo czytelna ponieważ mówi **CO** ma być zrobione (programowanie deklaratywne) a nie **JAK** (programowanie imperatywne).

```
def scrabble_score(word):  
    return sum([scores[letter] for letter in word])
```

List comprehension - wyrażenia listowe



- W ten sam sposób możemy przekształcić funkcję **hamming** z programu do ustalania ojcostwa.
- Wyrażenia listowe to potężne narzędzie! Sprawiają że bardzo krótkie, lakoniczne polecenie robi coś bardzo złożonego.
- Jest to obosieczny miecz, ponieważ bardzo trudno jest zrozumieć złożone wyrażenie listowe, szczególnie jeśli napisał je ktoś inny niż my albo napisaliśmy je dawno temu i już nie pamiętamy co oznacza.
- Zachowanie odpowiedniego balansu pomiędzy stosowaniem pętli a wyrażeń listowych jest sztuką, której musi nauczyć się każdy dojrzały Pythonista.

```
def hamming(strand_a, strand_b):  
    return sum([1 for a, b in zip(strand_a, strand_b) if a == b])
```

Dict comprehension - wyrażenia słownikowe



- Analogicznie do wyrażeń listowych możemy tworzyć wyrażenia słownikowe, które nowy słownik utworzą na podstawie pewnej sekwencji.
- Klasycznym przykładem wyrażenia słownikowego jest stworzenie słownika w którym klucze i wartości będą zamienione miejscami względem oryginalnego słownika.

```
In [1]: old_dict = {'Policja': 997, 'Pizza': 383, 'Mama': 593}

In [2]: new_dict = {v: k for k, v in old_dict.items()}

In [3]: print(new_dict)
{997: 'Policja', 383: 'Pizza', 593: 'Mama'}
```

Set comprehension - wyrażenia zbiorowe



- Do kompletu zostały nam już tylko wyrażenia nad zbiorami.
- Tworzy się je tak samo jak wyrażenia listowe, zastępując jedynie nawiasy kwadratowe klamrowymi.
- Wyobraźmy sobie że chcemy poznać wszystkie długości wyrazów na liście.
- Co się stanie jeśli klamry zamienimy na nawiasy kwadratowe?

```
In [1]: word_list = ['ala', 'ola', 'alek', 'mirek', 'ela']

In [2]: word_lengths = {len(word) for word in word_list}

In [3]: print(word_lengths)
{3, 4, 5}
```