

Python: podstawy programowania



Dzień 2 | v. 1.1.1

Listy i krotki

Nawigowanie po liście

Jak pobrać dowolny element listy?

Czasami potrzebujemy tylko konkretnego elementu listy.

Możemy go uzyskać w następujący sposób:

`lista[indeks]`

Indeks **dodatni** oznacza, że bierzemy kolejny znak z łańcucha

Indeks **ujemny** oznacza, że bierzemy kolejny znak **od końca** łańcucha.

Przykład:

```
bbt = ["Leonard", "Sheldon", "Penny",  
       "Howard", "Raj", "Bernardette", "Amy"]
```

```
bbt[0] # "Leonard"
```

```
bbt[4] # "Raj"
```

```
bbt[-1] # "Amy"
```

```
bbt[-3] # "Raj"
```

Zaawansowane nawigowanie po liście

Jak pobrać fragment listy?

Czasami potrzebujemy tylko kawałka listy, nie całości. W takiej sytuacji możemy poprosić Pythona o podanie dowolnego fragmentu listy:

`lista[początek:koniec]`

Zakres podany w tym miejscu, to zbiór prawostronnie otwarty!

`lista[:numer]` oznacza, że szukamy od początku stringa do parametru **numer**.

`lista[numer:]` oznacza, że szukamy od parametru **numer** do końca stringa.

Indeks ujemny oznacza przeszukiwanie od końca łańcucha tekstowego.

Przykład:

```
bbt = ["Leonard", "Sheldon", "Penny",  
       "Howard", "Raj", "Bernardette", "Amy"]  
  
bbt[0:1] # ['Leonard']  
bbt[2:5] # ['Penny', 'Howard', 'Raj']  
bbt[4:]  # ['Raj', 'Bernardette', 'Amy']  
bbt[:2]  # ['Leonard', 'Sheldon']  
bbt[:-4] # ['Leonard', 'Sheldon', 'Penny']  
bbt[-3:] # ['Raj', 'Bernardette', 'Amy']  
bbt[2:-2] # ['Penny', 'Howard', 'Raj']
```

Zaawansowane nawigowanie po liście

Jak pobrać co któryś element listy?

Czasami chcemy pobrać co drugi, co trzeci, itp. elementy listy.

W takiej sytuacji możemy poprosić Pythona o podanie dowolnego listy (fragmentu albo całości) ze skokiem:

`lista[początek:koniec:skok]`

Przykład:

```
bbt = ["Leonard", "Sheldon", "Penny",  
       "Howard", "Raj", "Bernardette", "Amy"]  
  
bbt[0:6:2] # ['Leonard', 'Penny', 'Raj']  
bbt[::3] # ['Leonard', 'Howard', 'Amy']  
bbt[:-2:2] # ['Leonard', 'Penny', 'Raj']  
bbt[2::2] # ['Penny', 'Raj', 'Amy']  
bbt[4:2:-1] # ['Raj', 'Howard']  
bbt[:-4:-1] # ['Amy', 'Bernardette', 'Raj']  
bbt[::-1] # ['Amy', 'Bernardette', 'Raj',  
            'Howard', 'Penny', 'Sheldon', 'Leonard']
```

Listy i typy danych

Typy danych

Listy mogą przechowywać różne typy danych:

- liczby całkowite (integer),
- liczby długie (long),
- liczby zmiennoprzecinkowe (float),
- liczby zespolone (complex),
- stringi,
- słowniki,
- wartości logiczne – boolean,
- inne listy.

```
mix_types = ["Ala", 23, True]
```

Indeksy (klucze) list rozpoczynają się od 0.

```
print(mix_types[0]) # wypisze "Ala"  
print(mix_types[1]) # wypisze 23  
print(mix_types[2]) # wypisze True
```

Aby pobrać wielkość tablicy, korzystamy z funkcji `len()`

```
print(len(mix_types)) # 3
```

Modyfikacja listy

Listę można zmieniać!

Listy można modyfikować:

- zmieniać istniejące elementy,
- dodawać nowe,
- usuwać istniejące z elementy z listy.

