

The Python logo, consisting of two interlocking snakes, one blue and one yellow, is centered in the background. The word "wprowadzenie" is written in white, sans-serif font across the middle of the logo.

wprowadzenie




python



Python jest językiem **interpretowanym** (przeciwieństwo to język kompilowany) - kod jest wykonywany od góry do dołu i wynik jest zwracany natychmiastowo

Python to język dynamicznie typowany, czyli nie ma potrzeby ustawiania typu dla zmiennych

dlaczego warto uczyć się pythona

-  python został stworzony tak, aby był łatwy do zrozumienia i przyjazny dla początkujących
-  kod Pythona można czytać jak zwykły tekst po angielsku. Jego składnia jest przejrzysta i zwięzła, jednak duże znaczenie ma formatowanie kodu i białe znaki
-  python obecnie króluje w dziedzinach jak *data science* czy *machine learning*. Może być także używany w Raspberry Pi

statystyki

dane na dzień 29.01.2021

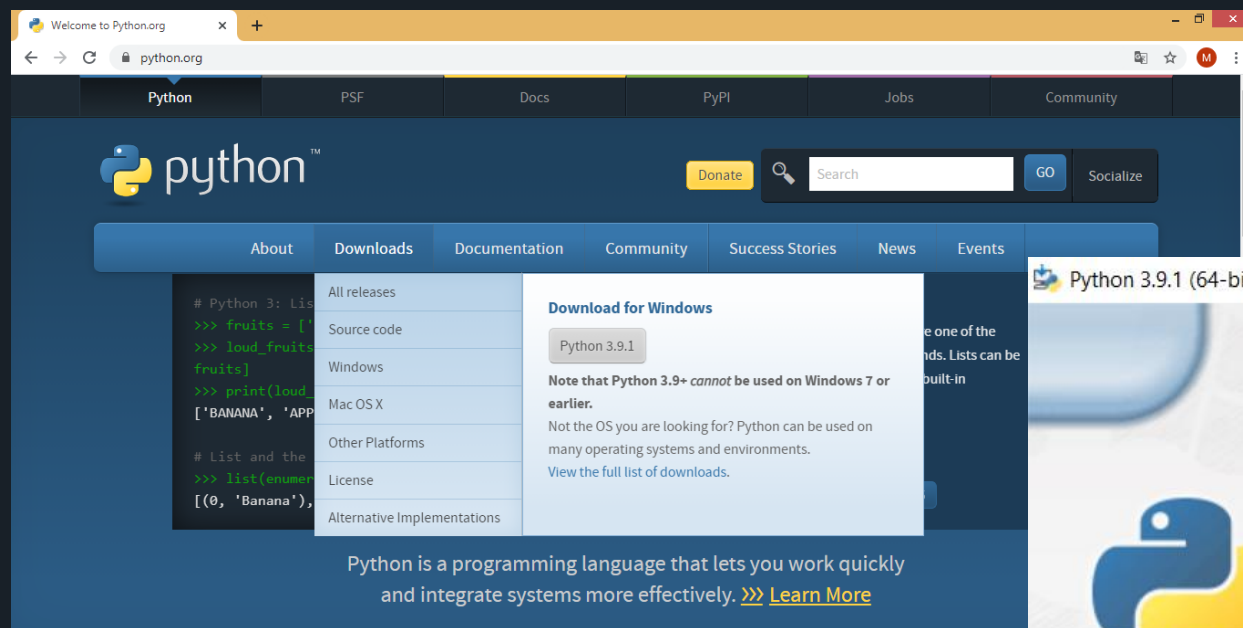


Na [Stack Overflow](#) jest 3 najczęściej tagowanym językiem z 1 639 466 zapytaniami



Na [Github](#) można znaleźć 1 821 506 repozytoriów

instalacja pythona



dodawanie Pythona do zmiennych środowiskowych

import this

filozofia społeczności Pythona jest zawarta w **Zen Pythona** opracowanym przez Tima Petersa.

dostęp do krótkiego zbioru reguł dotyczących tworzenia dobrego kodu w Pythonie można uzyskać używając w interpreterze polecenia **import this**

Beautiful is better than ugly.

#Piękno jest lepsze od brzydoty.

Simple is better than complex.

#Prostota jest lepsza od zawiłości.

Readability counts.

#Przejrzystość ma znaczenie.

Now is better than never.

#Teraz jest lepsze od nigdy.

The Python logo, consisting of two interlocking snakes, one blue and one yellow, is centered in the background. Overlaid on the logo is the text "Hello world!" in a white, sans-serif font.

Hello world!

Hello world!

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

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

wynik działania:

Hello world!

Hello world!

The Python logo, consisting of two interlocking snakes, one blue and one yellow, is centered in the background. Overlaid on the logo is the text "zmienne i typy danych" in a white, sans-serif font.

zmienne i typy danych

jak prawidłowo nazywać zmienne

zmienne mogą składać się z liter,
podkreślnika oraz cyfr

nazwy składające się z dwóch lub więcej
wyrazów powinny być oddzielone
podkreślinkiem - notacja snake_case.
Używanie spacji jest niedozwolone

wielkość liter ma znaczenie

cyfra nie może być pierwszym znakiem
w nazwie

ciągi tekstowe

w Pythonie typem dla ciągów tekstowych jest **str**

```
str1 = "to jest ciąg tekstowy"
```

```
str2 = 'to również jest ciąg tekstowy'
```

```
nested = 'taka elastyczność pozwala na użycie dosłownych "znaków cytowania" w ciągach tekstowych'
```

```
print(nested)
```

ciągi tekstowe – białe znaki

w celu użycia tabulatora należy użyć sekwencji znaków \t

```
print("Python")  
print("\tPython")
```

wynik działania:

```
Python  
    Python
```

aby dodać znaki nowego wiersza należy użyć \n

```
print("Popularne języki programowania:\nPython\nJavaScript")
```

zmienne w ciągach tekstowych

aby wstawić wartość zmiennej do ciągu tekstowego, należy tuż przed znakiem cytowania umieścić literę f

nazwa każdej zmiennej, której wartość ma się znaleźć w ciągu tekstowym, musi być umieszczona w nawiasie klamrowym

```
first_name = "Jan"
last_name = "Kowalski"
full_name = f"{first_name} {last_name}"
print(full_name)
```

liczby całkowite

w Pythonie typem dla liczb całkowitych jest `int`

```
number1 = 10  
number2 = 8  
suma = number1 + number2  
print(suma)  
print(type(suma))
```

wynik działania:

```
18  
<class 'int'>
```

liczby całkowite

próba dodania łańcucha do liczby (i na odwrót) spowoduje błąd!

```
a = 10  
b = "8"  
c = a + b
```

wynik działania:

```
Traceback (most recent call last):  
  File "main.py", line 77, in <module>  
    c = a + b  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

liczby zmiennoprzecinkowe

typ zmiennej dla liczb zmiennoprzecinkowych to `float`

liczba, w której nie została podana część ułamkowa jest traktowana jako liczba całkowita. Ta, która ją posiada jest liczbą zmiennoprzecinkową

```
a = 5
print(type(a))
b = 5.0
print(type(b))
```

wynik działania:

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


liczby zmiennoprzecinkowe

liczba typu `float` jest wynikiem dzielenia dwóch liczb całkowitych

```
number1 = 10  
number2 = 8  
div = number1 / number2  
print(div)  
print(type(div))
```

wynik działania:

```
1.25  
<class 'float'>
```

wiele przypisań

w pojedynczym wierszu można przypisać wartość więcej niż jednej zmiennej

```
x, y, z = 1, 2, 3
```

poszczególne nazwy zmiennych muszą być rozdzielone przecinkami, podobnie jak wartości

o ile liczba wartości odpowiada liczbie zmiennych, Python prawidłowo je do siebie dopasuje

wartości logiczne

typ logiczny `bool` to typ danych, który ma tylko dwie wartości: prawdę (`True`) oraz fałsz (`False`)

```
true = True
print(type(true))
false = False
print(type(false))
```

wynik działania:

```
<class 'bool'>
<class 'bool'>
```

typ logiczny używany jest między innymi po to, aby stwierdzić prawdziwość jakiegoś warunku:

```
print(3 > 5)
```

wynik działania:

