Python - podstawy

Ogólne

Python jest wydany na zmodyfikowanej licencji GPL, możliwa jest sprzedaż swoich programów. Pliki zapisujemy z rozszerzeniem *.py. W przeciwieństwie do Java'y, która wymaga kompilacji, Python jest językiem interpretowanym (występuje zawsze jako czysty tekst). Minusem braku kompilacji jest brak wykrycia błędów już w fazie kompilowania: żeby wykryć błędy, trzeba pliki *.py po prostu uruchomić. Jedyny wyjątek stanowią moduły: wczytane po raz pierwszy, są kopiowane do postaci *.pyc (czyli są częściowo kompilowane, aby przyspieszyć każde następne wykorzystanie modułu).

Aby zainstalować interpreter (oraz biblioteki i IDLE) w systemie *Windows*, należy udać się na stronę *http://python.org* i pobrać odpowiedni pakiet. W czasie instalacji pamiętać o opcji „*Add Python to PATH*".

- **CPython** - oryginalna kanoniczna wersja Pythona;
- **Cython** - narzędzie do zmiany kodu Python na kod języka C;
- **PyPy** - narzędzie developerów języka Python, napisane w RPython (nowe funkcjonalności testowane są poprzez przetłumaczenie w locie na język C);

Środowiska programistyczne:

- **IDLE** („***I****ntegrated* ***D****eve****L****opment* ***E****nvironment*" - czysty shell, środowisko dopuszczone do matury; odpowiednikiem w Linuksie będzie czysta konsola: po prostu wpisujesz „python"... i już!); można także użyć konsoli kolorowanej: program „*ipython*" (obecny w *Gentoo*);
- konsole online: https://www.python.org/shell/, https://repl.it/languages/python3
- **Mu-Editor** (bardziej graficzne, notatnik do kodu, umożliwia pisanie skryptów: *File / New File...*); nieobecny w dystrybucji *Gentoo*;
- **Bluefish** - w wersji linuksowej jest tylko opcja pythona 2.7;
- **CodeCombat** (przeglądarkowa gra dla uczniów ucząca programowania, wymaga rejestracji);
- **PyCharm** - profesjonalne środowisko, w wersji *Community* jest bezpłatne; do pełnego działania wymaga jednak instalacji dodatkowych zamkniętych wtyczek;
- **web2py** - framework dedykowany rozwiązaniom webowym (nieobecny w *Gentoo*);
- **Flask** - framework (obecny w *Gentoo*);
- **Django** - framework (obecny w *Gentoo*);
- **Eric** - obecny w *Gentoo*, obsługuje całe projekty;
- **Geany** - środowisko dla *Python* (choć także dla C, Pascal, HTML), obecny w *Gentoo*; ważna opcja: *Edycja / Preferencje / Terminal / [x] Wykonywanie programów we wbudowanym terminalu* oraz *[x] Nie używaj skryptu uruchomieniowego*;
- **Spyder** - preferowany, spory kombajn z wieloma zależnościami (obecny w *Gentoo*), ale też prosty w obsłudze;

Uwagi wstępne

- Python jest czuły na wielkość liter.
- Python akceptuje polskie znaki w zmiennych, ale nie powinno się ich używać (kod ma być czytelny dla wszystkich nacji).
- Nazwa zmiennej nie może zaczynać się od liczby.
- Lepiej stosować komentarze w języku angielskim (kod ma być czytelny dla wszystkich).

- Można używać średników na końcach linii, ale można je także pominąć (sam ENTER wystarczy).
- Stosowanie wcięć w kodzie - jest **konieczne**! Wcięcia muszą mieć tę samą długość: domyślnie są to cztery spacje (edytor zamienia tabulator na cztery spacje). Ale będzie działać nawet wtedy, gdy wszystkie wcięcia będą mieć dwie spacje (ważna jest konsekwencja stosowania).
- Podręczna pomoc: **help(***nazwaWbudowanejFunkcji***)**. W przypadku zewnętrznych funkcji (bibliotek), należy je wcześniej zaimportować.
- Szczegółowa dokumentacja znajduje się na: https://docs.python.org

Przykłady na shellu

>>> help(print) *(wypisuje pomoc na temat „print”)*
Alt+P - (*preview*) poprzednie polecenie;
Alt+N - (*next*) następne;

>>> print(*„Jakiś tekst*”)

Działania matematyczne:
>>> 2+2
4
>>> 3 / 2
1.5
>>> 3 // 2
1 *(całkowite)*

Zmienne:
>>> zmienna1 = *„Jakiś Tekst*”
>>> zmienna1
‘Jakiś tekst’

>>> zmienna1.upper() *(zamienia na duże litery)*
>>> zmienna1.count(‘k’) *(liczy ile razy wystąpiła litera „k”)*

>>> zmienna2 = 4
>>> type(zmienna2)
<class ‘**int**’> *(wypisuje typ zmiennej: string, float, int)*
>>> zmienna2 = int(liczba1) *(zamiana stringa na int)*
>>> dir(zmienna2) *(wypisuje metody danego obiektu)*

Zapytania:
>>> rok_urodzenia = int(input(*„Podaj swój rok urodzenia: ”*))
>>> int(‘10’) *(zamiana stringa na int)*
10

Komentarze

 # To jest komentarz
zmienna = 7 *# Tu też można dawać komentarze*

```
"""
Tutaj może znaleźć się cokolwiek.
Wszystko zostanie zignorowane.
"""
```

```
'''
Można użyć także trzech pojedynczych znaków cudzysłowu
'''
```

Znaki ucieczki

zmienna = '*I\'m student*' *# Znakiem ucieczki jest tutaj \.*

Znaki końca linii

Uwaga! System *Windows* i *Linux* mają odmienne znaki końca linii:

- Linux --> **LF** (kod ASCII: 10), znak \n;
- Windows --> **CR**+**LF** (kod ASCII: 13+10), znak \r\n.

Operatory matematyczne

Nawiasy decydują o kolejności działań; w przypadku braku nawiasów, stosowane są zasady matematyki formalnej.

/ - dzielenie;

***** - mnożenie;

****** - potęgowanie (np. 2 ** 3);

****0.5** - pierwiastkowanie;

e - wykładnik potęgowy (*exponent*), np. 3e8 = $3x10^8$, zaś stała Plancka (6.62607×10^{-34}) byłaby zapisana w Pythonie jako 6.62607e-34;

// - dzielenie z zaokrąglaniem w dół (wynik 3,85 zostanie zaokrąglony do 3);

% - modulo (to, co zostało po dzieleniu w reszcie, w liczniku, np. 7 % 3 = 2,3 = 2 $\frac{1}{3}$ = 1 reszty; lub 8 % 3 = 2,7 = 2 $\frac{2}{3}$ = 2 reszty; oraz 9 % 3 = 3 = 0 reszty);

powiększanie o procent - np. dodanie 23% VAT to pomnożenie przez 1.23 (inaczej: 1 + 23/100);

+=, **-=**, ***=**, **/=** - powiększanie / pomniejszanie / przypisanie, np. x += 4 (powiększ x o 4);

round(liczba, miejscaPoPrzecinku**)** - zaokrąglanie liczby;

Inne sposoby zaokrąglania (formatowania) wyniku (*String Formatting Operators*):

- '%.f' % 3.14 - domyślne zaokrąglenie liczby rzeczywistej do sześciu miejsc po przecinku (czyli: 3.140000);
- '%.1f' % 3.14 - zaokrąglenie liczby rzeczywistej (float) do jednego miejsca po przecinku (czyli: 3.1);
- '%.2f' % 3.14 - zaokrąglenie liczby rzeczywistej do dwóch miejsc po przecinku (czyli: 3.14);
- '%03d' % 7 - trzy liczby wiodące (czyli: 007);
- '%d' % 0b1111 - zamiana wyniku na dziesiętny (czyli: 15);

Uwaga: Język Python sam wybiera formę wyniku działań matematycznych (wybiera zawsze formę bardziej ekonomiczną).

Operatory porównania

> - większy niż;
< - mniejsze niż;
== - równe wartości zmiennych / obiektów (równe mogą być tylko te same typy danych);
is - równe obiekty (obiekt jest tożsamy, zajmuje ten sam segment pamięci);
>= - większe bądź równe;
<= mniejsze bądź równe;
!= - nierówne;
Uwaga: Porównywać można też łańcuchy znaków. W tym przypadku napis zawierający więcej liter jest większy, a także małe litery mają większą wartość niż duże (tak naprawdę porównywane są ich kody ASCII).

Operatory logiczne

and - przykładowo: **if** (warunek **and** innyWarunek);
or - przykładowo: **if** (warunek **or** innyWarunek);
not - przykładowo: **if** (**not** (warunek **and** innyWarunek));

Operatory bitowe (system binarny)

& - koniunkcja bitowa;
| - alternatywa bitowa;
~ - negacja bitowa;
^ - alternatywa rozłączna (xor), czyli „albo - albo”; poniżej przykład operacji *xor* na wartościach bitowych:
0b**101**
0b**110**
========
0b**011** *(gdyż 1 i 1 - daje fałsz (czyli 0), a 1 i 0 - daje prawdę (czyli 1)*.

Operacje na stringach

Łączenie stringów:
print(imie + „ „ + nazwisko**)**

Długie stringi:
print(„*To jest bardzo długi tekst. To jest pierwsza linia tego tekstu.*\
To jest druga linia tego tekstu.\
To jest trzecia linia tego tekstu.”**)**
zmienna = *”””To jest bardzo długi tekst. To jest pierwsza linia tego tekstu.*
To jest druga linia tego tekstu.
To jest trzecia linia tego tekstu.”””
lub
print(*”Jakiś tekst*” , ”*Inny tekst*”, end=” ”) *# No końcu printa nie będzie nowej linii, lecz spacja*
print(*”Niby druga linia, ale będzie wyświetlona razem z poprzednią*”)

lub
print(”*Imię*”,”*Nazwisko*”, sep=”_”**)** *# Wynik: Imię_Nazwisko.*

wybor = **input(**”””*Wybierz odpowiednią opcję:*
a) PL
b) DE
c) EN”””**)** *# Nie jest to traktowane jako komentarz w tym przypadku*
Zamiana liczb (int lub float) na string:
str(13**)**

Zamiana stringów na liczby:
int(‘13’**)**
float(‘3.14’**)**

Łączenie stringów i liczb:
print(„*Jakiś tekst*”, 2)
lub
print(„*Jakiś tekst*” + **string**(2))
wygeneruje pięć znaków ‘@’:
print(5 * ‘@’)

Wyszukiwanie elementu w stringu:
jakiśString = ‘*Agnieszka*’
print(‘*sz*’ in jakiśString) *# Zwraca True*
Uwaga: Nie można dodawać żadnych elementów do środka stringa ani ich usuwać, gdyż nie jest on indeksowany (tak jak listy). Można jednak zamienić tekst na listę: list(„abcdef”).