Modyfikacja:

```
lista[index] = nowa_wartosc
```

Dodanie nowej wartości:

```
lista.append(nowa_wartosc)
```

Usunięcie elementu listy:

```
lista.pop(index)
```

```
jedi = ["Anakin", "Qui-Gon", "Obi-Wan"]  
  
jedi.pop(1) # ["Anakin", "Obi-Wan"]  
  
jedi[0] = "Vader" # ["Vader", "Obi-Wan"]  
  
jedi.append("Luke") # ["Vader", "Obi-Wan",  
"Luke"]
```


Listy i pętle

Listy i pętle

Jeżeli chcielibyśmy w prosty sposób wypisać wszystkie elementy listy w konsoli, możemy użyć pętli. Zobacz przykład obok.

Taka lista jest zestawem danych, po którym można iterować.

Możemy użyć zatem pętli **for**, aby to zrobić.

```
nexus6 = ["Roy Batty", "Pris", "Zhora",  
"Leon"]
```

```
for android in nexus6:  
    print(android)
```

Wynik:

```
Roy Batty  
Pris  
Zhora  
Leon
```


Listy i pętle

Listy i pętle

```
nexus6 = ["Roy Batty", "Pris", "Zhora",  
"Leon"]
```

```
for android in nexus6[2:]:  
    print(android)
```

Wynik:

Zhora
Leon

```
for android in nexus6[:-1:2]:  
    print(android)
```

Wynik:

Roy Batty
Zhora

```
for android in nexus6[::-1]:  
    print(android)
```

Wynik:

Leon
Zhora
Pris
Roy Batty

Lista składana

Lista składana to specyficzny rodzaj tworzenia listy.

Wyobraź sobie, że chcesz utworzyć listę kolejnych liczb parzystych od 0 do 20.

Kod:

```
my_list = []  
for i in range(0, 11):  
    my_list.append(i*2)  
  
print(my_list)
```

Wynik:

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

Lista składana

Można zrobić to znacznie prościej, używając **listy składanej**:

Kod:


```
my_list = [2 * i for i in range(0, 11)]  
  
print(my_list)
```

Wynik:

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Lista składana

```
my_list = [2 * i for i in range(0, 11)]
```



`2 * i` to wynik
każdej iteracji pętli



`for...` to pętla, która iteruje
po jakiejś kolekcji

Lista składana

W listach składanych możemy również używać warunków:

Kod:


```
[i for i in range(0, 18) if i % 3 == 0]
```

Wynik:

```
[0, 3, 6, 9, 12, 15]
```

Lista składana


```
my_list = [i for i in range(0, 18) if i % 3 == 0]
```



`i` to wynik
każdej iteracji pętli



`for...` to pętla, która iteruje
po jakiejś kolekcji



`if...` to warunek, który musi być
spełniony, aby dodać wynik
do listy

Krotka (tuple)

Czym są krotki?

Krotki (ang. **tuple**), są bardzo podobne do list.

Definiuje się je bardzo podobnie do list, ale zamiast nawiasu kwadratowego używa się okrągłego:

```
my_numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Formalnie jedyną różnicą między krotką, a listą jest to, że krotki **nie można modyfikować!**

Poza tą niedogodnością wszystko działa tak samo jak w listach.

Krotki są szybsze od list!

```
hogwarts = ("Hufflepuff", "Gryffindor",  
            "Ravenclaw", "Slytherin")
```

```
hogwarts[1] # 'Gryffindor'  
hogwarts[1] = "Hacked by Slytherin!"
```

```
Traceback (most recent call last):  
  File "<pyshell#112>", line 1, in <module>  
    hogwarts[1] = "Hacked by Slytherin!"  
TypeError: 'tuple' object does not support item assignment
```

Więcej nt. różnic między krotkami, a listami znajdziesz tutaj:

<http://news.e-scribe.com/397>

Krotki vs listy

W każdej chwili możesz zmienić typ

Jeśli chcesz szybko zmienić typ z krotki na listę, możesz użyć funkcji **list()**:

```
harrys_tuple = ("Ginny", "Gryffindor",  
"Voldemort", "quidditch")  
harrys_list = list(harrys_tuple)  
print(harrys_list)
```

Wynik:

```
["Ginny", "Gryffindor", "Voldemort",  
"quidditch"]
```

Jeśli chcesz szybko zmienić typ z listy na krotkę, możesz użyć funkcji **tuple()**:

```
ginnys_brothers = ["Bill", "Charlie",  
"Percy", "Fred", "George", "Ron"]  
ginnys_tuple = tuple(ginnys_brothers)  
print(ginnys_tuple)
```

Wynik:

```
("Bill", "Charlie", "Percy", "Fred",  
"George", "Ron")
```

Słowniki

Słownik (dictionary)

Czym są słowniki?


Słownik (ang. **dictionary**) to specyficzna i bardzo użyteczna struktura danych.

Jest to zestaw par: **klucz** – **wartość**.

Klucz to zazwyczaj łańcuch tekstowy (choć niekoniecznie, można używać innych typów danych)

Wartość, to dowolny typ danych.

```
star_wars = {  
    "episode": 4,  
    "title": "A New Hope",  
    "characters": ["Luke", "Leia",  
        "Han Solo", "Obi-Wan", "Chewbacca",  
        "Vader", "C3PO", "R2D2"]  
}
```

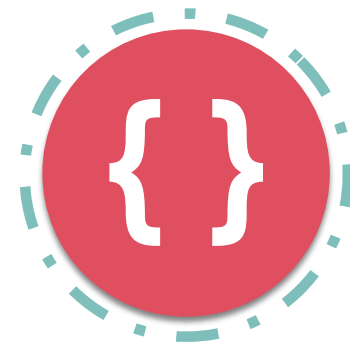


Wartością może być wszystko:
nawet lista albo inny słownik

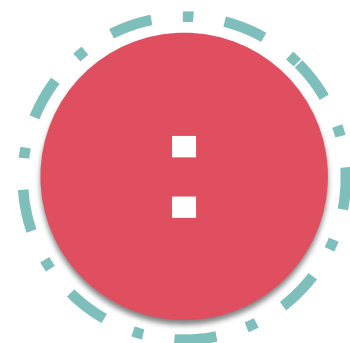
Słownik (dictionary)

Definicja słownika

Słownik rozpoczynamy i kończymy nawiasem klamrowym:



Klucz oddzielamy od wartości dwukropkiem:



Poszczególne hasła słownika oddzielamy przecinkiem:



```
star_wars = {  
    "episode": 4,  
  
    "title": "A New Hope",  
  
    "characters": ["Luke", "Leia",  
                  "Han Solo", "Obi-Wan", "Chewbacca",  
                  "Vader", "C3PO", "R2D2"]  
}
```

Słownik (dictionary)

Pobieranie danych ze słownika:

Aby pobrać dane z naszego słownika, musimy podać:

- nazwę zmiennej, w której go przechowujemy,
- nazwę klucza (w kwadratowych nawiasach).

Przykład:

```
star_wars["episode"] # 4
star_wars["title"] # "A New Hope"
star_wars["characters"] # ["Luke",
"Leia"... ]
star_wars["characters"][2] # "Han Solo"
star_wars["year"] # KeyError!
```

```
star_wars = {
    "episode": 4,

    "title": "A New Hope",

    "characters": ["Luke", "Leia",
        "Han Solo", "Obi-Wan", "Chewbacca",
        "Vader", "C3PO", "R2D2"]
}
```

Słownik (dictionary)

Dodawanie danych do istniejącego słownika:

Oczywiście gotowy słownik możemy rozbudować o nowe hasła:

```
star_wars["year"] = 1976
```

← Ta data jest niepoprawna, ale zaraz ją poprawimy! :-)

Ta wartość zostanie dodana do słownika i będziemy mogli z ni skorzystać ej później.