```
False
```

wartość None

słowo kluczowe **None** jest używane do definiowania wartości null lub braku wartości

None jest własnym typem danych (NoneType)

```
x = None  
print(type(x))
```

wynik działania:

```
<class 'NoneType'>
```

zmienne stałe

stała przypomina zmienną, której wartość nie ulega zmianie w trakcie całego cyklu życia programu

python nie ma wbudowanego typu przeznaczonego dla stałych

konwencją stosowaną przez programistów jest używanie **tylko wielkich liter** do wskazania zmiennej, która ma być traktowana jak stała i nigdy nie zmieniać wartości

```
MAX_LENGTH = 1000
```

zadania

dane są następujące zmienne:

```
var1 = None
```

```
var2 = 25
```

```
var3 = 25.0
```

```
var4 = True
```

```
var5 = 'True'
```

wydrukuj do konsoli typ każdej zmiennej w osobnej linii

The Python logo, consisting of two interlocking snakes, one blue and one yellow, is centered in the background.

wyświetlanie i wprowadzanie danych

funkcja input()

istnieją dwie podstawowe wersje funkcji `input`, która służy do wczytywania tekstu z klawiatury:

- `input()` – z pustymi nawiasami, pobiera dane wprowadzone przez użytkownika
- `input(tekst)` – przed wczytaniem występuje na ekranie podany tekst zapisany w cudzysłowie

funkcja `input` zwraca typ `str`. Aby pobrać od użytkownika liczbę typu `int` należy wykonać rzutowanie – czyli przed instrukcją `input` wpisać `int`.

```
a = int(input("podaj liczbę: "))
```


funkcja print()

funkcję `print` można wywołać z jednym lub z różną ilością argumentów

- `print(tekst)` – wyświetla tekst podany w cudzysłowie lub zmienną
- `print(tekst1, tekst2, ..., tekstN)` – wyświetla listę tekstów podanych w cudzysłowie lub zmiennych

inne sposoby wywołania funkcji `print`

```
print("Czy Ty masz", 18, "lat?")
```

```
name = "Zosia"  
print("Nazywam się " + name)
```

funkcja print()

za pomocą znaku + nie można połączyć dwóch zmiennych różnych typów

```
year = 2000
```

```
print("urodziłem się w roku" + year)
```

wynik działania:

```
Traceback (most recent call last):
```

```
File "main.py", line 66, in <module>
```

```
    print("urodziłem się w roku" + year)
```

```
TypeError: can only concatenate str (not "int") to str
```

funkcja print()

aby wyświetlić zmienne różnych typów należy oddzielić je przecinkiem

```
year = 2000
```

```
print("urodziłem się w roku", year)
```

wynik działania:

```
urodziłem się w roku 2000
```

zadania

pobierz od użytkownika imię, nazwisko i rok urodzenia w taki sposób, aby wiedział jakie dane podaje. Pobrane dane wypisz do konsoli w jednej instrukcji w taki sposób, aby program policzył wiek użytkownika oraz, aby informacja o jego nazwisku i wieku była wyświetlona w poniższej formie:

```
Użytkownik to: Jan Kowalski. Ma 37 lat.
```

upewnij się, że imię i nazwisko użytkownika będą zawsze rozpoczynały się wielką literą nawet, jeżeli użytkownik wprowadzi dane małymi literami. Użyj w tym celu funkcji `.capitalize()`

zadania - rozwiązanie

```
name = input("podaj imię: ").capitalize()
last_name = input("podaj nazwisko: ").capitalize()
year = int(input("podaj swój rok urodzenia: "))
current_year = 2021

print(f"Użytkownik to: {name} {last_name}. Ma {current_year -
year} lat.")
```



operatory arytmetyczne

dla zmiennych liczbowych typu `int` lub `float` można wykonywać podstawowe działania arytmetyczne

- zmiana znaku: `-a`
- suma: `a + b`
- różnica: `a - b`
- iloczyn: `a * b`
- iloraz: `a / b`
- iloraz całkowity: `a // b`
- reszta z dzielenia całkowitego (modulo): `a % b`
- potęgowanie: `a ** b`

operatory arytmetyczne

```
x = 1
```

```
y = 2
```

```
x += 2
```

```
x = x + y
```

```
y = x + y
```

```
y = x + y
```

```
print(y)
```


działania na łańcuchach

konkatenacja (łączenie zmiennych typu `str`)

```
first_name = "Jan"  
last_name = "Kowalski"  
print(first_name + last_name)
```

wynik działania:

JanKowalski

działania na łańcuchach

in - sprawdzenie czy podany łańcuch zawiera się w drugim łańcuchu. Wynikiem jest True lub False

```
print("Ala" in "Moja siostra Ala")
```

wynik działania:

True

działania na łańcuchach

operator wycinania - `tekst[liczba_początkowa : liczba_końcowa]`

```
name = "Python"
```

```
print(name[1])
```

 – zwraca znak z indeksem 1

```
print(name[-1])
```

 – zwraca ostatni znak

```
print(name[1:4])
```

 – zwraca znaki z przedziału <1; 4)

```
print(name[:])
```

 – zwraca cały łańcuch

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

 – zwraca co drugi znak

```
print(name[1:len(name)])
```

 – zwraca znaki od indeksu pierwszego do ostatniego

działania na łańcuchach

inne działania na łańcuchach

- `len(tekst)` – zwraca długość łańcucha liczoną ze spacjami
- `n*tekst` – powtórzenie tekstu n razy (w miejsce tekst może być wprowadzona zmienna lub łańcuch znaków bezpośrednio w cudzysłowie)
- `tekst_pierwszy not in tekst_drugi` – sprawdza czy tekst_pierwszy **nie jest** fragmentem łańcucha tekst_drugi

operatory porównania

==

(równy z)

porównuje dwie wartości (liczby, stringi, wartości boolean) i sprawdza czy są takie same
wynikiem porównania "3" == 3 jest False
wynikiem porównania 0 == false jest True

>

(większy niż)

sprawdza czy liczba po lewej stronie jest większa niż liczba po prawej stronie
wynikiem porównania 4 > 3 jest True
wynikiem porównania 3 > 4 jest False

<

(mniejszy niż)

sprawdza czy liczba po lewej stronie jest mniejsza niż liczba po prawej stronie
wynikiem porównania 2 < 3 jest True
wynikiem porównania 5 < 4 jest False

!=

(różny)

porównuje dwie wartości (liczby, stringi, wartości boolean) i sprawdza czy są różne
wynikiem porównania 0 != true jest True
wynikiem porównania 3.0 != 3 jest False

>=

(większy niż lub równy)

sprawdza czy liczba po lewej stronie jest większa niż lub równa liczbie po prawej stronie
wynikiem porównania 3 >= 3 jest True
wynikiem porównania 3 >= 4 jest False

<=

(mniejszy niż lub równy)

sprawdza czy liczba po lewej stronie jest mniejsza niż lub równa liczbie po prawej stronie
wynikiem porównania 2 <= 3 jest True
wynikiem porównania 5 <= 4 jest False

operatory porównania

w Pythonie poza liczbami można porównywać także ciągi tekstowe.

„większy” będzie taki tekst, którego pierwszy znak (i ewentualnie kolejne) wystąpią później w porządku alfabetycznym

```
print("Asia" < "Marta")
```

wynik działania:

True

skoro „Asia” jest mniejsze od „Marta” to oznacza, że w kolejności alfabetycznej „Asia” jest wcześniej niż „Marta” – co jest prawdą

operatory porównania

„większy” będzie również taki tekst, który rozpoczyna się małą literą