Zamiana na małe litery:
jakiśString.**lower()**

Tylko pierwsza litera duża, reszta mała:
‘*test StrinGu*’.**title()**

Zamiana znaków:
jakiśString.**replace(**‘*wyraz1*’,’*wyraz2*’) *# Zamienia w stringu ‘wyraz1’ na ‘wyraz2’*

Seryjne przekazywanie parametrów (wszystkie elementy pierwszego parametru są przekazywane do wszystkich elementów drugiego parametru):
map(par1, par2**)**
filter(par1, par2**)**

Usuwanie znaków początkowych:
jakiśString.**lstrip()** *# Usuwanie początkowych spacji*
jakiśString.**lstrip(‘dr ‘)** *# Usuwanie początkowego „dr”*

Dzielenie stringów:
‘*jeden dwa trzy*’.**split()** # Wynikiem będzie lista wyrazów (spacja traktowana jest jako znak oddzielenia, nawet gdy jest kilka spacji pod rząd)

Łączenie stringów (odwrotność dzielenia):
‘ ‘.**join(**[‘jeden’, ‘dwa’, ‘trzy’**)** # Połączenie wyrazów (separatorem jest spacja)

Usuwanie znaków wiodących:
‘*... abc ...*’.**strip(**‘...’**)** # Likwiduje trzykropek zarówno przed stringiem, jak i za nim

Czy string kończy się na „a”:
if ‘*Agnieszka*’.**endswitch(**‘a’**)**:
    print(‘*Tak, kończy się na A*’**)**
else:
    print(‘*Nie, nie kończy się na B*’**)**
Uwaga: Odwrotnością tej metody jest *startswitch()*, zaś metoda *strip()* likwiduje znaki wiodące zarówno przed stringiem, jak i za nim.

Czy string zawiera wyłącznie znaki alfanumeryczne:
if ‘*JakiśString*’.**isalnum()**:
    print(‘*Jest OK*’**)**
else:
    print(‘*Nie możemy zaakceptować wpisanych znaków*’**)**

Inne testy:
‘*JakiśString*’.**isalpha()** - czy zawiera wyłącznie litery;
‘*JakiśString*’.**isdigit()** - czy zawiera wyłącznie liczby;
‘*JakiśString*’.**islower()** - czy wszystkie litery są małe;
‘*JakiśString*’.**issupper()** - czy wszystkie litery są duże;
‘*JakiśString*’.**isspace()** - czy zawiera wyłącznie spacje;

Wydobycie z liter kodu ASCII:
print(**ord(**'A'**)**) *# Zwraca 65*
print(**chr(**65**)**) *# Zwraca ‘A’*

Odstępy i nowe linie:
print(„*Jakiś string*\n”) *# Nowa linia*
print(„*Imię*”, „*Nazwisko*”, sep=” ”) *# Domyślną wartością „sep” (separation) jest spacja (dotyczy każdego „print”), ale można ją zamienić na cokolwiek innego, np. sep=”\n” lub sep=jakasZmienna.*

Instrukcje warunkowe

if (jakiśWarunek):
    jakaśInstrukcja
elif (jakiśInnyWarunek):   *# Jest to skrót od „else if”*
    jakaśInnaInstrukcja
else (pozostałyWarunek):
    instrukcja
Uwaga: Instrukcje możemy zagnieżdżać (np. wewnątrz „elif” może znajdować się wewnętrzny „if”.
if (jakiśWarunek **and** innyWarunek):
    jakaśInstrukcja

if (jakiśWarunek **or** innyWarunek):
    jakaśInstrukcja
if (**not** (jakiśWarunek **and** innyWarunek)):
    jakaśInstrukcja

Zmienne predefiniowane

__main__ - reprezentuje nazwę aktualnie uruchamianego pliku (bez rozszerzenia);
__name__ - wywołana w bieżącym pliku, zwraca „*__main__*” (daje znać, że jest w głównym pliku); jeśli jest wywołana z modułu (zewnętrznego pliku), zwraca nazwę pliku tego modułu (bez rozszerzenia);
__module__ - w większości sytuacji, *__module__* = *__main__*;
__dict__ - zmienna w klasie, zawiera nazwy i ich wartości wszystkich właściwości klasy;
__zmienna - prywatna zmienna, która nie powinna być zmieniana przez innych użytkowników (to tylko konwencja);

Funkcje

Uwaga: Aby łatwiej odróżnić nazwy zmiennych od nazw funkcji, można użyć konwencji, iż nazwy funkcji rozdzielane są „podłogą” (znak _).

Tworzenie funkcji:
def nazwa_Funkcji():
    instrukcje
Wywołanie funkcji:
nazwa_Funkcji()

Zasięg zmiennych:
def nazwa_Funkcji():
    x = 5   *# Zmienna lokalna rozpoznawalna tylko wewnątrz funkcji.*
    return x   *# Zwraca jedynie wynik / wartość, czyli 5 (a nie zmienną „x”). Samo słowo „return” powoduje przerwanie wykonywania funkcji i powrót do miejsca wywołania funkcji.*
 # Jednak odwrotnie działa: zmienna globalna będzie rozpoznawana wewnątrz funkcji:
x = 17
def nazwa_Funkcji():
    print(x)   *# Zwróci „17”*
 # Jednak w przypadku wątpliwości (ta sama nazwa zmiennej funkcjonuje lokalnie i globalnie), dobrze jest określić, którą z nich mamy na myśli:
x = 20
def nazwa_Funkcji():
    global x   *# Mamy na myśli „x” globalne, a nie lokalne*
    x = x + 5
Uwaga: Lepiej jest używać „*return*” w funkcji niż „*global*”.

Tworzenie funkcji z parametrem:
def powitanie_Uzytkownika(imie):
    print(„*Witam Cię na mojej stronie, ”*, imię**)**

imiona = [‘*Adamie*’, ‘*Agnieszko*’, ‘*Kasiu*’]

```
for kazdyElement in imiona:
    powitanie_Uzytkownika(kazdyElement)
```

Uwaga: Przesłany argument staje się w funkcji parametrem i zmienną **lokalną** wewnątrz funkcji.

```
def jakaśFunkcja(*args): # Taka funkcja przyjmie dowolną ilość argumentów, np. dowolna ilość liczb do obliczania średniej.
def  jakaśFunkcja(*kwargs): # Argumenty keywordowe, np. "a = 3" (pary), pakowanie do słownika;
```

Uwaga: Można także utworzyć domyślną wartość argumentu (gdy użytkownik nie poda żadnego argumentu, funkcja skorzysta z domyślnego argumentu). Przydatne, na przykład, gdy rozbudowujemy zastany już program: dodanie domyślnego argumentu nie spowoduje błędnego działania starych wywołań funckji.

```
def jakas_Funkcja(parametr1, parametr2 = 5):
....
jakas_Funkcja(3)
```

Uwaga: Domyślnie, argumenty są pozycyjne (przesyłane według miejsca / pozycji). Można jednak przesyłać argumenty kluczowe (ale muszą się one znajdować zawsze po pozycyjnych):

```
jakas_Funkcja(argument2 = 2, argument1 = 8)
```

```
# Można także pogrupować kilka wartości w jeden argument (argument wielowartościowy) zapisywany wtedy jako *nazwaArgumentu. Do tego typu argumentu będą pobierane wszystkie przekazane argumenty pozycyjne (nienazwane) i będą tworzyć krotkę. Na samym końcu musimy więc wysłać argument kluczowy.
def jakas_Funkcja(parametr1, *parametr2, parametr3):

jakas_Funkcja(1, 34, 43, argument3 = 50)

# Jeśli chcielibyśmy wysłać kilka wartości nazwanych (kluczowych) jako jeden argument, musimy zmienną argumentu zapisać jako **nazwaArgumentu i będą one tworzyć słownik:
def jakas_Funkcja(parametr1, parametr2, **parametr3):
jakas_Funkcja(1, argument2 = 34, x = 5, y = 8)
# Możemy także wypisać wartość z takiego argumentu:
print(argument3.get(y)) # Wypisze 8
```

Rekurencja:

Funkcje mogą wywoływać same siebie (zapętlenie). Każdy przebieg takiego zapętlenia generuje jakiś wynik i są one do siebie dodawane na koniec działania pętli (*return*). Ważne, aby funkcja zawierała w sobie jakiś warunek przerywający jej działanie (to zapętlenie), gdyż inaczej będzie działać w nieskończoność (zawiesi się lub sam python uniemożliwi działanie takiej funkcji).

Funkcje anonimowe / lambda:

Tego typu funkcja jest przydatna, gdy chcemy użyć jej tylko raz (nie ma ona nazwy).

```
nazwaZmiennej = lambda nazwaArgumentu: nazwaArgumentu * 2 # Nie zawiera nazwy funkcji
print(nazwaZmiennej(7))    # Zamiast wywołania po nazwie
```

Zwracanie wyniku:

Jeśli chcielibyśmy użyć wyniku funkcji w innym miejscu programu (poza funkcją), musimy na końcu funkcji użyć słowa „*return*” (funkcja zwraca wynik na zewnątrz **do miejsca wywołania**; innymi słowy „wywołanie funkcji jest zastępowane przez to, co znajdzie się po słowie „return”; ten wynik jest jakby przypisany do nazwy funkcji jako zmiennej i możemy na tej funkcji wykonywać różne operacje):

```
def pole_Prostokata(a, b):
    return a * b      # Bez „return” zadziała tylko „print” w tym miejscu,
                      # ale lepiej jest printować wynik funkcji poza funkcją, a nie w jej wnętrzu
                      # Instrukcja „return” kończy też działanie funkcji (jeśli są jakieś
                      # instrukcje po „return” - nie będą wykonane).
print(2 * pole_Prostokata(2, 5))
```

Uwaga: Instrukcja „return” bez parametrów, zwraca surowy wynik: True, False, None, itp. Istnieje też polecenie „*yield*”, które służy do wielokrotnego zwracania wyniku podczas przebiegu pętli (nie zatrzymuje jej tak jak „*return*”).