```
print(star_wars["year"])
```

Wynik:

1976

```
star_wars = {  
    "episode": 4,  
  
    "title": "A New Hope",  
  
    "characters": ["Luke", "Leia",  
                  "Han Solo", "Obi-Wan", "Chewbacca",  
                  "Vader", "C3PO", "R2D2"],  
  
    "year": 1976  
}
```

← Klucz „year” i wartość „1976” właśnie się pojawiły!

Słownik (dictionary)

Modyfikacja danych:

Ojej! Data powstania Gwiezdnych Wojen jest błędna!

Możemy ją zmienić w bardzo prosty sposób:

```
star_wars["year"] = 1977  
print(star_wars["year"])
```

Wynik:

1977

```
star_wars = {  
    "episode": 4,  
  
    "title": "A New Hope",  
  
    "characters": ["Luke", "Leia",  
                  "Han Solo", "Obi-Wan", "Chewbacca",  
                  "Vader", "C3PO", "R2D2"],  
  
    "year": 1977  
}
```


Słownik (dictionary)

Usuwanie danych:

A co, jeśli chcemy usunąć jakąś daną ze słownika?

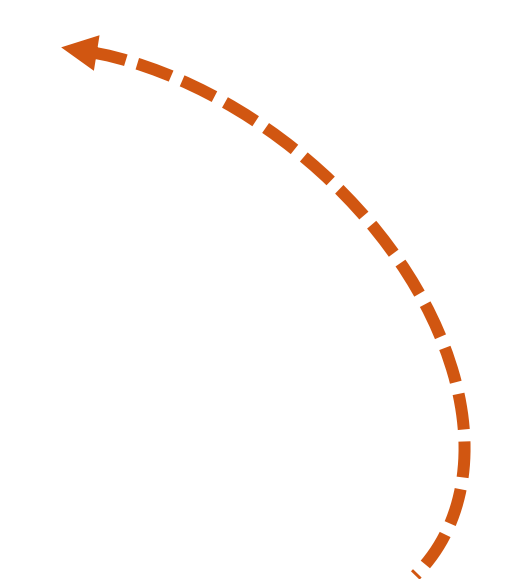
Możemy to zrobić, używając funkcji **del()**

```
del(star_wars["year"])
print(star_wars["year"])
```

Wynik:

```
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    print(star_wars["year"])
KeyError: 'year'
```

```
star_wars = {
    "episode": 4,
    "title": "A New Hope",
    "characters": ["Luke", "Leia",
                  "Han Solo", "Obi-Wan", "Chewbacca",
                  "Vader", "C3PO", "R2D2"]
}
```



Zniknął klucz „year”
i powiązana z nim wartość!

Słowniki, a pętle

Iterowanie po słownikach:

Używając pętli, możemy też iterować po kluczach słownika:

```
for key in star_wars:  
    print(key)
```

Wynik:

```
title  
characters  
episode
```

Komendzie **print** możesz podać wiele informacji,
Zobacz tutaj:

<https://docs.python.org/3/tutorial/inputoutput.html>,
http://www.python-course.eu/python3_print.php

W prosty sposób możemy dostać się do dowolnej pary klucz: wartość:

```
for key in star_wars:  
    print(key, star_wars[key], sep=": ")
```

Wynik:

```
title: A New Hope  
characters: ['Luke', 'Leia', 'Han Solo',  
'Obi-Wan', 'Chewbacca', 'Vader', 'C3PO',  
'R2D2']  
episode: 4
```

Słowo kluczowe
„in”

Słowo kluczowe „in”

Iterowanie po zestawach danych

Jeśli chcemy iterować po poszczególnych elementach listy, użyjemy słowa kluczowego **in** w następującym kontekście:

```
droids = ["C3P0", "R2D2", "BB-8"]
```

```
for droid in droids:  
    print(droid)
```

```
starship = {  
    "type": "X-wing",  
    "allegiance": "Resistance",  
    "owner": "Poe Dameron"  
}  
  
for feature in starship:  
    print(feature, starship["feature"])
```

Słowo kluczowe „in”

Ułatwienie przeszukiwania:

Jeśli chcemy sprawdzić, czy dany element znajduje się na liście, użyjemy słowa kluczowego **in** w następującym kontekście:

```
droids = ["C3P0", "R2D2", "BB-8"]
```

```
"BB-8" in droids # True
```

```
"2-1B" in droids # False
```

Słowo kluczowe **in** w słowniku, sprawdza czy dany klucz istnieje:

```
starship = {  
    "type": "X-wing",  
    "allegiance": "Resistance",  
    "owner": "Poe Dameron"  
}
```

```
"allegiance" in starship # True
```

```
"color" in starship # False
```

Łańcuchy tekstowe

Łańcuch tekstowy (string)

Czym jest string?

Łańcuch tekstowy, czyli **string**, to po prostu napis.

Napisem jest wszystko umieszczone w cudzysłowach:

Można napis umieszczać w normalnym, podwójnym cudzysłowie:

```
begin = "a long time ago in a galaxy  
far, far away..."
```

Ale można i w pojedynczym, dla Pythona nie ma żadnej różnicy:

```
title = 'Episode IV: A New Hope'
```

Jak już wiecie, Python 2 ma problemy z kodowaniem np. polskich znaków diakrytycznych.

Dlatego, jeśli programujecie w Pythonie 2, na początku pliku najlepiej umieścić magiczny komentarz:

```
# -*- coding: utf-8 -*-
```

Python 3 nie ma takich problemów, domyślnie stosuje typ Unicode.

Łańcuch tekstowy (string)

Czym jest string?

Możemy również definiować wielolinijkowe stringi!

Wystarczy zamiast pojedynczego cudzysłowu (lub znaczka ') użyć potrójnego.

```
screenplay = """
```

VADER

If only you knew the power of the Dark Side. Obi-Wan never told you what happened to your father.

LUKE

He told me enough! He told me you killed him!

VADER

No, I am your father.

LUKE

(shocked)

No. No! That's not true! That's impossible!"""

Łańcuch tekstowy (string)

Ciekawe właściwości stringa

Co ciekawe: Python traktuje string jak listę pojedynczych znaków, co oznacza że możemy poruszać się po nim, dokładnie tak samo, jak po liście:

```
book = "Harry Potter and the Prisoner of  
Azkaban"
```

```
book[0:5] # 'Harry'  
book[-7:] # 'Azkaban'
```

```
print(book[::-1])
```

Wynik:

nabakzA fo renosirP eht dna rettoP yrraH

Można też iterować po łańcuchu tekstowym:

```
name = book[:5]  
for letter in name:  
    print letter
```

Wynik:

H
a
r
r
y

Łańcuch tekstowy (string)

Łączenie i formatowanie stringów

Jak już zaprezentowano, łańcuchy tekstowe można łączyć używając operatora `+`.

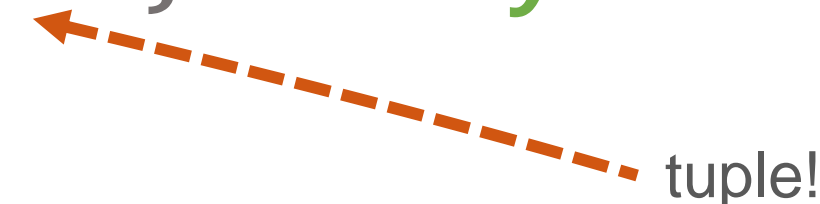
```
print("Bilbo" + "Baggins")
```

Wynik:

BilboBaggins

Inną popularną metodą (dyskusyjne jest, czy lepszą) jest formatowanie łańcuchów tekstowych:

```
hobbits = ("Frodo", "Sam", "Merry",  
           "Pippin")  
  
desc = "Hobbits in LOTR: %s, %s, %s and  
       %s." % hobbits  
  
print(desc)
```



Wynik:

Hobbits in LOTR: Frodo, Sam, Merry and Pippin.

Łańcuch tekstowy (string)

Jeszcze o formatowaniu stringów:

Podobnym sposobem formatowania stringów jest formatowanie przy pomocy słownika:

```
hobbit = {  
    "name": "Bilbo Baggins",  
    "occupation": "burglar"  
}
```