```
print("Asia" < "asia")
```

wynik działania:

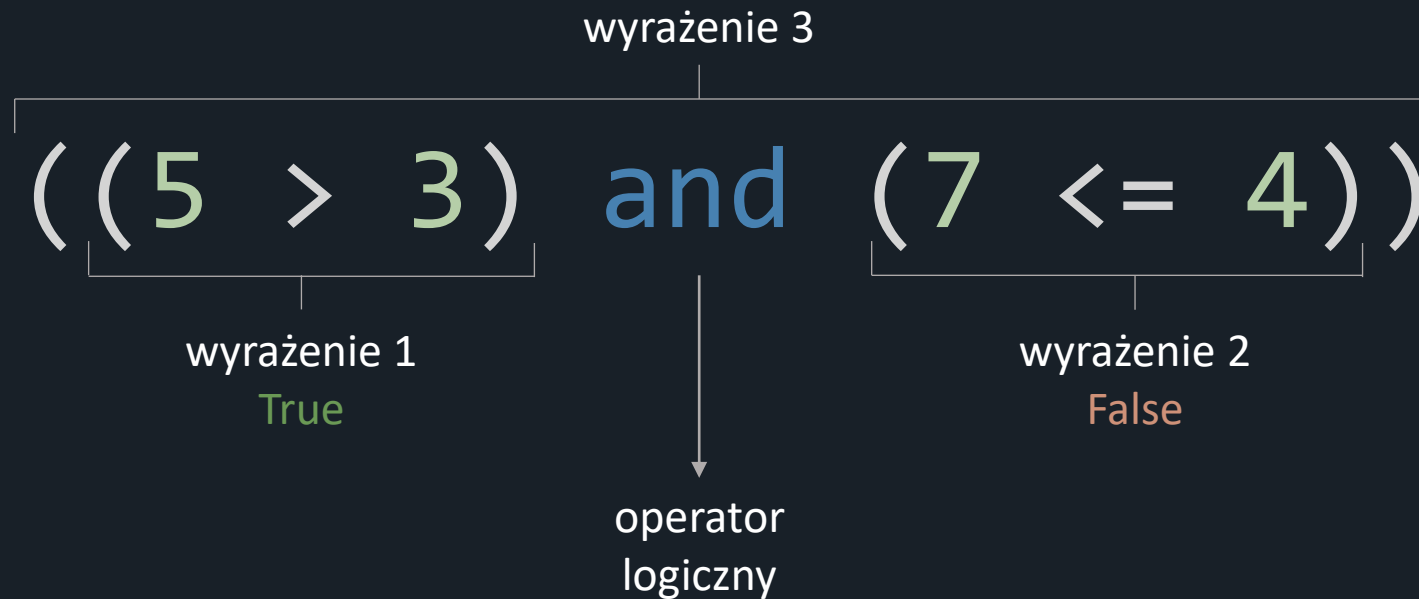
True

wynika to z kolejności występowania w kodach ASCII

<http://www.algorytm.edu.pl/wstp-do-c/ascii.html>

operatory logiczne

operatory logiczne pozwalają na porównywanie wyników działania co najmniej dwóch operatorów porównania



operatory logiczne

w Pythonie pierwszeństwo ma negacja, następnie koniunkcja, a na końcu alternatywa. Aby nadać priorytety działaniom należy użyć nawiasów

negacja (zaprzeczenie) **not**

not

```
not(2 < 1)
```

not True zwraca False

not False zwraca True

koniunkcja (iloczyn) **and**

and

```
(5 > 3)and(7 <= 4)
```

True and True zwraca True

True and False zwraca False

False and True zwraca False

False and False zwraca False

alternatywa (suma) **or**

or

```
(5 > 3)or(7 <= 4)
```

True or True zwraca True

True or False zwraca True

False or True zwraca True

False or False zwraca False

zadania

ocień wartość logiczną poniższych wyrażeń:

- `(x and y) or not z`, dla `x, y, z = True, False, True`
- `x and (y or not z)`, dla `x, y, z = True, False, True`
- `not x or y and x or not z`, dla `x, y, z = True, False, True`
- `(x and z) or (y and z)`, dla `x, y, z = True, False, True`
- `not (x or z) and (y or not z)`, dla `x, y, z = True, False, True`

zadania

co będzie wynikiem poniższych programów?

```
#pierwszy program
```

```
print("Asia" > "asia")
```

```
#drugi program
```

```
print(7 >= 6,99)
```

```
#trzeci program
```

```
x = "Joanna"
```

```
y = "Anna"
```

```
z = y in x
```

```
print(z)
```



listy

mutowalne, uporządkowane struktury danych

tworzenie listy:

- `empty_list = list()` – pusta lista
- `empty_list = []` – pusta lista
- `names = ['Ola', 'Asia', 'Zosia', 'Bartek']`
- `mixed = ["Python", 3.7, 4, True]`
- `nested = [[1, 2, 3], ['a', 'b', 'c']]` – lista zagnieżdżenia

listy

lista jest mutowalna, więc jej elementy można zmieniać

```
names = ['Ola', 'Asia', 'Zosia', 'Bartek']  
names[1] = 'Patryk'  
print(names)
```

wynik działania:

```
['Ola', 'Patryk', 'Zosia', 'Bartek']
```

zagnieżdżanie list

```
first_list = [1, 2, 3]
second_list = [4, 5, 6]
full_list = [first_list, second_list]
print(full_list)
```

wynik działania:

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

łączenie list

```
first_list = [1, 2, 3]  
second_list = [4, 5, 6]  
full_list = first_list + second_list  
print(full_list)
```

wynik działania:

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


operator wycinania

operator wycinania - `lista[liczba_początkowa : liczba_końcowa]`

```
numbers = [1, 5, 17, 25, 98, 150]
```

```
print(numbers[index])
```

 – zwraca pojedynczy element listy

```
print(numbers[-1])
```

 – zwraca ostatni element listy

```
print(numbers[1:4])
```

 – zwraca elementy listy z przedziału <1; 4)

```
print(numbers[:])
```

 – zwraca całą listę

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

 – zwraca co drugi element listy

```
print(numbers[1:len(numbers)])
```

 – zwraca elementy od indeksu pierwszego do ostatniego

```
print(numbers[index:])
```

 – zwraca elementy od podanego indeksu do końca listy

```
print(numbers[:index])
```

 – zwraca elementy od początku listy do podanego indeksu (ale bez niego)

metody używane na listach

<code>append()</code>	dodaje element na końcu listy
-----------------------	-------------------------------

<code>extend()</code>	dodaje jedną listę do innej listy
-----------------------	-----------------------------------

<code>insert()</code>	dodaje element do listy w konkretnym jej miejscu (indeksie)
-----------------------	-------------------------------------------------------------

<code>remove()</code>	usuwa określony element z listy
-----------------------	---------------------------------

<code>clear()</code>	czyści zawartość listy
----------------------	------------------------

<code>pop()</code>	usuwa element z konkretnej pozycji listy
--------------------	------------------------------------------

metody używane na listach

`index()` wyszukuje indeks konkretnego elementu

`count()` zlicza elementy określonego typu na liście

`sort()` sortuje rosnąco elementy listy

`reverse()` zmienia kolejność elementów na liście (od tyłu)

`copy()` kopiuje listę

`print(dir(list))` – podgląd na dostępne metody

funkcje używane na listach

<code>len()</code>	zwraca liczbę elementów dostępnych na liście
--------------------	----------------------------------------------

<code>list()</code>	konwertuje krotkę (tupkę) na listę
---------------------	------------------------------------

<code>max()</code>	zwraca największą wartość na liście
--------------------	-------------------------------------

<code>min()</code>	zwraca najmniejszą wartość na liście
--------------------	--------------------------------------

<code>sum()</code>	zwraca sumę elementów listy (jeżeli lista składa się typów liczbowych)
--------------------	------------------------------------------------------------------------

zadania

do podanej listy `cities = ["Tokio", "Sydney", "Paris"]` dopisz Londyn, aby znalazł się na pozycji z indeksem 2, oraz Tokio na końcu listy

w liście, która powstała sprawdź ile razy występuje miasto "Tokio". Wykorzystaj do tego odpowiednią metodę, a odpowiedź wydrukuj w konsoli tak jak poniżej:

```
Liczba wystąpień Tokio: 2
```

teraz ułóż elementy w kolejności alfabetycznej, a na koniec usuń ostatni element listy (załóż, że nie wiesz ile lista ma elementów) i wydrukuj listę do konsoli.

wynikiem powinna być lista: `['London', 'Paris', 'Sydney', 'Tokio']`

zadania - rozwiązanie