Funkcja obliczająca wiek:

```
def obliczanie_Wieku(obecny_rok, rok_urodzenia):
'''
Komentarz wieloliniowy...
'''
ileMamLat = obecny_rok - rok_urodzenia
if ileMamLat > 18:
    jestesDorosly = True
else:
    jestesDorosly = False

return ileMamLat, jestesDorosly
```

Wywoływanie funkcji:

```
obliczanie_Wieku(2020, 1970)
print(obliczanieWieku)
```

Biblioteki / Moduły / Pakiety

Pojęcia:

1. Dzielenie programu na moduły nazywamy dekompozycją (każdy z modułów może rozwijać inny programista).

2. Przestrzeń nazw (*namespace*) to część programu, w których nie kolidują ze sobą nazwy (np. nazwy zmiennych), są one w tej przestrzeni unikatowe. Importując moduł, zawarte w nim nazwy nie wchodzą do (nie naruszają) przestrzeni nazw naszego programu. Aby skorzystać ze zmiennych lub funkcji modułu, należy odwołać się do tej zmiennej podając jej pełny adres (wraz z nazwą modułu), np. **math.pi**.

3. Zaimportowany moduł możemy używać pod inną nazwą (tzw. aliasing), co może być konieczne w przypadku identycznych nazw funkcji pochodzących z różnych modułów, np.:

```
import nazwaModułu as nazwaAliasu (nazwa oryginalnego modułu nie będzie dostępna, tylko alias)
```

lub

```
from nazwaModułu import nazwaFunkcji as nazwaAliasu
```

lub

```
from nazwaModułu import nazwaFunkcji1 as nazwaAliasu1, nazwaFunkcji2 as nazwaAliasu2
```

4. Moduł jest importowany przez nasz skrypt tylko raz. Kolejna próba zaimportowania go, jest ignorowana.

5. Aby zaimportować moduł z innej lokalizacji niż folder naszego programu:

```
from sys import path
path.append('C:\\Users\\adam\\py\\modules') # Znaki ucieczki
# lub relatywnie:
path.append('..\\jakiśFolderNadrzędny')
import twójModuł
```

6. Aby moduł / skrypt nie miał problemu z uruchomieniem w Linux, może w pierwszym komentarzu zdefiniować jego interpreter:

```
#! /usr/bin/env python3
lub
#! /usr/bin/env python
```

Skrypt można uruchamiać za pomocą polecenia:

```
# python plik.py
```

Aby uruchamiać go bezpośrednio, należy nadać mu prawa do uruchamiania („x") i wywołać go jako:

```
# ./plik.py
```

Różnice w importowaniu modułów:

import *moduł* - importujemy zmienne do osobnych *namespace'ów*, są one wtedy dostępne poprzez podanie **nazwaModułu**.nazwaKlasy;

from *moduł* import * - importuje wszystko do wspólnej *namespace*, nazwy klas są dostępne bezpośrednio, ale mogą zachodzić konflikty;

Przykład:

```
from time import sleep
sleep(5)
```

vs

```
import time
time.sleep(5)
```

Wewnętrzne biblioteki:

```
import math # Importuje bibliotekę matematyczną. Można zaimportować kilka modułów jednocześnie w jednej linii (należy oddzielić je przecinkiem)
import mojPrywatnyModul # Zaimportuje pod warunkiem, że plik z modułem (mojPrywatnyModul.py) będzie w tym samym folderze lub w podfolderze
print(math.sqrt(9))
from math import sqrt # Importuje z biblioteki tylko metodę sqrt(), choć można wskazać kilka metod rozdzielanych przecinkiem. Uwaga: W tej metodzie importu nie trzeba pisać całejj
print(sqrt(9)) # W tym przypadku nie trzeba (a nawet nie można) podawać nazwy „math"
print(__name__) # Wyświetla nazwę pliku (bez rozszerzenia), w którym komenda jest umieszczona
```

Uwaga: Aby wyświetlić wszystkie funkcje zawarte w danym module, można użyć komendy *dir()*:

```
import math
print(dir(math))
```

Własne biblioteki / moduły:

Jeśli plik z własną biblioteką znajduje się w tym samym folderze, co nasz skrypt, wystarczy jedynie:

import podatekVAT *# Jeśli plik ma nazwę „podatekVAT.py”*

Zewnętrzne biblioteki / moduły:

Biblioteka zewnętrznych modułów znajduje się na *http://pypi.org* (**Py**thon **P**ackage **I**ndex). Gdy znajdziemy nazwę pakietu, który nas interesuje:

$ **pip install** *nazwaPakietu* **--user** (instalacja na koncie użytkownika i tak ma być!)

lub

$ **python -m pip install -I** *nazwaPakietu*

gdzie parametr **-I** oznacza wymuszenie reinstalacji.

lub

$ **pip install** *nazwaPakietu* **--force-reinstall**

Od tego momentu będzie działać import:

import *nazwaBiblioteki*

$ **pip list** *(wykaz zainstalowanych modułów)*

$ **pip search pandas** *(wyszukiwanie pakietów zawierających słowo „pandas”)*

Uwaga 1: W wielu przypadkach zainstalowane pakiety będą skojarzone tylko z jedną daną wersją języka Python. Z poziomu innej wersji, mogą być niewidoczne.

Uwaga 2: W systemie Gentoo za obsługę pakietów pip odpowiedzialny jest emerge. Tylko w sytuacji, gdy danej biblioteki nie ma w emerge, można korzystać z pip. Te dwa systemy instalacyjne wykluczają się i nie powinno instalować się danego pakietu podwójnie. Aby skorzystać z pip:

$ **python -m venv** */home/nazwaUżytkownika/.nazwaŚrodowiskaWirtualnego/*

$ **. /home/***nazwaUżytkownika/.nazwaŚrodowiskaWirtualnego***/bin/activate**

(.pip) user@localhost ~ $ **pip install** *nazwaModułu*

(.pip) user@localhost ~ $ *./nazwaSkryptu* (z modułem)

(.pip) user@localhost ~ $ **deactivate**

Ciekawsze moduły:

- **tkinter** - graficzne okienko, które będzie generowane zarówno w Linux, jak i w Windows; z tej biblioteki korzysta moduł **turtle**;
- **pyautogui** - sterowanie myszką, klikanie, screenshots, wszystkie operacje w środowisku graficznym, rozpoznawanie elementów graficznych;
- **matplotlib** - wykresy na podstawie liczb (zob. matplotlib.org/stable/gallery/index.html);

Pakiety:

Pakiet to zbiór modułów (plików z modułami) umieszczonych w jednym folderze. W każdym takim nadrzędnym folderze (bo mogą być subfoldery) znajduje się plik inicjalizacyjny „***__init__.py***”, który jest wykonywany w pierwszej kolejności (może być pusty, ale musi istnieć). Struktura folderów powinna wyglądać następująco:

- */home/użytkownik/skryptyPython/modules*
- */home/użytkownik/skryptyPython/packages*
- */home/użytkownik/skryptyPython/programy*

Wczytanie funkcji z modułu, który jest częścią takiego pakietu, odbywa się następująco:

from sys import path

path.**append(**'*../packages*'**)** *# W systemie Windows: path.append('..\\packages')*

import *nazwaPakietu.nazwaModułu*

lub

import *nazwaPakietu.nazwaPodfolderu.nazwaModułu*

lub

from *nazwaPakietu.nazwaModułu* **import** *nazwaFunkcji*

Można stworzyć **aliasy** do zaimportowanych modułów:
import *nazwaPakietu.nazwaModułu* **as** *abc*
print(abc.nazwaFunkcji()**)**

Enumeracja

Biblioteka „enum" służy do tworzenia spisów / wyliczeń.
from enum **import** IntEnum *# Możemy także użyć zwykłego „Enum" jeśli zależy nam na stringach, a nie na „Integer".*
Menu = **IntEnum(***'Moje_Menu'*, *'Opcja1, Opcja2, Opcja3, Opcja4'***)** *# Jako drugi argument możemy utworzyć także listę lub słownik.*
wybor = **int(**input(„Podaj opcję:")**)** *# Użytkownik podaje numer, a nie „Opcja1"!*
if (wybor == Menu.Opcja1):
    instrukcja

Debugowanie

W edytorze „*Mu*" klikamy w „*Debuguj*", a następnie linijka po linijce klikamy w „*Przekrocz*".

Listy (*aka* tablice, *mutable*)

Info: W przeciwieństwie do tablic z innych języków programowania, listy mogą zawierać ***różne*** *typy danych i dynamicznie zwiększają swoją objętość w pamięci bez wcześniejszych deklaracji (jak w C++ lub jak w SQL). Listy indeksowane są od 0 i można je zmieniać (są „mutable"). Poniżej, przykład listy robionej „z palca":*

Zmienna1 = „*bla bla bla*"

mojaLista1 = [12, „*Jakiś tekst*", Zmienna1]

print(mojaLista1)

 # Zamiana stringa na listę (ale nadal zawierającą stringi):

liczby = "*11, 12, 13*"

lista = **list(**liczby.**split(**", "**))**

 # Zamiana listy zawierającej stringi na listę zawierającą liczby całkowice:

integerLista = **list(map(**int, stringLista**))**

lub

integerLista = [int(kazdyElement) for kazdyElement in lista]

 # Kopiowanie listy:

mojaLista2 = [1, 2, 3]

mojaLista3 = mojaLista2 *# Dwie zmienne wskazują na tę samą listę*

mojaLista3 = mojaLista2[:] *# Utworzenie niezależnej kopii listy*

Uwaga: Listy mogą być dwuwymiarowe (jak szachownica) i nazywane są wtedy matrix. Reprezentowane są wtedy przez dwa indeksy, np. lista[3] [7] (trzeci rząd, siódme miejsce). Przydatne np. do cogodzinnych pomiarów przez 7 dni w tygodniu. Można także tworzyć trójwymiarowe listy (a także n-wymiarowe).