```
description = "%(name)s is a  
%(occupation)s." % hobbit
```

```
print(description)
```

Wynik:

Bilbo Baggins is a burglar.



Moduły i importowanie

Moduły Pythona

Python jest dostarczany razem z wieloma dodatkowymi **modułami** (bibliotekami) rozszerzającymi jego funkcjonalności.

Moduł to wydzielony fragment kodu. Grupuje powiązane ze sobą funkcje (np. funkcje daty i czasu) i pozwala na lepsze zorganizowanie oprogramowania.

Aby móc wykorzystać moduł w swoim programie należy go **zimportować**, czyli powiadomić interpreter Pythona, że będziemy używać tego modułu.

Przykłady standardowych modułów:

datetime — zawiera funkcje operujące na datach, czasie, itp.

math — zawiera funkcje matematyczne,

os — umożliwia interakcję z systemem operacyjnym,

random — liczby pseudolosowe,

urllib — biblioteka umożliwiająca łączenie z serwerami,

curses — zaawansowana obsługa terminala.

Importowanie modułów

Aby zaimportować moduł i móc go wykorzystać, należy użyć dyrektywy **import**:

```
import <nazwa-modułu>
```

Jeśli chcemy użyć modułu **random** i użyć go należy napisać:

```
import random
```

Importy powinniśmy wykonywać na początku pliku jeden pod drugim.


Jeśli zaimportujemy moduł w taki sposób, możemy wywołać funkcje znajdujące się wewnątrz tego modułu. W tym celu musimy użyć **nazwy modułu**, **kropki** i **nazwy potrzebnej funkcji**:

```
moduł.funkcja(parametry)
```

Przykład:

```
rnd = random.randint(10, 20)
```

 moduł **random**

 funkcja **randint** znajdująca się w module **random**

Importowanie modułów

Przykłady

```
import random
rnd = random.randint(10, 20)

print("Los w zakresie 10-20:", rnd)
```

Wynik (przykładowy):

Los w zakresie 10-20: 11

```
import calendar
# weekday to funkcja obliczająca numer
# dnia tygodnia dla podanej daty,
# day_name to tablica z nazwami dni
# tygodnia
```

```
day = calendar.weekday(1977, 5, 25)
print(calendar.day_name[day])
```

Wynik:

Wednesday

Importowanie funkcji

W module **calendar** kryje się mnóstwo funkcji. Do naszego kodu nie potrzebujemy wszystkich — w programie z poprzedniego przykładu użyliśmy tylko dwóch.

Aby nie marnować zasobów komputera (procesor, czas, pamięć) warto zaimportować tylko te funkcje, których potrzebujemy.

Zajrzyj do dokumentacji modułu calendar:
<https://docs.python.org/3/library/calendar.html>

Aby zaimportować tylko potrzebne funkcje, należy użyć następującej konstrukcji:

```
from module import func1, func2, ..., funcN
```

Jeżeli zatem chcemy zaimportować funkcję **weekday** i listę **weekday** z modułu **calendar**, powinniśmy napisać:

```
from calendar import weekday, day_name
```

Od tej pory możesz w programie używać tych dwóch zaimportowanych nazw.

Importowanie modułów

Przykład:

```
from calendar import weekday, day_name
# weekday to funkcja obliczająca
# numer dnia tygodnia dla podanej daty,
# day_name to tablica z nazwami dni
# tygodnia
```

```
day = weekday(1977, 5, 25)
print(day_name[day])
```

Wynik:

Wednesday



Zwróć uwagę, że już nie używamy nazwy modułu podczas wywoływania zaimportowanej funkcji.

Importowanie funkcji

Jeśli chcesz zaimportować wszystkie elementy z modułu możesz użyć gwiazdki:

```
from <nazwa-modułu> import *
```

Na przykład:

```
from random import *
```

Taka metoda importowania, mimo że poprawna, nie jest polecana i jest uznawana za złą praktykę!

Jeśli chcesz dowiedzieć się więcej na temat standardowej biblioteki Pythona (czyli wszystkich modułów dołączonych do języka), zajrzyj do dokumentacji języka Python:

<https://docs.python.org/3/library/>

Tworzenie własnych modułów

Własne moduły

Moduł Pythona to w rzeczywistości plik z funkcjami, zmiennymi, klasami, itp. Jeśli stworzymy taki plik i zapiszemy go na dysku, mamy prosty moduł. Jego nazwą jest nazwa pliku (bez rozszerzenia **.py**)

Plik `hobbit_mod.py`:

```
hobbit = {  
    "name": "Bilbo Baggins",  
    "occupation": "burglar"  
}  
  
def description():  
    return "Mr %(name)s is a "  
           "%(occupation)s." % hobbit
```

Przykład użycia funkcjonalności modułu:

```
from hobbit_mod import hobbit, description  
  
print("Dane hobbita:", hobbit)  
print(description())
```

Wynik:

```
Dane hobbita: {'occupation': 'burglar',  
              'name': 'Bilbo Baggins'}  
Mr Bilbo Baggins is a burglar.
```

Biblioteki

Python jest dostarczany z rozbudowaną biblioteką standardową, ale ona nie wyczerpuje wszystkich potrzeb ani możliwości.

Wielu zewnętrznych programistów, tworzy i udostępnia różne rozwiązania, które można łatwo wykorzystać w swoich projektach. Te projekty są zgromadzone w jednym miejscu: **Python Package Index** (w skrócie **PyPI**).

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

O włączeniu, bądź usunięciu biblioteki z PyPI decyduje **Python Packaging Authority**.

<https://www.pypa.io/en/latest/>

Przykłady bibliotek z PyPI:

django — (wymawiane z niemym „D”) framework do tworzenia aplikacji webowych,

requests — łatwa obsługa połączeń HTTP,

mysql-connector — umożliwia pracę bazą danych MySQL w Pythonie,

BeautifulSoup — biblioteka do obsługi HTML/XML,

pygame — biblioteka do pisania gier w Pythonie,

PIL — obsługa i manipulacja grafiką 2D.

Biblioteki

Do instalowania bibliotek służy narzędzie o nazwie **pip**.

Od wersji 3.4 jest włączone do dystrybucji języka. W wersji 2 i 3, niższej od 3.4 narzędzie trzeba doinstalować ręcznie z terminala:

Python 2:

```
sudo apt install python-pip
```

Python 3:

```
sudo apt install python3-pip
```

Instalowanie biblioteki zewnętrznej:

Aby zainstalować bibliotekę w systemie, należy wydać komendę w terminalu:

```
pip3 install <nazwa>
```

np.

```
pip3 install django
```

Biblioteka instaluje się **zawsze** w najnowszej wersji, ale można ją zaktualizować:

```
pip3 install --upgrade <nazwa>
```

Biblioteki

pip freeze

Jeśli nasz projekt używa wielu bibliotek i nie chcemy instalować ich ręcznie, można użyć pliku tekstowego z listą bibliotek, które chcemy zainstalować.

Aby przygotować taki plik należy wpisać

```
pip freeze > <nazwa-pliku>
```

Instalacja bibliotek z pliku:

```
pip install -r <nazwa-pliku>
```

Instrukcja obsługi programu pip razem ze wszystkimi komendami:

<https://pip.pypa.io/en/stable/>

Pakiety języka Python

Pakiety

Pakiety

Do tej pory zapoznaliśmy się ze sposobem organizowania kodu Pythona w moduły. Moduł to, przypomnijmy, zwykły plik **.py**, w którym zawarte są funkcje, zmienne globalne, albo klasy (w programowaniu obiektowym, o czym później).

Kod Pythona można organizować jeszcze bardziej szczegółowo: w **pakiety**.

Pakiet, to mówiąc krótko, katalog modułów Pythona.

Ułatwia to tworzenie i dystrybucję bibliotek programistycznych.

Aby Python traktował katalog z modułami, jako pakiet, wewnątrz tego katalogu musi znaleźć się plik **__init__.py**. Może być pusty.

Przykład

Wyobraźmy sobie taką strukturę:

```
my_package
+-- __init__.py
+-- my_module1.py
+-- my_module2.py
+-- my_module3.py
```

W takiej sytuacji możemy zaimportować np.

```
import my_package.my_module1
from my_package import (my_module1,
                        my_module2)
from my_package.my_module1 import *
```