```
cities=["Tokio","Sydney","Paris"]
cities.insert(2, "London")
cities.append("Tokio")

print(f"Liczba wystąpień Tokio: {cities.count('Tokio')}")

cities.sort()
cities.pop(-1)
print(cities)
```

The Python logo, consisting of two interlocking snakes, one blue and one yellow, is centered in the background. Overlaid on the logo is the text "tuple (krotki)" in a white, sans-serif font.

tuple (krotki)

tuple

- niemutowalne, uporządkowane struktury danych
- raz ułożone elementy nie mogą zostać zmienione

tworzenie tupli:

- `empty_tuple = tuple()` – pusta tupla
- `empty_tuple = ()` – pusta tupla
- `Ola = ("Ola", "Poland", 1)`
- `Sarah = "Sarah", "USA", 2` – tuple można utworzyć również bez nawiasów

tuple

tupla jest niemutowalna, więc jej elementów nie można zmieniać

```
Sarah = ("Sarah", "USA", 2)
Sarah[1] = "France"
print(Sarah)
```

wynik działania:

```
Traceback (most recent call last):
  File "main.py", line 9, in <module>
    Sarah[1] = "France"
TypeError: 'tuple' object does not support item assignment
```

tuple

poszczególne elementy tupli można wyciągnąć i przypisać je do nowych zmiennych

```
Sarah = ("Sarah", "USA", 2)
name_sarah = Sarah[0]
print(name_sarah)
```

wynik działania:

Sarah

konwersja tupli na listę

```
my_tuple = ('pies', 'kot', 'jeź', 'pająk')
```

```
print(type(my_tuple))
```

```
my_list= list(my_tuple)
```

```
print(type(my_list))
```

wynik działania:

```
<class 'tuple'>
```

```
<class 'list'>
```

zagnieżdżanie tupli

```
Ola = ("Ola", "Poland", 1)  
Sarah = ("Sarah", "USA", 2)
```

```
users = (Ola, Sarah)  
print(users)  
print(type(users))
```

wynik działania:

```
(( 'Ola', 'Poland', 1), ( 'Sarah', 'USA', 2))  
<class 'tuple'>
```

łączenie tupli

```
Ola = ("Ola", "Poland", 1)  
Sarah = ("Sarah", "USA", 2)
```

```
users = Ola + Sarah  
print(users)  
print(type(users))
```

wynik działania:

```
('Ola', 'Poland', 1, 'Sarah', 'USA', 2)  
<class 'tuple'>
```

rozpakowanie tupli

```
Ola = ("Ola", "Poland", 1)
```

```
name, country, id = Ola  
print(name, type(name))  
print(country, type(country))  
print(id, type(id))
```

wynik działania:

```
Ola <class 'str'>  
Poland <class 'str'>  
1 <class 'int'>
```

zamiana wartości

```
x, y = 10, 15  
print(x)  
print(y)
```

wynik działania:

10
15

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

wynik działania:

15
10

zadania

obiekty typu `tuple` są niemutowalne. Dany jest obiekt typu `tuple`:

```
members = (('Kasia', 23), ('Tomek', 19))
```

wstaw pomiędzy Kasię i Tomka obiekt `tuple` `('Marta', 37)`, a wynik wydrukuj do konsoli

oczekiwany rezultat:

```
('Kasia', 23), ('Marta', 37), ('Tomek', 19))
```


zadania - rozwiązanie

I sposób

```
members = (('Kasia', 23), ('Tomek', 19))  
kasia, tomek = members  
marta = ("Marta", 37)  
members = (kasia, marta, tomek)  
print(members)
```

II sposób

```
members = (('Kasia', 23), ('Tomek', 19))  
members = (members[0], ('Marta', 37), members[1])  
print(members)
```

The Python logo, consisting of two interlocking snakes, one blue and one yellow, is centered in the background. Overlaid on the logo is the text "sety (zbiory)" in a white, sans-serif font.

sety (zbiory)

zbiory

zbiór jest nieuporządkowaną strukturą danych
posiada jedynie unikatowe wartości – brak powtórzeń

tworzenie zbiorów:




```
set1 = set()  
set2 = {"Gdańsk", "Poznań", "Warszawa", "Gdańsk"}
```

wynik działania:

```
set() <class 'set'>  
{'Poznań', 'Warszawa', 'Gdańsk'} <class 'set'>
```

kiedy warto używać zbioru

zbiór przyda się szczególnie, gdy chcemy:

-  **usunąć duplikaty z kolekcji** – można przekonwertować listę albo tuplę na zbiór i wszystkie duplikaty zostaną usunięte
-  **sprawdzać istnienie elementu w kolekcji** – jeśli mamy wiele elementów, to w przypadku zbioru będzie to dużo wydajniejsze niż dla listy czy krotki
-  **wykonywać matematyczne operacje na kolekcjach** – np. wyznaczanie części wspólnej dwóch zbiorów, różnicy, itd.

zbiory

funkcja `set()` na łańcuchach

```
name = set("my Python")  
print(name)
```

wynik działania:

```
{'t', 'h', 'o', 'y', ' ', 'P', 'n', 'm'}
```

zbiór zawiera jedynie wartości unikalne

działania na zbiorach

```
A = {1, 2, 3, 4, 5, 6, 7}
```

```
B = {5, 6, 7, 8, 9}
```

suma zbiorów

```
print("Suma:", A | B)
```

```
print("Suma:", A.union(B))
```

wynik działania:

```
Suma: {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
Suma: {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

działania na zbiorach

```
A = {1, 2, 3, 4, 5, 6, 7}
```

```
B = {5, 6, 7, 8, 9}
```

część wspólna, iloczyn zbiorów

```
print("Część wspólna:", A & B)
```

```
print("Część wspólna:", A.intersection(B))
```

wynik działania:

```
Część wspólna: {5, 6, 7}
```

```
Część wspólna: {5, 6, 7}
```

działania na zbiorach

`A = {1, 2, 3, 4, 5, 6, 7}`

`B = {5, 6, 7, 8, 9}`

różnice zbiorów:

`print("Różnica A-B:", A - B)` - elementy, które należą do zbioru A, ale nie należą do zbioru B

`print("Różnica B-A:", B - A)` - elementy, które należą do zbioru B, ale nie należą do zbioru A

`print("Różnica symetryczna:", A ^ B)` - elementy nie będące częścią wspólną zbiorów A i B

wynik działania:

Różnica A-B: {1, 2, 3, 4}

Różnica B-A: {8, 9}

Różnica symetryczna: {1, 2, 3, 4, 8, 9}

działania na zbiorach

```
A = {1, 2, 3, 4, 5, 6, 7}
```

```
B = {5, 6, 7, 8, 9}
```

różnice zbiorów (zapis z metodami):

```
print("Różnica A-B:", A.difference(B))
```

```
print("Różnica B-A:", B.difference(A))
```

```
print("Różnica symetryczna:", A.symmetric_difference(B))
```

wynik działania:

```
Różnica A-B: {1, 2, 3, 4}
```

```
Różnica B-A: {8, 9}
```

```
Różnica symetryczna: {1, 2, 3, 4, 8, 9}
```

działania na zbiorach

podzbiory

Sprawdzenie czy zbiór B jest podzbiorem zbioru A, czyli czy wszystkie elementy zbioru B znajdują się w zbiorze A:

```
A = {0, 1, 2, 3}
```

```
B = {1, 2}
```

```
print("Czy zbiór B jest podzbiorem zbioru A:", B.issubset(A))
```

wynik działania:

True

metody używane na zbiorach

`add()` dodaje element do zbioru

`update()` zwraca zbiór A wraz z elementami dodanymi ze zbioru B - `(A.update(B))`

`discard()` usuwa określony element ze zbioru

`clear()` czyści zawartość zbioru

`pop()` usuwa element ze zbioru i zwraca go

The Python logo, consisting of two interlocking snakes, one blue and one yellow, is centered in the background. Overlaid on the logo is the word "słowniki" in a white, sans-serif font.