 # Wypisujemy sumę wartości w listy oraz najmniejszy element i największy:

print(**sum**(mojaLista2))

print(**min**(mojaLista2))

print(**max**(mojaLista2))

 # Ładne pojedyncze wypisywanie elementów listy:

for kazdyElement **in** mojaLista1:

    print(kazdyElement)

Brzydki (niezalecany) sposób:

for kazdyElement **in** range(len(mojaLista1)):

    print(mojaLista1[kazdyElement])

 # Wypisanie wybranych elementów z listy:

print(mojaLista1[2]) *# Wypisuje trzeci element*

print(mojaLista1[-1]) *# Wypisuje pierwszy element od końca*

Uwaga: Indeksy mogą być także ujemne, np. lista[-1] oznacza pierwszy element od końca.

 # Podmiana elementów w listy:

mojaLista1[3] = nowaWartosc

 # Mnożenie elementów w listy:

print(4 * nazwaListy) *# Ilość elementów w liście zostanie pomnożona*

 # Dodawanie elementów do początku listy:

nazwaListy = [4, 5, 19] + nazwaListy *# Elementy zostaną dodane do początku listy*

 # Dodawanie jednego elementu do końca listy (nie można dodać kilku elementów). Tym pojedynczym elementem może być także cała lista, funkcja lub metoda:

nazwaListy.**append**(32)

 # Dodawanie wielu elementów do końca listy (rozszerzanie listy poprzez dołączenie innej listy):

nazwaListy.**extend**([7, 8, 9, 10])

 # Dodawanie elementu na konkretne miejsce:

nazwaListy.**insert**(2, *„jakiśString”*) *# Dodanie stringa na 3 pozycję*

 # Na jakiej pozycji listy jest dany element:

print(nazwaListy.**index**(*„jakiśString”*)) *# Zwraca numer indeksu, pod którym funkcjonuje element*

 # Sprawdzanie długości listy:

```
print(len(nazwaListy)) # Skrót od „length”
# Liczenie wybranych elementów w listy:
nazwaListy = [2, 5, 23, 2, 5, 1]
print(nazwaListy.count(5)) # Wypisze, ile razy liczba 5 występuje w listy
# Ile razy dany element wystąpił w liście, wypisze w formie słownika:
nazwaListy = [‘Adam’, ‘Agnieszka’, ‘Bartek’, ‘Romek’, ‘Adam’]
from collections import Counter
wykaz = Counter(nazwaListy)
print(wykaz) # Zwróci: Counter({‘Adam’ : 2, ‘Agnieszka’ : 1, ...})
# Usunięcie jednego lub wielu elementów listy:
del lista[3]
del lista[1:7]
del lista # Usuwa listę, ale nie jej zawartość
# Usuwanie ostatniego elementu:
nazwaListy.pop()
# Usuwanie pierwszego wystąpienia elementu:
nazwaListy.remove(„Franek”) # Usunie pierwszy napotkany element „Franek”
# Czyszczenie całej listy:
nazwaListy.clear()
# Praca na kopii listy (aby uchronić oryginalną przed zmianami). Kopia płytka (cała lista ma inne ID niż oryginał, ale elementy tej listy nie):
nazwaListy.copy()
list(nazwaListy)        # Kopia płytka, inny sposób (niezalecany)
nazwaListy = list(krotka)    # Zamiana krotki na listę
nazwaListy[:]           # Kopia płytka, jeszcze inny sposób (niezalecany)
import copy             # Wczytujemy bibliotekę potrzebną do kopiowania głębokiego
copy.deepcopy(nazwaListy) # Kopia głęboka (elementy listy są nienaruszalne)

# Sortowanie elementów listy (domyślnie jest rosnąco). Polecenie zmienia pierwotną listę:
nazwaListy.sort()
nazwaListy.sort(reverse=True) # Sortowanie malejąco
# Sortowanie na kopii listy (pierwotna lista nie jest zmieniana):
kopiaListy = sorted(pierwotnaLista)
# Można także odwrócić kolejność elementów (ale bez sortowania):
```

```
nazwaListy.reverse()


# Wypisywanie określonej liczby elementów listy:
print(nazwaListy[0:6]) # Wypisze 6 pierwszych elementów
print(nazwaListy[0:6:2]) # Wypisz co drugi element z zakresu 0-6
# Sumowanie elementów wewnątrz listy:
sum(nazwaListy)
# Sprawdzanie, czy element jest w listy:
print(„jakaśNazwa” in nazwaListy) # Zwraca wartość True lub False; także można zastosować „not in”;
if („jakaśNazwa” in nazwaListy):
    print(„Element znajduje się w listy”)
if („jakaśNazwa” not in nazwaListy):
    print(„Element nie znajduje się w listy”)
# A teraz lista z danymi wczytywanymi z pliku:
with open(„plik-z-danymi.txt”) as plik:
    zewnetrzneDane = plik.readlines()
# Wczytał dane z pliku do automatycznie utworzone listy, dane są jednak typu „string”:
zewnetrzneDaneIntiger = []
for liczby in dane:
    zewnetrzneDaneIntiger.append(int(liczby))
print(f”Dane z pliku zewnętrznego: {zewnetrzneDaneIntiger}”)
# Aby odwrócić kolejność wczytywanych liczb:
print(f”Dane z pliku zewnętrznego: {zewnetrzneDaneIntiger[::-1]}”)
print(nazwaListy[:]) # Aby wyświetlić wszystkie elementy listy.
print(nazwaListy[1:]) # Aby wyświetlić wszystkie elementy listy, oprócz pierwszego.
print(nazwaListy[:-1]) # Aby wyświetlić wszystkie elementy listy, oprócz ostatniego.
print(nazwaListy[2:6]) # Aby wyświetlić elementy listy od 2 do 6 (ale nie włącznie z 6).
```

Zagnieżdżanie list:

```
listaUczestnikow = [
                        [‘Adam’, 23, ‘mężczyzna’],
                        [‘Andrzej, 43, ‘mężczyzna’],
                        [‘Agnieszka’, 41, ‘kobieta’]
                    ]
print(listaUczestnikow[1][0]) # Wypisze „Andrzej”
listaUczestnikow[0][1] = 24 # Poprawiamy wiek Adama na 24 lata
```

Uwaga: Jako zagnieżdżenie listy możemy dodać krotkę lub słownik.

Sumowanie elementów listy

Naszym zadaniem jest wygenerowanie listy z kolejnymi liczbami i zsumowanie tych elementów:

```
# Pierwszy sposób:
def sumuj(liczba):
    suma = 0
    for liczba in range(1,liczba+1):
        suma = suma + liczba
    return suma
podanaLiczba = int(input("Podaj liczbę dodatnią: "))
print(sumuj(podanaLiczba))

# Drugi sposób z listą:
liczbaUzytkownika = int(input("Wprowadź liczbę dodatnią: "))
lista = []
for kazdyElement in range(0, liczbaUzytkownika):
    kazdyElement += 1
    lista.append(kazdyElement)
print(sum(lista))

# Trzeci sposób z generatorem obiektu:
def sumuj(liczba):
    return sum([liczba for liczba in range(1,liczba+1)])
print(sumuj(10))

# Czwarty sposób z wyrażeniem listowym:
liczbaUzytkownika = int(input("Wprowadź liczbę dodatnią:"))
lista = [kazdyElement += 1
    for kazdyElement in range(0, liczbaUzytkownika)
    ]

# Piąty sposób. Jeśli w tablicy jest ciąg arytmetyczny, problem można rozwiązać za pomocą matematyki. Dodajemy do siebie pierwszą i ostatnią liczbę, dzielimy przez dwa i mnożymy przez ilość elementów listy: (1 + 5) / 2 * 5 = 15. Jest to najszybszy sposób ze wszystkich:
lista = [1, 2, 3, 4, 5]
def sumuj(liczba):
    return (1 + liczba) / 2 * liczba
print(sumuj(10))
```

Wyrażenia listowe (*List comprehension*)

Jeśli chcemy zamienić jedną listę na drugą (np. spotęgować każdą z liczb), możemy zamiast typowej pętli „*for*” użyć wyrażenia listownego (jest szybsze! zwane jest także „listą składaną”):

```
nowaLista = [kazdyElement ** 2 for kazdyElement in staraLista]
```

Powyższe można umieścić w kilku liniach:

```
# Co chcemy zrobić? Każdy element w liście podnieść do potęgi:
nowaLista = [kazdyElement ** 2
```

```
        # Na czym chcemy to zrobić? Skąd pobrać elementy?
        for kazdyElement in staraLista
        ]
```

Inny przykład (chcemy z listy wybrać tylko liczby parzyste):

```
nowaLista = [kazdyElement
        for kazdyElement in staraLista
        if (kazdyElemeny % 2 == 0)
        ]
```

Uwaga: Wyrażenia listowe służą do tworzenia / generowania list (to jest ich główne przeznaczenie). Mogą zawierać tylko jedną instrukcję, ale wiele warunków. Jeśli jest potrzeba wykonania wielu operacji (wielu instrukcji) na każdym elemencie listy, można w ostateczności wykorzystać zewnętrzną funkcję:

```
[nazwaFunkcji(kazdyElement) for kazdyElement in lista]
```

Wyrażenia warunkowe

```
for kazdyElement in range(10):
    mojaLista.append('Parzysta' if kazdyElement % 2 == 0 else 'Nieparzysta')
# Jeśli liczba parzysta, zwraca „Parzysta”...
```

Jeszcze krócej:

```
mojaLista = ['Parzysta' if kazdyElement % 2 == 0 else 'Nieparzysta' for kazdyElement in range(10)]
```

Krotka (*tuple; immutable*)

Rodzaj listy, ale **niezmiennej** (niemutowalnej [*immutable*]). Można z niej tylko korzystać lub wyświetlać. Elementy krotki są indeksowane. Może zawierać dane różnego typu. Należy jej użyć zawsze wtedy, gdy masz pewność, że nie będzie zmieniana. Używa mniej zasobów niż lista.

```
krotka = (1, 2, 5, 3) # Tworzenie krotki

krotka = 1, 2, 5, 3 # Taki zapis też jest możliwy, choć rzadziej się go używa

print(krotka[0]) # Wypisuje pierwszy element

del krotka # Usuwa krotkę

krotka = tuple(lista) # Zamieniamy listę na krotkę


a = krotka + (20, 23) # Można dodawać

b = krotka * 3 # Można mnożyć

len(krotka) # Długość krotki
```

Uwaga: Podobnie jak w przypadku list, możemy krotki zagnieżdżać (mieć krotkę w krotce).