słowniki

słowniki

słowniki to nieuporządkowane struktury danych, budowane na zasadzie
klucz : wartość

```
moj_slownik = {  
    klucz : wartość,  
    klucz : wartość ...}
```

słowniki

```
person = {  
    "name" : "Jan",  
    "last_name" : "Nowak",  
    "age" : 37  
}  
  
print(person, type(person))
```

wynik działania:

```
{'name': 'Jan', 'last_name': 'Nowak', 'age': 37} <class 'dict'>
```

dodawanie danych do słownika

Aby dodać nową wartość do słownika należy posłużyć się następującą składnią:

```
person["height"] = 1.80  
print(person)
```

wynik działania:

```
{'name': 'Jan', 'last_name': 'Nowak', 'age': 37,  
'height': 1.8}
```

usuwanie danych ze słownika

za pomocą instrukcji del można usunąć ze słownika parę klucz : wartość

```
del person["height"]  
print(person)
```

wynik działania:

```
{'name': 'Jan', 'last_name': 'Nowak', 'age': 37}
```


metody używane na słownikach

`update()` aktualizacja danych słownika danymi z drugiego słownika

`pop()` pobiera wartość ze słownika i usuwa ją

`get()` pobranie danej ze słownika

`clear()` czyszczenie słownika

`keys()` pobranie nazw kluczy zapisanych w słowniku

`print(dir(dict))` – podgląd na dostępne metody

The Python logo, consisting of two interlocking snakes, one blue and one yellow, is centered in the background. Overlaid on the logo is the text "kontrola przepływu programu" in white, sans-serif font, arranged in two lines.

kontrola przepływu programu

instrukcja warunkowa

instrukcje warunkowe pozwalają na podejmowanie w kodzie decyzji dotyczących dalszego sposobu jego wykonania

```
if warunek:
```

```
    #instrukcja do wykonania, gdy warunek jest prawdziwy
```

```
else:
```

```
    #instrukcja do wykonania, gdy warunek jest fałszywy
```

! każda instrukcja, która ma być wykonana po spełnieniu warunku poprzedzona jest
• tabulatorem

instrukcja warunkowa

konstrukcja if else sprawdza warunek. Jeżeli wartością warunku będzie true, to nastąpi wykonanie pierwszego bloku kodu. Jeśli wynikiem jest false, wykonany będzie drugi blok kodu.

```
x = int(input("Twoja aktualna prędkość w terenie zabudowanym (km/h): "))

if x > 50:
    print("Twoja prędkość jest nieprawidłowa")
else:
    print("Twoja prędkość jest prawidłowa")
```

instrukcja warunkowa

klauzula elif

```
x = int(input("Twoja aktualna prędkość w terenie zabudowanym (km/h): "))


if x > 50:
    print("Twoja prędkość jest przekroczona")
elif x == 50:
    print("Twoja prędkość jest maksymalną dozwoloną prędkością")
else:
    print("Twoja prędkość jest prawidłowa")
```


zagnieżdżanie instrukcji warunkowych

```
x = int(input("podaj pierwszą liczbę: "))
y = int(input("podaj drugą liczbę: "))


if x != y:
    print('Liczby są różne')
    if x > y:
        print(f'liczba {x} jest większa niż {y}')
    else:
        print(f'liczba {x} jest mniejsza niż {y}')
else:
    print(f'liczba {x} = {y}')
```

instrukcja warunkowa – co należy zapamiętać

 jeśli warunek jest spełniony wykonają się instrukcje bloku `if`, jeśli nie jest spełniony - bloku `else`

 po słowie kluczowym `if` oraz warunku następuje symbol dwukropka, podobnie po słowie kluczowym `else`

 instrukcje w blokach `if` oraz `else` poprzedzone są tabulatorem

 instrukcje niepoprzedzone symbolem tabulatora nie należą do bloków `if` oraz `else` i zostaną wykonane niezależnie od nich

pętla for

pętla for służy do wykonania określonych czynności na zbiorze danych. Zbiór ten może być np.: elementami listy czy liczbami z podanego zakresu

```
for zmienna in zbiór_danych:  
    instrukcje do wykonania
```

! każda instrukcja, która ma być wykonana w pętli poprzedzona jest tabulatorem

pętla for

```
animals = ["tygrys", "lew", "lampart", "puma"]  
  
for animal in animals:  
    print(f"ilość znaków w {animal} to: {len(animal)}")
```

wynik działania:

```
ilość znaków w tygrys to: 6  
ilość znaków w lew to: 3  
ilość znaków w lampart to: 7  
...
```

pętla for – analiza działania

```
animals = ["tygrys", "lew", "lampart", "puma"]
```

```
for animal in animals:
```

powyższe polecenie nakazuje Pythonowi pobranie pierwszej wartości z listy `animals` oraz umieszczenie jej w zmiennej `animal`. Pierwsza wartość to `"tygrys"`. Następnie Python przechodzi do kolejnego wiersza kodu.

pętla for – analiza działania

```
print(f"ilość znaków w {animal} to: {len(animal)}")
```

to polecenie powoduje wyświetlenie zmiennej `animal`, którą nadal jest `"tygrys"`, oraz długości tej zmiennej w zdaniu:

```
ilość znaków w tygrys to: 6
```

Ponieważ lista zawiera więcej wartości, Python powraca do pierwszego wiersza pętli `for animal in animals:`, pobiera kolejną wartość z listy i ponownie przechodzi do kolejnego wiersza pętli.

Kod jest powtarzany tak długo, dopóki pętla nie zostanie wykonana na wszystkich elementach listy.

wykonanie operacji po pętli for

wszystkie znajdujące się po pętli for wiersze kodu, które nie są wcięte, zostaną wykonane już tylko jeden raz, bez powtórzenia.

```
animals = ["tygrys", "lew", "lampart", "puma"]

for animal in animals:
    print(f"ilość znaków w {animal} to: {len(animal)}")

print("wszystkie zwierzęta to dzikie koty")
```

błędy związane z wcięciami

```
animals = ["tygrys", "lew", "lampart", "puma"]
```

```
for animal in animals:  
    print(f"ilość znaków w {animal} to: {len(animal)}")
```

wynik działania:

```
File "c:\Users\Magdalena Madej\Desktop\Python\new.py", line 12  
    print(f"ilość znaków w {animal} to: {len(animal)}")  
    ^
```

IndentationError: expected an indented block

funkcja range()

funkcja range() ułatwia generowanie serii liczb.

aby wygenerować serię liczb z danego zakresu należy podać odpowiednie wartości podczas wywołania funkcji range(x, y)

```
for value in range(1, 5):  
    print(value)
```

należy pamiętać, że za pomocą funkcji range(1, 5) wygenerowane zostaną liczby z przedziału lewostronnie domkniętego: <1; 5)

użycie funkcji range() do utworzenia listy liczb

```
numbers = list(range(5, 11))  
print(numbers)
```

wynik działania:

```
[5, 6, 7, 8, 9, 10]
```

użycie funkcji range() do utworzenia listy liczb

po przekazaniu funkcji range() trzeciego argumentu każda kolejna generowana liczba będzie zwiększana o wartość tego argumentu

```
lista = list(range(2, 11, 2))  
print(lista)
```

wynik działania:

```
[2, 4, 6, 8, 10]
```


funkcja range()

funkcji range() można przekazać tylko jeden argument, wówczas sekwencja liczb będzie rozpoczynała się od 0.

```
lista = list(range(10))  
print(lista)
```

wynik działania:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

instrukcja break

instrukcja break przerywa działanie pętli i przechodzi od razu do dalszej części programu

```
name = "Python"  
for x in name:  
    print(x)
```

wynik działania:

P
y
t
...

```
name = "Python"  
for x in name:  
    if x == "h" : break  
    print(x)
```

wynik działania:

P
y
t

instrukcja continue

instrukcja continue przerywa obieg pętli, pętla przechodzi bezpośrednio do kolejnego obiegu

```
name = "Python"
```

```
for x in name:
```

```
    if x == "h" : continue
```

```
    print(x)
```



wynik działania:

P

y

t

o

n

pętla while

pętla while działa dopóty, dopóki zdefiniowany warunek przyjmuje wartość True

```
number = 1
```

```
while number < 11:  
    if number % 2 == 0:  
        print(number)  
    else:  
        print("liczba nieparzysta")  
    number += 1
```

instrukcja pass

pass to instrukcja, która nie wykonuje żadnych działań

obiekt jest wykorzystywany jako wypełniacz linii kodu jeśli składnia wymaga podania instrukcji, ale nie jest potrzebne wykonanie żadnej czynności

w Pythonie 3.x instrukcję pass można zastąpić znakiem ...

instrukcja pass

```
lista = list(range(10))

for x in lista:
    if x == 0:
        pass
    elif x % 2 == 0:
        print(x, "liczba parzysta")
    else:
        print(x, "liczba nieparzysta")
```

zadania

za pomocą funkcji `range()` i pętli `for` utwórz dwie listy malejące.
Pierwsza ma zawierać kwadraty liczb podzielnych przez 3 z zakresu od 1 do 20, a druga pozostałe liczby, które nie zostały podniesione do drugiej potęgi.