Słownik (dictionary)

Pojemnik na dane funkcjonujące w parach na zasadzie „*klucz: wartość*”. Dane nie są ponumerowane (nie mają indeksów), choć są uporządkowane (ale dopiero od wersji 3.7). Klucze są unikalne i nie powinny się powtarzać (mogą być liczbami, stringami lub innymi typami), np.

uczniowie = {‘klucz1’ : ‘*Jan Kowalski*’, ‘klucz2’: ‘*Krzysztof Nowak*’, ‘klucz3’ : ‘*Agnieszka Barska*’}
uczniowie[klucz] = ‘*Adam Nowak*’ *# Krzysztofa zamieniamy na Adama.*
uczniowie[klucz] = ‘*Anna Mazurczyk*’ *# Dodanie elementu do słownika*
uczniowie.**update(**{klucz: ‘*Jessika Lambada*’, innyKlucz: ‘*Brian Kowalczyk*’}**)** *# Dodanie wielu elementów*
del(uczniowie[klucz]**)** *# Usuwanie danego klucza (wraz z wartością, bo wartości nie mogą istnieć bez kluczy)*
uczniowie.**pop(**klucz**)** *# Usunięcie danego klucza*
uczniowie.**popitem()** *# Usunięcie ostatniego klucza*
len(uczniowie**)** *# Ile kluczy jest w słowniku*
print(uczniowie) *# Wypisuje cały słownik*
print(uczniowie[klucz]) *# Wypisuje wartość klucza*
print(uczniowie.**get(**klucz**)**) *# Wypisuje wartość klucza (inny sposób)*
nazwaSlownika.**clear()** *# Usuwa wszystkie elementy słownika*
nowySłownik = starySłownik.**copy()** *# Kopiowanie całego słownika*
nazwaSlownika = **dict(**nazwaKrotki**)** *# Zamiana krotki na słownik. Krotka w tym przypadku musi być zagnieżdżona: ((‘klucz1’, ‘wartość1’), (‘klucz2’, ‘wartość2’)).*
for klucz **in** nazwaSlownika:
    print(klucz, nazwaSlownika[klucz]) *# Wypisuje w każdej linii klucz wraz z zawartością*
Inny sposób na wykorzystanie pętli for:
for klucz **in** nazwaSlownika.**keys()**:
    print(klucz, "→", nazwaSlownika[klucz]**)**
Sposób na sortowanie:
for klucz **in sorted(**nazwaSlownika.**keys())**:
    print(klucz, "→", nazwaSlownika[klucz]**)**
Jeszcze inny sposób z pętlą for:
for klucz, wartosc **in** nazwaSlownika.**items()**:
    print(klucz, "→", wartosc**)**
Prezentacja samych wartości (bez kluczy):
for wartosc **in** nazwaSlownika.**values()**:
    print(wartosc**)**
Sprawdzanie, czy klucz jest w słowniku:
if ‘klucz’ **in** nazwaSlownika:
    print(”*Słowo ‘klucz’ występuje w słowniku*”**)**

Wyrażenia słownikowe

Ze zbioru / tablicy / słownika możemy utworzyć osobny słownik. Wyrażenia słownikowe są szybsze niż pętle.

```
zbior = {‘Pies’, ‘Kot’, ‘Pingwin’, ‘Papuga’, ‘Pawian’}
wyrazenieSlownikowe = { element : len(element)
                        for element in zbior
                        }
print(wyrazenieSlownikowe)
{‘Pies’ : 4, ‘Kot’ : 3, ‘Pingwin’ : 7, ‘Papuga’ : 6, ‘Pawian’ : 6} # Ile liter ma każde hasło

# Zamiana stopni Celciusa na Fahrenheita:
celcius = {'t1': -20, 't2': -15, 't3': 0, 't4': 12, 't5': 24}
```

```
fahrenheit = { element : celcius * 1.8 + 32
             # Najpierw zamieniamy w locie słownik na krotkę:
             for element, celcius in celcius.items()
             }
print(fahrenheit)
```

Zbiory (*sets*)

W przeciwieństwie do słowników, zbiory nie mają kluczy.

```
nazwaZbioru = {1, 2, 40, 3, 5} # Uwaga! Elementy nie mogą się powtarzać! Elementy są od razu sortowane (domyślnie rosnąco).
nazwaZbioru.add(4) # Dodawanie elementów
nazwaZbioru.remove(50) # Usuwanie elementu o wartości 50; jeśli go nie ma, to spowoduje error
nazwaZbioru.discard(40) # Usuwanie elementu o wartości 40; jeśli go nie ma, wszystko jest w porządku
set(nazwaListy) # Przekształcamy listę w zbiór
print(nazwaZbioru1 & nazwaZbioru2) # Wypisze elementy, które są wspólne dla obu zbiorów (koniunkcja)
print(nazwaZbioru1 | nazwaZbioru2) # Wypisze wszystkie elementy obu zbiorów (suma), ale bez duplikatów; mogą to także być zbiory składające się z krotek
print(nazwaZbioru1 - nazwaZbioru2) # Wypisze wszystkie elementy zbioru pierwszego, których nie ma w zbiorze drugim (różnica)
print(nazwaZbioru1 ^ nazwaZbioru2) # Wypisze wszystkie elementy oprócz wspólnych (xor - alternatywa wykluczająca)
print(nazwaZbioru1.issubset(nazwaZbioru2)) # Zwróci prawdę lub fałsz zależnie od tego czy cały zbiór 1 jest częścią zbioru 2 czy też nie jest
```

Uwaga: W zbiorach możemy zagnieżdżać, na przykład, krotki.

Wyrażenia zbioru

Możemy ze zbioru utworzyć inny zbiór korzystając z wyrażenia zbioru (jest szybsze niż typowa pętla).

```
zbiorImion = {"arkadiusz", "Wioletta", "karol", "bartłomiej", "Jakub", "Ania"}
nowyZbior = {
          kazdyElement.capitalize() # Poprawiamy pierwszą literę na wielką
          for kazdyElement in zbiorImion
          }
print(nowyZbior)
```

Generatory obiektów

Aby zaoszczędzić pamięć przy przetwarzaniu dużej ilości danych, możemy stworzyć obiekt powiązany z takimi danymi (jakby wskaźnik do danych), którego rozmiar jest mały i stały bez względu na ilość elementów w tablicy na którą wskazuje (ilość danych w zewnętrznym pliku).

```
import sys
print(sys.getsizeof(nazwaListy)) # Sprawdzamy rozmiar listy w pamięci. Dla porównania możemy sprawdzić rozmiar obiektu wskazującego na tę listę.
obiektWskaznik = (kazdyElement
               for kazdyElement in nazwaTablicy
```

)

W dalszej części możemy przetwarzać taki obiekt dokładnie tak jak tablicę, ale szybciej i nie obciążając pamięci (nie wczytując wszystkich danych, lecz te, które akurat potrzebujemy).

Inne zadanie: Wybierz liczby z zakresu 1-100, które są podzielne przez 7, ale nie są podzielne przez 5.

```
obiektLiczby = ( kazdaLiczba                    # Co ma wygenerować
                 for kazdaLiczba in range(1, 101) # Z czego i gdzie
                 if (kazdaLiczba % 7 == 0)        # Jakie warunki musi spełnić to,
                 if (kazdaLiczba % 5 != 0)        # co wygeneruje (bez „and”)
                 )
for kazdaLiczba in obiektLiczby:
    print(kazdaLiczba)
```

Porównanie pojemników na dane:

Pojemniki	Elementy unikalne	Zachowana kolejność	Możliwość zmian elementów	Dodawanie nowych elementów
Lista	✗	✓	✓	✓
Krotka	✗	✓	✗	✗
Słowniki	✓	✗	✓	✓
Zbiory	✓	✗	✗	✓

Uwaga: Pojemniki, które nie wymagają zachowania kolejności elementów, są szybsze.

Porównanie wyrażeń:

Wyrażenie	Przykład	Uwagi
Wyrażenie listowe	[element **for** element **in** lista]	
Wyrażenie słownikowe	{element : operacja(element) **for** element **in** zbior}	
Wyrażenie zbioru	{element **for** element **in** zbior}	Wyszukiwanie elementu w zbiorze jest szybsze niż wyszukiwanie w liście (no chyba że będzie to akurat element z początku listy lub gdy będzie tych elementów mało).
Generator obiektów	(element **for** element **in** lista)	

Obiekty

Programowanie proceduralne oparte jest o funkcje i dane. Programowanie obiektowe oparte jest o obiekty, które możemy sobie swobodnie tworzyć opierając się na klasach (ciasto to obiekt, a przepis na ciasto to klasa). Analogicznie, jak w HTML konkretny element dziedziczy pewne cechy po klasie (ale nie wszystkie), jest konkretny i wyjątkowy.

Obiekt to coś, co ma właściwości, np. jabłko (jest czerwone / zielone, miękkie / twarde, soczyste, zdrowe, duże / małe, itp.). Teoretycznie, przyrównanie do siebie obiektów czyni je sobie równymi: x = y, więc y = x... a nawet: x = 4, y = 4, więc x = y (optymalizacja obiektów: nie tworzy się w danym momencie niepotrzebnie obiektów ponad miarę / potrzeby).

Obiekt to tak naprawdę zmienna, wobec której można stosować różne metody:

```
x = [1, 2, 3, 4]        # Obiekt niezmienny (immutable)
y = x                   # Oba obiekty mają ten sam adres. Zob. id(y)
y.append(5)             # Oba obiekty mają nadal ten sam adres, bo nie ma wyraźnej potrzeby, aby zmienna (obiekt) wystąpiła pod nowym identyfikatorem
y[0] = 20               # Nadal oba obiekty mają tą samą wartość „id”, są immutable

x = 10
y = x                   # Zmienna „y” nadal ma ten sam identyfikator, co „x”
y = 5                   # W tym momencie zmienna „y” ma już nowy identyfikator (jest muttable), gdyż nie jest już równa x
```

Klasy

W programowaniu strukturalnym tworzymy wiele przedmiotów (zmienne) i wykonujemy na nich jakieś czynności. Czynności mają pełnią władzy nad tymi przedmiotami.

W programowaniu obiektowym staramy się naśladować realny świat: obiekty mają swoje stany wewnętrzne, ale mogą też współdzielić pewne cechy z innymi obiektami, mogą na siebie oddziaływać, ale nie całkowicie sobą rządzić.