oczekiwany rezultat:

```
[324, 225, 144, 81, 36, 9]
```

```
[20, 19, 17, 16, 14, 13, 11, 10, 8, 7, 5, 4, 2, 1]
```

zadania – rozwiązanie

```
squares = []  
rest = []  
for a in range(20, 0, -1):  
    if a % 3 == 0:  
        squares.append(a**2)  
    else:  
        rest.append(a)  
  
print(squares)  
print(rest)
```


zadania

co będzie wynikiem programu?

```
#pierwszy program
liczba = 10
for x in range(0,10):
    liczba = liczba - 1
    if x == 4:
        break
print(liczba)
```

co będzie wynikiem programu?

```
#drugi program
lista = [4, 5, 9, 3, 5, 3, 7, 1]

liczba = 0
for x in lista:
    if x == 3 or x == 5:
        continue
    else:
        liczba = liczba + x
print(liczba)
```

The Python logo, consisting of two interlocking snakes, one blue and one yellow, is centered in the background. Overlaid on the logo is the text "input/output" in a white, sans-serif font.

input/output

wczytywanie plików

Słowo kluczowe *with* powoduje zamknięcie pliku, gdy dostęp do niego nie będzie dłużej potrzebny. Python sam ustali odpowiedni moment.

funkcja, która otworzy plik i pozwoli uzyskać do niego dostęp

nazwa pliku

zmienna, w której Python przechowuje obiekt, reprezentujący plik, zwrócony przez funkcję *open()*


```
with open("plik.txt", "r") as file:  
    content = file.read()  
print(content)
```


metoda *read()* wczyta całą zawartość pliku

wczytywanie plików

funkcja `open(file, mode)` pozwala otworzyć plik i zwraca go jako obiekt

tryby otwierania plików:

 `'r'` – read – otwiera plik do odczytu, zwraca błąd jeśli plik nie istnieje

 `'a'` – append – otwiera plik do dopisania, tworzy plik jeśli nie istnieje

 `'w'` – write – otwiera plik do zapisu, tworzy plik jeśli nie istnieje

wczytywanie plików

```
file = open('simple.txt', 'r')
for line in file:
    print(line, end="")
file.close()
```

wynik działania:

pierwsza linia
druga linia
trzecia linia

plik simple.txt

pierwsza linia
druga linia
trzecia linia

wczytywanie plików

Wczytywanie pliku za pomocą metody `readlines()`

```
with open("simple.txt", "r") as file:  
    lines = file.readlines()  
    for line in lines:  
        print(line.rstrip())
```

wynik działania:

pierwsza linia

druga linia

trzecia linia

zapisywanie do pliku

```
techs = ['python', 'java', 'css', 'c++']
```

```
with open('techs.txt', 'w') as file:
```

```
    for tech in techs:
```

```
        print(tech, file = file)
```

Python utworzy plik o nazwie techs.txt (jeśli go nie ma), a następnie doda do niego wszystkie elementy listy techs, każdą w nowej linii.

zapisywanie do pliku

Zapisywanie danych do pliku za pomocą funkcji `write()`

```
even_numbers = list(range(100))[::2]

with open('numbers.txt', 'w') as file:
    for number in even_numbers:
        file.write(str(number) + '\n')
```


zadania

Przygotuj plik names.txt, który będzie zawierał imiona 10 użytkowników (każdy użytkownik powinien być zapisany w nowej linii).

Następnie w pliku main.py stwórz pustą listę names = [], dodaj do niej imiona użytkowników z pliku i wyświetl zawartość listy.

oczekiwany rezultat:

```
['Magda', 'Kacper', 'Bartek', 'Zosia', 'Mariusz', 'Tomek',  
'Sylwia', 'Janek', 'Olek', 'Zuzia']
```

zadania - rozwiązanie

```
names = []
```

```
with open('names.txt', 'r') as file:  
    for line in file:  
        names.append(line.rstrip())  
print(names)
```






The Python logo, consisting of two interlocking snakes, one blue and one yellow, is positioned in the background. The text "przekształcanie struktur danych" is overlaid on the logo in a white, sans-serif font.

przekształcanie struktur danych

list comprehension

list comprehensions pozwalają na stworzenie listy wartości spełniających pewne warunki

list comprehensions dostępne są w 5 postaciach:

-  prostej
-  prostej warunkowej
-  rozszerzonej
-  rozszerzonej z jednym warunkiem
-  rozszerzonej z wieloma warunkami

list comprehension

tworzenie listy zawierającej kwadraty liczb parzystych mniejszych od 20

klasycznie

```
numbers = []

for number in range(20):
    if number % 2 == 0:
        numbers.append(number ** 2)

print(numbers)
```

z zastosowaniem list comprehension

```
numbers = [x ** 2 for x in range(20)
            if x % 2 == 0]

print(numbers)
```

list comprehension

postać prosta:

```
lista = [wyrażenie for zmienna in zbiór_danych]
```

przykłady:

```
lista1 = [x * 2 for x in range(20)]
```

```
lista2 = [(x, x ** 2) for x in range(1, 5)]
```

list comprehension

postać prosta warunkowa:

```
lista = [wyrażenie for zmienna in zbiór_danych if warunek]
```

przykład:

```
lista1 = [x ** 2 for x in range(20) if x % 3 == 0]  
print(lista1)
```

wynik działania:

```
[0, 9, 36, 81, 144, 225, 324]
```

list comprehension

postać rozszerzona:

```
lista = [wyrażenie for zmienna1 in zbiór_danych1  
            for zmienna2 in zbiór_danych2]
```

przykład (wszystkie możliwe pary elementów):

```
lista1 = [(x,y) for x in range(1,5) for y in range(4,0,-1)]  
print(lista1)
```

wynik działania:

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


list comprehension

postać rozszerzona z jednym warunkiem:

```
lista = [wyrażenie for zmienna1 in zbiór_danych1  
          for zmienna2 in zbiór_danych2  
          if warunek]
```

list comprehension

przykład (wszystkie możliwe pary elementów, jeżeli pierwszy element jest mniejszy od drugiego):

```
lista1 = [(x, y) for x in range(1, 5)
           for y in range(6, 3, -1) if x < y]
print(lista1)
```

wynik działania:

```
[(1, 6), (1, 5), (1, 4), (2, 6), (2, 5), (2, 4), (3, 6), (3, 5), (3, 4), (4, 6), (4, 5)]
```

list comprehension

listy zagnieżdżone:

```
lista1 = [[j for j in range(10)] for i in range(3)]  
print(lista1)
```

wynik działania:

```
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],  
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],  
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]
```

set comprehension

tworzenie zbioru zawierającego kwadraty liczb z dowolnego zakresu

```
set1 = {i**2 for i in range(5)}  
print(set1)
```

wynik działania:

```
{0, 1, 4, 9, 16}
```

set comprehension

przykład II: zbiór unikalnych wyrazów

na początku tworzona jest zmienna `text`, w której wszystkie litery są zamieniane na małe za pomocą metody `lower()`.