Klasa to jakby gatunek zwierząt, zawiera obiekty (poszczególne zwierzęta). Klasa jest więc zestawem obiektów. Klasy niższe (podgatunki) mogą dziedziczyć właściwości po klasach wyższego rzędu (także dziedziczyć po kilku klasach jednocześnie), np.

```
class Podrzędna(Nadrzędna):
    def __init__(self): # Konstruktor to coś, co tworzy cechy obiektu
        Nadrzędna.__init__(self)
        self.nowaWłaściwość
```

czyli (ogólnie):

```
class Ssak():
    # tutaj konstruktor tworzy jakieś cechy

class Czlowiek(Ssak):
    # Dziedziczenie cech po klasie Ssak, może odwoływać się do cech, które utworzył konstruktor w klasie Ssak.
```

Można także dziedziczyć po wielu klasach (choć ilość możliwych problemów wtedy wzrasta):

```
class Czlowiek(Ssak, Neandertalczyk):
```

Uwaga: Klasa podrzędna nie powinna zmieniać cech klasy nadrzędnej.

Klasy mogą być także puste (w przeciwieństwie do innych języków, Python to umożliwia):

```
class nazwa():
    pass        # Pusty obiekt (bez deklaracji pól klasy przez konstruktor), można do niego wrzucać różne rzeczy.
```

Wywołanie klasy:

```
nazwaKlasy()    # W istocie jest to wywołanie konstruktora, który jest w tej klasie.
```

Przykład klasy:

```
class Pogoda:
```

```
    # Poniżej inicjujemy metodę (nie obiekt). Obiektem (instancją) staje się dopiero, gdy utworzymy go poza klasą.
    def __init__(self,dzien,temperatura,opad,kategoria_chmur,wielkosc_chmur):
        self.dzien=dzien
        self.temperatura=temperatura
        # Poniżej, właściwość prywatna (_), dostępna tylko wewnątrz klasy,
        # chyba że odwołamy się poprzez: obiekt._Pogoda__opad
        self.__opad=opad
        self.kategoria_chmur=kategoria_chmur
        self.wielkosc_chmur=wielkosc_chmur
```

gdzie __init__ to konstruktor tworzący nową metodę (sam siebie), musi mieć co najmniej jeden parametr: *self* (identyfikuje on obiekt, dla którego metoda została wywołana). Po użyciu takiego konstruktora, można utworzyć z klasy obiekt, a nawet kilka jego kopii (z osobną przestrzenią nazw, wartości zmiennych tych obiektów nie wpływają na siebie). Obiekty są instancjami klasy (można by je także nazwać avatarami, wcieleniami --> likwidacja avatara nie usuwa klasy):

```
nowyObiekt = Pogoda()
nowyObiekt2 = Pogoda()
```

Można odwołać się do właściwości tego obiektu:

```
print(nowyObiekt.temperatura)
```

Jeśli właściwość została poprzedzona w klasie dwoma podkreślnikami „__”, to staje się prywatna (uległa enkapsulacji), nie ma do niej dostępu z zewnątrz klasy. Poniższe zwróci błąd:

```
print(nowyObiekt.__opad)
```

Ale to już zadziała:

```
print(nowyObiekt._Pogoda__opad)
```

Uwaga: Również metody - na tej samej zasadzie - wewnątrz klasy można definiować jako prywatne:

```
def __nazwaMetody(self):
```

Do czego służy „self”?

```
def metoda1():
    instrukcje
def metoda2(self):
    print("cokolwiek")
    self.metoda1 (wywołuje metodę już wcześniej zdefiniowaną w tej klasie)
```

Info: Metoda może być wywołana bez argumentów, ale nie może być zadeklarowana bez żadnych parametrów (musi być co najmniej „self”).

Zakresy dostępu do składowych klasy:

- **public** - element jest dostępny dla wszelkich użyć;
- **private** - element jest niedostępny z zewnątrz - widoczny jest tylko w danej klasie;
- **protected** - element jest częściowo chroniony przed dostępem - widoczny tylko dla danej klasy oraz klas je dziedziczących.

Sposób deklaracji tego dostępu nie odbywa się poprzez słowa kluczowe, ale poprzez użycie podkreślnika:

```
class jakaśKlasa():
  def __init__(self):
    self.publiczna = None
    self._chroniona = None
    self.__prywatna = None
```

```
    def metoda_publiczna(self):
        pass

    def _metoda_chroniona(self):
        pass

    def __metoda_prywatna(self):
        pass
```

Inne funkcje:

- **super()** - służy do uzyskania dostępu do atrybutu nadklasy lub jej metod;

Sprawdzamy, czy obiekt ma atrybut:

```
if hasattr(Pogoda, 'cisnienie'):
```

Stos

Stos to inaczej LIFO (*Last In - First Out*), gdzie położenie danych na stosie zwane jest „*Push*”, a ich zdjęcie ze stosu - „*Pop*”.

Mierzenie wydajności

```
import time
start = time.perf_counter() # Domyślnie zwraca w sekundach
jakiś kod...
end = time.perf_counter()
print(end-start)
```

Można ten kod trochę skrócić tworząc funkcję:

```
def licznikCzasu(start):
    end = time.perf_counter()
    return end - start

start = time.perf_counter()
jakiś kod...
print(licznikCzasu(start))
```

Najlepiej jednak umieścić wszystko w funkcji:

```
# Funkcja, której szybkość chcemy sprawdzić:
def sumuj(liczba):
    return (1 + liczba) / 2 * liczba


# Funkcja sprawdzająca szybkość, ma miejsca na dwa argumenty. Do pierwszego wysyłamy „sumuj” (czyli funkcję, której szybkość chcemy sprawdzić), do drugiego wysyłamy ilość liczb, jakie ma wygenerować.
def wydajnosc(funkcjaSumujaca, iloscLiczb):
```

```
    start = time.perf_counter()
    funkcjaSumujaca(iloscLiczb)
    end = time.perf_counter()
    return end - start
print(wydajnosc(sumuj, 500000))
```

Uwaga: Warto w funkcji badającej wydajność dodać pętlę for, która wykona test określoną ilość razy, aby zwiększyć dokładność testu. Po wykonaniu może ona zsumować kolejne wyniki, lub wyciągnąć z nich średnią.

Pętla for

Powinno się to czytać: „*dla każdego elementu w zakresie 0-8, wykonaj...*”.

```
Info: Celem jest zebranie liczb wprowadzonych przez użytkownika, ich zsumowanie i wyciągnięcie średniej.
mojaLista = []
for nazwaZmiennej in range(8):
    liczba = int( input(f„Wprowadź liczbę numer {nazwaZmiennej + 1}: ”))
mojaLista.append(liczba)
sumaWartosci = sum(mojaLista)
iloscLiczb = len(mojaLista)
print(f„Suma wartości to {sumaWartosci}, a ich średnia to {sumaWartości/iloscLiczb}”)
```

```
# Inny, prosty przykład pętli, która sumuje 4 liczby podane przez użytkownika:
for licznik in range(0, 4):  # Dla każdego elementu „licznik” z zakresu 0-4, wykonaj instrukcję:
# Uwaga: Funkcja „range” nie wymaga pierwszego argumentu, więc może być: range(4). Nie musi to nawet być liczba, może być np. string ‘abc’ (trzy elementy, więc wykona się 3 razy) lub lista [1,3,6,8] - wykona się 4 razy, bo zawiera 4 elementy (zawartość jest bez znaczenia). Funkcja może posiadać także trzeci argument (wartość przyrostu), np. range(1, 100, 2) oznacza, że wypisze tylko liczby parzyste (co drugą liczbę). Możliwe jest także odliczanie do tyłu: range(100, 1, -1).
    liczba = int(input(„Podaj liczbę: ”))
    wynikSumowania += liczba
print(„Wynik sumowania: ”, wynikSumowania)
```

Można zastosować także jednoczesny przebieg na obu elementach:

```
zmienna = slownik{["a","b","c"], [1,2,3]}
for x,y in zmienna:
    jakaśInstrukcja
```

Uwaga: Jeśli chcielibyśmy przerwać wykonywanie pętli z jakiś powodów, możemy użyć instrukcji „***break***” (program przejdzie wtedy do wykonywania kodu znajdującego się poza pętlą). Gdy użyjemy „***continue***”, to przerwane zostaje jedynie bieżące przejście pętli (czyli wszystko, co jest po instrukcji „*continue*”), ale następne przejście pętli jest ponownie wykonywane.

Pętla while

```
# Prosta pętla:
licznik = 0
```

```
while licznik <= 8:
    print(f„Numeruję po kolei: {licznik}”)
    licznik += 1        # Za każdym przebiegiem będzie zwiększać się o 1
```

```
# Pętla sumująca 4 podane przez użytkownika liczby:
wynikSumowania = 0
licznik = 0
while licznik <= 4:
    liczba = int(input(„Podaj liczbę: „))
    wynikSumowania += liczba
    licznik += 1
print(f„Wynik dodawania: {wynikSumowania}„)
lub
print(„Wynik dodawania:”, wynikSumowania)
```

Generowanie losowe

Języki programowania (algorytmy) zawsze generują liczby pseudolosowe (nigdy losowe).

```
import random
```

lub

```
from random import randint,seed
seed(10) # Mając to samo ziarno, zawsze będziemy mieli te same wyniki losowania. Jako ziarno może występować długi skomplikowany ciąg stringa. Zastosowanie funkcji seed() bez wartości pobierze bieżący czas.
zakresLiczb = []
for element in range(1-50):
    zakresLiczb.append(randint(1, 50)) # Losowanie liczby z zakresu 1-50
print(zakresLiczb)
```

Losowanie na zbiorze liczb całkowitych:

```
randrange(0, 24, 2) # Wylosuje jedną liczbę w zakresie od 0 do 23, ale tylko z co drugiej z nich
randint(1, 24) # Losowanie w zakresie od 1 do 24
```

Losowanie na zbiorze liczb rzeczywistych:

```
random() # Losowanie od 0 do 1 (liczby rzeczywiste)
uniform(1.0, 20.0) # Losowanie od 1 do 20 (liczby rzeczywiste)
```

Losowanie z elementów listy:

```
nazwaListy = [„element1”, „element2”, „element3”]
print(random.choice(nazwaListy))
```

Preferencyjne losowanie z elementów listy:

Możemy ustalić, które elementy będą miały większe prawdopodobieństwo wylosowania.

```
nazwaListy = [„element1”, „element2”, „element3”]
print(random.choices(nazwaListy, [2, 1, 1], k = 20))
```

lub procentowo:

```
print(random.choices(nazwaListy, [75, 10, 15], k = 20))
```

Legenda:

[] - określenie wagi (jest to średnia ważona; im wyższa wartość, tym częściej będzie losowana)

k - ilość losowań

Losowa zmiana kolejności elementów:

```
nazwaListy = [„element1”, „element2”, „element3”]
random.shuffle(nazwaListy)
print(nazwaListy)
```

Losowanie Lotto:

Z tego powodu, że wylosowane liczby nie mogą się powtarzać, losowanie musi być unikalne:

```
import random
def wylosujLiczby():
    print(random.sample(range(1, 50), 6)) # Zakres do 50, aby losowało też liczbę 49
wylosujLiczby()
```

Można także losować z listy:

```
random.sample(nazwaListy)
```

Uwaga: Unikalność dotyczy elementów listy, ale nie wartości tych elementów. Jeśli w liście znajdują się dwie takie same wartości (np. dwie liczby 10), będą mogły być one wylosowane.