następnie metoda `replace()` zmienia wszystkie kropki w zmiennej `text` na puste znaki, a metoda `split()` rozdziela wyrazy w zmiennej, tworząc listę.

```
text = "Python rządzi. Python radzi. Python nigdy Cię nie  
zdradzi. :P"
```

```
words = text.lower().replace('.', ' ').split()
```

set comprehension

teraz, za pomocą set comprehension, tworzony jest słownik, zawierający jedynie unikalne wyrazy

```
unique_words = {word for word in words}  
print(unique_words, type(unique_words))
```

wynik działania:

```
{'nigdy', 'radzi', 'nie', 'python', ':p', 'rządzi',  
'zdradzi', 'cię'} <class 'set'>
```

set comprehension

na koniec jego zawartość zostaje zmodyfikowana do wyrazów, których długość jest większa niż 4

```
unique_words_gt_4 = {word for word in words  
                      if len(word) > 4}  
print(unique_words_gt_4)
```

wynik działania:

```
{'nigdy', 'radzi', 'python', 'rządzi', 'zdradzi'}
```

dict comprehension

podstawowy zapis dict comprehension

```
dict1 = {"key1" : "value1", "key2" : "value2"}
```

```
dict_compr = {key : value for (key, value) in  
dict1.items() }
```


dict comprehension

```
names = { "Tomek" : "Tomasz", "Zuzia" : "Zuzanna",  
          "Ania" : "Anna", "Darek" : "Dariusz" }
```

tworzenie zbioru tupli za pomocą dict comprehension

```
names_set = {(key, value) for (key, value) in names.items()}  
print(names_set, type(names_set))
```

wynik działania:

```
{('Ania', 'Anna'), ('Darek', 'Dariusz'), ('Tomek', 'Tomasz'),  
 ('Zuzia', 'Zuzanna')} <class 'set'>
```

dict comprehension

odwracanie kluczy i wartości (dla unikalnych nazw)

```
names = { "Tomek" : "Tomasz", "Zuzia" : "Zuzanna",  
          "Ania" : "Anna", "Darek" : "Dariusz"}  
names_invert = {value : key for (key, value) in names.items()}  
print(names_invert)
```

wynik działania:

```
{'Tomasz': 'Tomek', 'Zuzanna': 'Zuzia', 'Anna': 'Ania',  
'Dariusz': 'Darek'}
```

zadania

Podany jest słownik:

```
stocks = {'Boombit' : 22, 'CD Project' : 295,  
'Playway' : 350}
```

Wykorzystując dict comprehension wydobądź ze słownika pary klucz : wartość o wartości powyżej 100. Wyniki wydrukuj do konsoli

oczekiwany rezultat:

```
{ 'CD Project' : 295, 'Playway' : 350 }
```

zadania – rozwiązanie

```
stocks = {'Boombit' : 22, 'CD Project' : 295,  
          'Playway' : 350}
```

```
stocks1 = {key : value for (key, value) in  
stocks.items() if value > 100}  
print(stocks1)
```



funkcje

- 🐍 funkcja jest wyodrębnionym fragmentem kodu, zamkniętym w pewnej strukturze posiadającej swoją nazwę
- 🐍 odwołując się poprzez tę nazwę wykonujemy dany fragment kodu w dowolnym miejscu programu, bez konieczności powielania tych samych treści

funkcje wbudowane

funkcje wbudowane, które pojawiały się do tej pory

<code>input()</code>	pobiera dane od użytkownika
----------------------	-----------------------------

<code>print()</code>	wyświetla przekazane argumenty
----------------------	--------------------------------

<code>range()</code>	generuje serię liczb
----------------------	----------------------

<code>open()</code>	otwiera plik
---------------------	--------------

<code>type()</code>	sprawdza typ danych
---------------------	---------------------

<code>dir()</code>	zwraca listę atrybutów i metod wskazanego w argumencie obiektu
--------------------	----------------------------------------------------------------

<code>len()</code>	sprawdza długość łańcucha znaków lub ilość elementów w liście
--------------------	---------------------------------------------------------------

funkcje wbudowane

funkcje zmieniające typ

<code>str()</code>	tworzy pusty string lub zmienia dane na łańcuch znaków
<code>int()</code>	tworzy liczbę 0 lub zmienia dane na liczbę całkowitą
<code>float()</code>	tworzy liczbę 0.0 lub zmienia dane na liczbę zmiennoprzecinkową
<code>list()</code>	tworzy pustą listę lub zmienia dane na listę
<code>tuple()</code>	tworzy pustą tuplę lub zmienia dane na tuplę
<code>set()</code>	tworzy pusty zbiór lub zmienia dane na zbiór
<code>dict()</code>	tworzy pusty słownik

funkcje wbudowane

funkcje kodów ASCII

`chr(liczba)` zmienia liczbę na odpowiedni znak z tabeli kodów ASCII

`ord(znak)` zmienia znak na odpowiadający mu kod ASCII

```
print(chr(65), ord("M"))
```

wynik działania:

A 77

funkcje wbudowane

inne funkcje wbudowane

<code>abs()</code>	zwraca wartość bezwzględną
<code>eval()</code>	pozwała wykonywać działania zapisane jako tekst
<code>round()</code>	zaokrągla wartość
<code>filter()</code>	jako pierwszy argument pobiera funkcję zwracającą True lub False, stosuje ją do każdego elementu sekwencji podanej jako argument drugi i zwraca tylko te, które spełniają założony warunek
<code>zip()</code>	łączy ze sobą elementy różnych obiektów iterowalnych
<code>map()</code>	daje możliwość wykonania zadanej funkcji dla każdego elementu kolekcji

funkcje wbudowane – enumerate()

funkcja `enumerate()`

funkcja, która umożliwia iterację po obiektach, takich jak lista, przy jednoczesnej informacji, która iteracja jest wykonywana

```
countries = ['Polska', 'Niemcy', 'Grecja', 'Szwecja', 'Rosja']
```

```
for i, country in enumerate(countries):  
    print(i, country)
```

funkcje wbudowane – enumerate()

```
people = [('Dawid', 25), ('Jan', 23), ('Marcin', 22)]  
  
for i, (name, age) in enumerate(people):  
    print(f"użytkownik {i + 1}. {name}, wiek: {age}")
```

wynik działania:

użytkownik 1. Dawid, wiek: 25

użytkownik 2. Jan, wiek: 23

użytkownik 3. Marcin, wiek: 22

funkcje wbudowane

pozostałe funkcje wbudowane wraz z opisami i przykładami znajdują się na stronie głównej Pythona pod linkiem:

<https://docs.python.org/3/library/functions.html?highlight=built%20functions>

metody typów prostych

metody typu str

<code>.capitalize()</code>	pierwsza litera tekstu zmieniana jest na wielką
<code>.title()</code>	pierwsza litera w każdej części tekstu zmieniana jest na wielką
<code>.lower()</code>	zmienia wszystkie litery tekstu na małe
<code>.upper()</code>	zmienia wszystkie litery tekstu na wielkie
<code>.swapcase()</code>	zmienia w tekście wielkie litery na małe, a małe na wielkie
<code>.join()</code>	tworzy string łącząc stringi w liście przekazanej jako argument
<code>.split()</code>	tworzy listę słów danego łańcucha

metody typów prostych

metody typu str zwracające True lub False

<code>.isalpha()</code>	zwraca True, jeśli w tekście, na którym działa są same litery
<code>.isalnum()</code>	zwraca True, jeśli w tekście, na którym działa są znaki alfanumeryczne
<code>.isdigit()</code>	zwraca True, jeśli w tekście, na którym działa są same cyfry
<code>.islower()</code>	zwraca True, jeśli w tekście, na którym działa są tylko małe litery
<code>.isupper()</code>	zwraca True, jeśli w tekście, na którym działa są tylko wielkie litery
<code>.startswith()</code>	zwraca True, jeśli tekst zaczyna się od znaków zapisanych w nawiasie
<code>.endswith()</code>	zwraca True, jeśli tekst kończy się znakami zapisanymi w nawiasie

jak zdefiniować własną funkcję

funkcja definiowana jest za pomocą
słowa kluczowego def

↓

```
def nazwa_funkcji (parametry):  
    działania do wykonania w funkcji
```

↑

działania, które mają się wykonać w funkcji, są wpisane w
kolejnych liniach, które rozpoczynają się od wcięcia

funkcje bez parametru

```
def witaj ():  
    name = input("Podaj swoje imię: ")  
    print(f"Witaj {name}")  
witaj()
```

wynik działania:

Podaj swoje imię: Ola

Witaj Ola

funkcje z parametrem

definiując funkcję należy określić listę **parametrów**, przy wywołaniu funkcji przekazane są do niej **argumenty**.

```
def suma(liczba1, liczba2):  
    suma = liczba1 + liczba2  
    print(f"Suma {liczba1} i {liczba2} wynosi: {suma}")  
suma(49, 56)
```

wynik działania:

Suma 49 i 56 wynosi: 105

wartości domyślne funkcji

parametry funkcji mogą mieć przypisane wartości domyślne. Zostaną użyte wtedy, gdy przy wywołaniu funkcji nie zostanie podana żadna wartość

```
def witaj(name = "Marta"):
    print(f"Witaj {name}!")
witaj()
```

wynik działania:

Witaj Marta!