Losowanie kart:

W tym przypadku, wylosowane karty nie tylko mają być unikatowe, ale także usunięte z talii kart, aby drugi gracz nie mógł ich wylosować.

```
import random
listaKart = ["9", "9", "9", "9",
             "10", "10", "10", "10",
             "Walet", "Walet", "Walet", "Walet",
             "Królowa", "Królowa", "Królowa", "Królowa",
             "Król", "Król", "Król", "Król",
             "As", "As", "As", "As",
             "Joker", "Joker"]
def wylosujKarty():
    random.shuffle(listaKart) # Uwaga: Tasowanie odbywa się na oryginale listy.
    print("\nPotasowana lista kart:\n", listaKart)
    wylosowaneKarty = []
    for kazdyElement in range(5):
        karta = listaKart.pop()
        wylosowaneKarty.append(karta)
    print("\nWylosowane karty to:\n", wylosowaneKarty)
    print("\nPozostała pula kart:\n", listaKart)
wylosujKarty()
```

Operacje na plikach

Parametry otwierania plików (można je ze sobą łączyć):

- **r** - *read* *(zwykłe otwarcie pliku)*
- **r+** - *możliwe jest czytanie i zapisywanie, jeśli plik wcześniej istniał, to go **nie** usunie*
- **w** - *write* *(tworzenie nowego pliku; jeśli istniał, to zastąpi go nowym, czyli nową treścią)*
- **w+** - *możliwe jest czytanie i zapisywanie, jeśli plik wcześniej istniał, to go usunie, a jeśli nie istniał, to go utworzy*
- **a** - *append* *(dopisywanie treści do pliku w miejscu wskaźnika)*

- **a+** - *tryb dopisywania **zawsze** na końcu bez względu, gdzie znajdował się wskaźnik, i jednocześnie czytania od początku lub dowolnego miejsca*
- **b** - *tryb binarny (parametr funkcjonuje tylko w połączeniu z „r/w/a”)*
- **t** - *tryb tekstowy (parametr funkcjonuje tylko w połączeniu z „r/w/a”)*

Zapisanie do pliku:

```
mojPlik = open(”nazwaPliku.txt”, ”w”, encoding=”UTF-8”) # Przypisujemy zmienną do tego otwartego pliku, aby móc się do niego odwoływać i na nim pracować; zmienna ta zwana jest „uchwytem_do_pliku”.
mojPlik.write(”Jakiś tekst”) # Uwaga: Na razie ten tekst jest zapisany w pliku w pamięci RAM, a nie w pliku na dysku.
mojPlik.close() # Zamykanie pliku to tak naprawdę jego zapisywanie na dysku (choć to zależy od tego, czy jest włączone bufforowanie danych). Zwalniamy przy okazji pamięć RAM.
```

Wypisanie zawartości pliku:

```
wiersz = open(‘wiersz.txt’, ‘rt’, encoding=’utf-8’)
print(wiersz.read())
# print(wiersz.read(10)) # Odczytuje pierwsze 10 znaków
# print(wiersz.readline()) # Odczytuje całe linie
# print(wiersz.readlines()) # Odczytuje cały plik
```

Operacje na pliku binarnym:

```
miejsceNaZdjecie = bytearray(2000) # w bajtach
zdjecie = open(‘plik.bin’, ‘wb’)
# readinto() - metoda, która odczytuje dane binarne z pliku i umieszcza je w obiekcie
```

Operacje obarczone ryzykiem dobrze jest wstawić do bloku wyjątków (może być wiele bloków „*try:*” w kodzie, także zagnieżdżonych wewnątrz macierzystego „try:”):

```
try:
    mojPlik = open(”nazwaPliku.txt”, ”w”, encoding=”UTF-8”)
    print(0/0) # Tu wystąpi błąd dzielenia przez 0 i zatrzymanie przetwarzania
    mojPlik.write(”Jakiś tekst”)
finally: # Wykonaj chociaż to, bez względu na możliwe błędy (zawsze zostanie wykonane)
    mojPlik.close()
```

Można także przed otwarciem pliku sprawdzić, czy plik istnieje.

Metoda 1:

```
from os.path import exists      # funkcja exist() z modułu os.path; lub „import os.path”
sprawdzanyPlik = exists(‘nazwaPliku.txt’)   # Zwraca True lub False i taka wartość zostanie przypisana do zmiennej sprawdzanyPlik; jeśli zaimportowaliśmy metodą „import os.path” to podajemy pełną ścieżkę: os.path.exists.
```

Metoda 2:

```
from os.path import isfile
sprawdzanyPlik = isfile('nazwaPliku.txt')
print(sprawdzanyPlik)                       # Zwróci True lub False
```

Metoda 3 (poprzez import klasy *Path* z modułu *pathlib*):

```
from pathlib import Path
sprawdzanyPlik = Path(‘nazwaPliku.txt’)
if sprawdzanyPlik.is_file():                # Zwraca True lub False
    print(‘Plik istnieje’)
```

else:
    print(‘*Plik nie istnieje*’**)**

Uwaga: Wpisując ścieżkę dostępu należy używać znaku **/** (nawet w *Windows*!).

Sprawdzanie typu pliku:

```
# Moduł magic jest modułem zewnętrznym, którego import wymaga wiedzy informatycznej (a więc nie dla zwykłych użytkowników). Z kolei moduł "bs4 / beautyfulsoup" - i tak najpierw otwiera plik, aby go zbadać (więc jest to bez sensu). Wszystko wskazuje na to, że nie ma pewnego sposobu na ustalenie typu i kodowania pliku. Dlatego stosujemy „mimetypes”:
import mimetypes    # Niestety, bazuje na rozszerzeniach plików :-(
# Poniżej, indeks [0] oznacza pierwszy zwracany wyraz, czyli ‘text/plain’:
typPliku = mimetypes.guess_type('nazwaPliku.txt')[0]
# Poniżej, indeks [0:4] oznacza zwracane pierwsze 4 litery, czyli 'text' z pierwotnego 'text/plain'. Stosowane, gdyż inne pliki tekstowe mają postać typu 'text/x-python', itp.
if typPliku[0:4] == 'text':
   print('To jest plik tekstowy.')
else:
   print('To nie jest plik tekstowy.')
```

Otwarcie pliku z jego automatycznym zamknięciem:

```
with open(”nazwaPliku.txt”, ”w”, encoding=”UTF-8”) as zmiennaMojPlik:
    trescPliku = zmiennaMojPlik.read().splitlines() # Jeśli chcemy wyciąć znaki nowej linii (\n), a wszystkie elementy tekstowe pliku umieścić w liście. Uwaga: Jeśli chcemy sprawdzić w jakim kodowaniu otwierany jest plik: zmiennaMojPlik.encoding.
    trescPliku = zmiennaMojPlik.readline() # Czyta pierwszą linię, przechodzi do następnej... i czeka na dalsze instrukcje. Możemy dowiedzieć się, w którym miejscu akurat czeka za pomocą komendy „zmiennaMojPlik.tell()”. Możemy także ustawić focus na dowolne miejsce: „zmiennaMojPlik.seek(0)” (liczba 0 oznacza tutaj początek pliku; liczby oznaczają numer znaku w tekście, brane pod uwagę są również znaki specjalne, np. \n).
    trescPliku = zmiennaMojPlik.readlines() # Czyta wszystkie linie i umieszcza je w liście
    Inny sposób:
    for kazdaLinia in zmiennaMojPlik:
        print(kazdaLinia)
    zmiennaMojPlik.write(”Jakiś tekst”)
```

Uwaga: Teoretycznie plik musi być otwierany za każdym razem, gdy chcę go przeczytać.

Aby dopisać tekst do istniejącego pliku (ale nie kasować już zawartych w nim informacji):

```
with open(”nazwaPliku.txt”, ”w”, encoding=”UTF-8”) as zmiennaMojPlik:
    zmiennaMojPlik.write(”\nJakiś dodany tekst”)
```

Operacje na rejestrze

```
import winreg
konkretnyKlucz = winreg.OpenKey(winreg.HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\CentralProcessor\0\VendorIdentifier)
winreg.CloseKey(konkretnyKlucz)
```

Wyjątki / błędy

Standardowo, po wystąpieniu błędów, Python kończy program i wyświetla numer błędu. Jeśli chcemy coś więcej, inaczej obsłużyć taki błąd, definiujemy obsługę wyjątków (błąd kończy działanie programu, wyjątek niekoniecznie). Prosta forma:

```
try:
    ryzykowna instrukcja 1...
    instrukcja 2...
except:
    print(„Wystąpił błąd") # Instrukcja 2 nie zostanie wykonana
```

Forma bardziej rozbudowana:

```
try:
    ryzykowna instrukcja...
except nazwaBłędu1:
    awaryjna instrukcja...
except (nazwaBłędu2, nazwaBłędu3):
    awaryjna instrukcja...
else:
    instrukcja-gdy-OK
finally:
    instrukcja (wykonana będzie i wtedy gdy jest wyjątek i wtedy gdy nie ma, czyli zawsze)
```

Szczegółowa informacja o błędzie:

```
except AttributeError as bladAtrybutu:
        print('Jest problem z nazwami / atrybutami poleceń. Szczegóły: {}.' .format(bladAtrybutu))
    raise SystemExit
```

Info: Nazwy wyjątków (błędów) to, na przykład, „*FileNotFoundError*", „*IndexError*", „*ZeroDivisionError*", „*ValueError*". Jest ich łącznie 63 i mają strukturę drzewa (wyjątki ogólne mogą zawierać podgrupę bardziej szczegółowych wyjątków). Program zatrzyma się po wystąpieniu pierwszego napotkanego wyjątku (kolejność definiowania wyjątków ma więc znaczenie i powinno przebiegać od szczegółowych do bardziej ogólnych). Wyjątek, który wystąpił wewnątrz funkcji, może być obsłużony w tej funkcji lub przekazany do obsługi na zewnątrz za pomocą „return instrukcja".