wartości domyślne funkcji

funkcje mogą zawierać wiele parametrów, z których część (lub wszystkie) może mieć wartości domyślne

```
def suma(liczba1, liczba2 = 78):  
    suma = liczba1 + liczba2  
    print(f"Suma {liczba1} i {liczba2} wynosi: {suma}")  
suma(49, 56)  
suma(49)
```

wynik działania:

Suma 49 i 56 wynosi: 105

Suma 49 i 78 wynosi: 127

wartości domyślne funkcji

- 🐍 aby można było pominąć parametry z wartościami domyślnymi, trzeba je zdefiniować na końcu listy parametrów.
- 🐍 parametr z wartością domyślną zapisany na początku listy spowoduje błąd i uniemożliwi korzystanie z funkcji

wartości domyślne funkcji

```
def suma(liczba1 = 78, liczba2):  
    suma = liczba1 + liczba2  
    print(f"Suma {liczba1} i {liczba2} wynosi: {suma}")  
suma(49)
```

wynik działania:

File "c:\Users\Magdalena Madej\Desktop\Python\new.py", line 1

```
def suma(liczba1 = 78, liczba2):
```

^

SyntaxError: non-default argument follows default argument

funkcje zwracające wynik

funkcja zwracająca jakąś wartość ma na końcu słowo `return`, a za nim wartość, która ma być jej wynikiem

```
def devide_by_3(x):  
    return x / 3  
print(devide_by_3(6))
```

wynik działania:

2.0

dowolna liczba argumentów `*args`

kiedy nie wiadomo, jaką liczbę argumentów będzie musiała akceptować funkcja, można skorzystać z parametru, który pozwala funkcji pobrać dowolną liczbą argumentów z wywołującego ją polecenia: `*args`

gwiazdka w nazwie `*args` informuje Pythona o konieczności utworzenia pustej krotki o nazwie `args` i umieszczenia w niej otrzymanych wartości

`args` jest ogólną nazwą parametru. Może przyjąć każdą inną nazwę.

dowolna liczba argumentów *args

```
def test_args(x, *args):  
    print(f"pierwszy parametr: {x}")  
    for arg in args:  
        print(f"kolejny parametr z *args: {arg}")
```

```
test_args(4)
```

```
test_args(4, 5, 6, 7, 8)
```

dowolna liczba argumentów *args

funkcja, która będzie sumowała dowolną liczbę argumentów:

```
def suma(*args):  
    return sum(args)
```

```
suma(1, 3, 6, 10, 5, 7, 8)
```

wynik działania:

40

dowolna liczba argumentów *args

funkcja, która będzie obliczała średnią dowolnej liczby argumentów:

```
def srednia(*args):  
    return sum(args)/len(args)
```

```
round(srednia(1, 3, 6, 10, 5, 7, 8), 2)
```

wynik działania:

5.71

kiedy pojawiają się błędy

nie zostaną przekazane
wszystkie wymagane
argumenty funkcji

podanych zostanie zbyt dużo
argumentów

nazwa funkcji zostanie źle
podana (błąd może dotyczyć
również wielkości liter)

wartości domyślne funkcji
zostaną wpisane na początku
listy parametrów

dwie różne funkcje będą miały
taką samą nazwę, przez co
druga nadpisze pierwszą

wpisane zostaną wartości
funkcji, które nie odpowiadają
potrzebom programu (np.
łańcuch tam, gdzie powinna być
liczba)

zadania

stwórz funkcję `make_pizza`, która pozwoli wybrać rozmiar pizzy i dowolną ilość składników. Złożone zamówienie wydrukuj do konsoli.

oczekiwany rezultat:

wybrałeś pizzę o wielkości 30 cm, z następującymi dodatkami:

- salami
- ser
- szynka

zadania - rozwiązanie

```
def make_pizza(size, *ingredients):  
    print(f"wybrałeś pizzę o wielkości {size} cm, z  
        następującymi dodatkami:")  
    for ingredient in ingredients:  
        print(f"- {ingredient}")  
  
make_pizza(30, "salami", "ser", "szynka")
```

The Python logo, consisting of two interlocking snakes, one blue and one yellow, is centered in the background.

wyrażenie lambda

wyrażenie lambda

słowo kluczowe `lambda` pozwala utworzyć zwięzły odpowiednik prostej, jednowyrażeniowej funkcji

słowo kluczowe lambda

kod, który ma być wykonany

```
lambda x: x ** 2 + 3
```

parametr

wyrażenie lambda

przykłady:

```
y = lambda x: x ** 2 + 3  
print(y(30))
```

wynik działania:

903

```
f_2 = lambda w: w.upper()  
print(f_2('python'))
```

wynik działania:

PYTHON

wyrażenie lambda

generowanie kwadratów liczb określonych w zakresie funkcji range()

```
kwadraty = map(lambda x: x ** 2, range(10))  
print(list(kwadraty))
```

wynik działania:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

wyrażenie lambda

```
def apply_function(x, fn):  
    return fn(x)
```

```
apply_function(5, lambda x: x ** 2)
```

```
apply_function([15, 20], lambda x: sum(x))
```

wynik działania:

25

35

wyrażenie lambda

Zastosowanie lambda w kluczu, w funkcji sorted()

```
numbers = [-3, -2, -1, 0, 1, 2, 3]  
sorted(numbers, key = lambda x: abs(x))
```

wynik działania:

```
[0, -1, 1, -2, 2, -3, 3]
```

podgląd na argumenty funkcji sorted(): `help(sorted)`

zadania

Podana jest lista miast:

```
city = ['Warsaw', 'London', 'Berlin', 'New York']
```

Używając wyrażenia lambda stwórz listę zawierającą trzy pierwsze litery każdego miasta. Wynik wydrukuj do konsoli.

oczekiwany rezultat:

```
['War', 'Lon', 'Ber', 'New']
```

zadania - rozwiązanie

```
city = ['Warsaw', 'London', 'Berlin', 'New York']  
  
print(list(map(lambda word: word[:3], city)))
```



generatory

generatory to funkcje zwracające obiekt, po którym można iterować. Od zwykłych funkcji różnią się tym, że:

- 🐍 zwracają iterator za pomocą słowa kluczowego `yield`,
- 🐍 zapamiętują swój stan z momentu ostatniego wywołania, są więc wznowialne
- 🐍 zwracają następną wartość ze strumienia danych podczas kolejnych wywołań metody `next()`

generator

```
def gen_parzyste(N):  
    for i in range(N):  
        if i % 2 == 0:  
            yield i
```

```
gen = gen_parzyste(10)  
print(list(gen))
```

wynik działania:

```
[0, 2, 4, 6, 8]
```

funkcja next()

funkcja next() pobiera kolejny element z iteratora

```
def gen_parzyste(N):  
    for i in range(N):  
        if i % 2 == 0:  
            yield i
```

```
gen = gen_parzyste(10)  
next(gen)  
next(gen)  
print(list(gen))
```

zadania

Dana jest lista zawierająca nazwy plików

```
files = ['run_me.py', 'README.md', 'help.txt.',  
'script.js', 'menu.py', 'main.py', 'py']
```

stwórz generator, który przefiltruje podane pliki w liście i zwróci tylko te z rozszerzeniem .py

oczekiwany rezultat:

```
['run_me.py', 'menu.py', 'main.py']
```

zadania – rozwiązanie

```
files = ['run_me.py', 'README.md', 'help.txt.',  
        'script.js', 'menu.py', 'main.py', 'py']  
def generator_py(lista):  
    for item in lista:  
        if item.endswith('.py'):  
            yield item  
  
gen = generator_py(files)  
list(gen)
```



lista słów kluczowych

aby sprawdzić listę słów kluczowych, którymi nie można nazywać zmiennych czy funkcji, wystarczy wpisać i uruchomić poniższy kod

```
import keyword  
print(keyword.kwlist)
```

funkcje `dir()` i `help()`

`dir()`

funkcja `dir()` wywołana z argumentem zwróci listę atrybutów i metod wskazanego obiektu

```
np. dir(list)
```

`help()`

funkcja `help()` wywołuje wbudowany system pomocy

```
np. help(list)
```

bibliografia

 Dawson Michael - Python dla każdego. Podstawy programowania, 2014

 Matthes Eric - Python. Instrukcje dla programisty, 2020

 Lutz Mark - Python. Wprowadzenie. Wydanie V, 2020

 <https://www.python.org/>