Obsługa JSON

Rekordy bazy danych przekształcamy do uniwersalnego formatu odczytywanego w różnych językach programowania.

```
import json
json.dumps(naszeDane, ensure_ascii=False) # Zapis do stringa
json.dump(naszeDane, nazwaPlikuDoZapisu, ensure_ascii=False) # Zapis do pliku *.json. Plik oczywiście musimy wcześniej otworzyć i przypisać do zmiennej (uchwytu / handler):
with open("plik.json", "w", encoding="UTF-8") as plikJSON:
    json.dump(naszeDane, plikJSON, ensure_ascii=False)
```

Info: Znaki z bazy danych domyślnie są przekształcane do formatu ASCII, czyli znaki narodowe są przekształcane do kodu ASCII. Aby tego uniknąć, dajemy „*False*".

Otwieranie pliku JSON do formatu Python (różnice występują, na przykład, na poziomie wielkości znaków):

```
with open(”plik.json”, ”r”, encoding=”UTF-8”) as plikJSON:
    wynik = json.load(plikJSON)
```

Info: Wczytany plik JSON zawiera domyślnie dane w jednej linii. Aby sformatować to do osobnych linii, należy użyć parametru „*indent=4*”, co wstawi do treści pliku znaki \n, jednak funkcja print wyświetli dane w osobnych liniach:

```
dane = json.dump(naszeDane, plikJSON, ensure_ascii=False, indent=4)
print(dane)
```

Alternatywnie, możemy użyć funkcji „*pprint*” (*Pretty Print*):

```
import pprint
pprint.pprint(wynik)
```

Info: Można dodać także parametr „*sort_keys=True*”, co posortuje nam klucze alfabetycznie.

Interakcje z www

Przykładowe zasoby JSON (ale nie w formacie Python):
http://jsonplaceholder.typicode.com

```
import requests
import json
zapytanie = requests.get(”http://jsonplaceholder.typicode.com/todos”) # Jeśli strona istnieje, zwróci kod 200.
zapytanie2 = requests.get(”http://jsonplaceholder.typicode.com/users”)
uzytkownicy = zapytanie2.json()
zadania = json.loads(zapytanie.text)
```

lub:

```
try:
    zadania = zapytanie.json()
except json.decoder.JSONDecodeError:
    print(”Niepoprawny format”)
```

webbrowser.open_new_tab(adres_strony) - otwiera stronę / strony

Przykłady skryptów

Zrobienie screenshota

```
import pyautogui

obrazek = pyautogui.screenshot('screenshot-01.png')
```

Stosowanie zmiennych

```
imie = input(„Podaj swoje imię: ”)

rok_urodzenia = input(„Podaj swój rok urodzenia: ”)

# Uwaga: Domyślnie zmienne przybierają typ „string”. Dlatego w poniższym poleceniu musimy wymusić typ „int”.

wiek = 2023 - int(rok_urodzenia)

print(f„Cześć {imię}, masz {wiek} lat.”)
```

```
if wiek > 17:
        print(f„{imie}, jesteś osobą pełnoletnią”)
else:
        print(f„{imie}, jesteś osobą niepełnoletnią”)
```

Można także sprawdzić, jakiego typu jest zmienna:

type(nazwaZmiennej)

Można też hurtowo definiować zmienne:

a, **b**, **c** = 2, „*Agnieszka*”, True

lub

a = **b** = **c** = 43

Czy liczba jest dodatnia i parzysta?

```
liczba = int(input("Podaj liczbę: "))
wartoscModulo = liczba % 2 # Jeśli jest parzysta, reszta z dzielenia będzie 0
if (wartoscModulo == 0 and liczba > 1):
    print("Liczba", liczba, "jest parzysta i dodatnia!")
elif (liczba < 0):
    print("Liczba ", liczba, "nie jest dodatnia.")
else:
    print("Liczba", liczba, "jest nieparzysta.")
```

Sumowanie 3 liczb dodatnich i parzystych

```
licznik = 0
wynik = 0
while licznik < 3:
    liczba = int(input("Podaj parzystą liczbę dodatnią: "))
    wartoscModulo = liczba % 2 # Jeśli jest parzysta, reszta z dzielenia będzie 0
    if (wartoscModulo == 0 and liczba > 1):
        wynik += liczba
        licznik += 1
    else:
        print("Miałeś podać dodatnią liczbę parzystą!")
print("Suma liczb: ", wynik)
```

Odliczanie

```
# Liczenie do 5:
import time
for x in range(0,5):
        print(x)
        time.sleep(1)
print("Szukam!")
```

Zgadnij liczbę

```
# ZGADNIJ LICZBĘ
print("ZGADNIJ LICZBĘ...")
sekretnaLiczba = 40
liczbaUzytkownika = 0
while liczbaUzytkownika != sekretnaLiczba:
   liczbaUzytkownika = int(input("Podaj liczbę: "))
   if (liczbaUzytkownika == sekretnaLiczba):
      print("Brawo! Zgadłeś! Tajna liczba to: ", liczbaUzytkownika, ".")
   elif (liczbaUzytkownika > sekretnaLiczba):
      print("Podałeś za dużą liczbę! Spróbuj jeszcze raz!")
   else:
      print("Podałeś za małą liczbę. Spróbuj jeszcze raz!")
```

Silnia

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

Kobieta / mężczyzna

```
imię = „Anna”
ostatniaLitera = imie[-1]  # Numerowanie stringów od tyłu: -1 to pierwszy element, -2 to drugi
```

Pierwiastkowanie sześcienne

```
# Ile liczb z zakresu 1-1000 daje wynik całkowity podczas pierwiastkowania sześciennego?
```

```
listaLiczbPotegi3 = []

liczba = 1

for kazdaLiczba in range(1,1000):

   liczba=kazdaLiczba**3

   if(liczba>1000):

      break

   listaLiczbPotegi3.append(liczba)

print("W zakresie 1-1000, występuje ", len(listaLiczbPotegi3), " liczb, które dają wynik całkowity podczas operacji pierwiastkowania sześciennego i są to:")

print(listaLiczbPotegi3)
```

Tworzenie gier

Można do tego celu wykorzystać bibliotekę *Pygame Zero*. W edytorze „*Mu*” włączamy tryb gier. Zdjęcia tła i postaci kopiujemy do folderu projektu (tam, gdzie jest zapisany plik programu).

```
# Wielkość liter w predefiniowanej zmiennej, ma znaczenie:
TITLE = „Napis w tytule okna”
WIDTH = 800
HEIGHT = 600
# Wykorzystujemy predefiniowaną funkcję „draw”, w ramach której podajemy tło gry, oraz początkowe współrzędne tego tła (lewy górny róg). Słowo „blit” można tłumaczyć jako „bit block transfer” / „block image transferrer”, „szybkie przesuwanie mapy bitowej” (animacja) i pochodzi z roku 1974 od Xerox; postać w grze jest odseparowana od tła (podobna technologia to „sprite” [duszki], ale ona obsługiwana jest sprzętowo przez chip „blitter”, który odciąża procesor). Możliwa jest też technika, że przed wyświetleniem, dwie bitmapy (np. postać i tło) łączone są w jedną:
duszek = Actor(„duszek.png”)
        def draw():
                screen.blit(„tlo.jpg”, (0, 0))
                duszek.draw()


def update():
        if keyboard.d:
        # Jeśli klawisz „d” zostanie naciśnięty, to powiększ wartość współrzędnej „x” o 5:
                duszek.x += 5
        if keyboard.a:
                duszek.x -= 5
        if keyboard.w:
                duszek.y -= 5
        if keyboard.s:
                duszek.y += 5
        # Jeśli duszek dojdzie do krawędzi okna (800px, 600px), wtedy odbije się od niego:
        if duszek.x > 800:
                 duszek.x -= 10
        if duszek.x < 0:
                duszek.x += 10
```

```
    if duszek.y > 600:
        duszek.y -= 10
    if duszek.y < 0:
        duszek.y += 10
```

Kompilacja do *.exe

Skrypt skompilowany w *Linux* - będzie działał tylko w *Linux*, a skompilowany w *Windows* - będzie działał tylko w *Windows*. Procedura dla *Linux*:

$ **pip install pyinstaller --user**

$ **~/.local/bin/pyinstaller --onefile** *plik.py*

$ **./***nazwaPliku* (uruchamiamy utworzony plik)

Procedura dla Windows (wątpliwa):

C:\> **pip install pyinstaller**

C:\> **pyinstaller --onefile** *plik.py*

Uwaga: Utworzony plik będzie znajdował się w folderze „*Dist*”. Program *Windows Defender* zinterpretuje go jako wirusa (*Trojan*) i nie pozwoli go uruchomić. Wyjściem z sytuacji jest użycie polecenia „*pyinstaller plik.py*” (ale wtedy do działania będzie potrzebował innych utworzonych plików) lub użycie pakietu „*Nuitka*” i skompilowanie naszego skryptu z użyciem kompilatora *migGW64* (znanego z C++):

C:\> **pip install nuitka**

C:\> **nuitka --mingw64** *plik.py*

Problemy i tipsy

ModuleNotFoundError: No module named 'tkinter'

Rozwiązanie:

USE=”tk” emerge =dev-lang/python-3.9.*

Uwaga: Składnik „*tk*” (*tkinter*), wymagany przez moduł *matplotlib*, przypisany jest do konkretnej wersji Pythona i nie będzie widoczny jako moduł w innej wersji.

pip wywołuje gnome-keyring

Rozwiązanie:

$ **pip uninstall keyring**

pip uninstall keyring

Deinstalacja *keyring* w konkretnej wersji pakietu:

python3.8 -m pip uninstall keyring

Zasoby i materiały

- http://edube.org - certyfikacje;
-

Ostatnia aktualizacja: 3 kwietnia 2023.