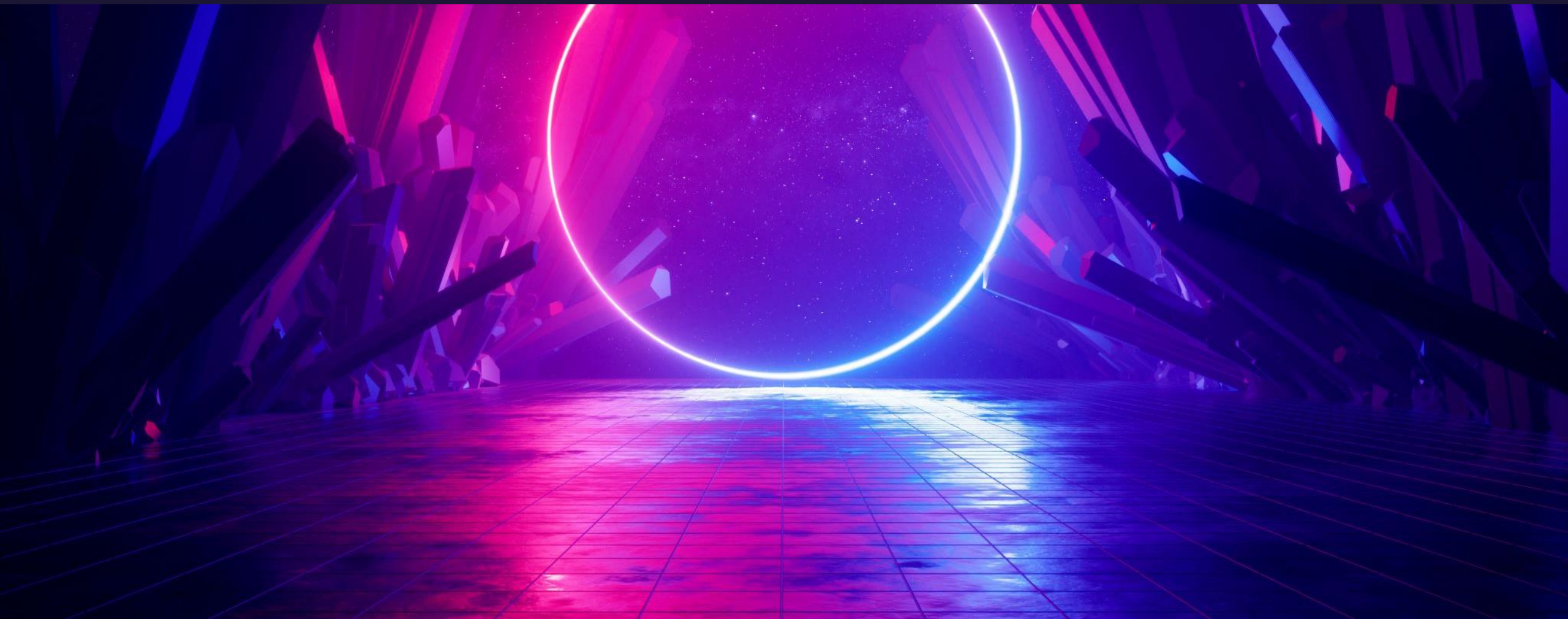
Programowanie w Pythonie

Łukasz Mioduszewski, UKSW 2022

Klasy, iteratory, biblioteki os, sys, argparse, glob



Obiekty klasowe

- Z metod i atrybutów klasy można korzystać tak jak dla obiektu:

main.py

```
1  class MyClass:
2      """A simple example class"""
3      i = 12345
4
5      def f(self):
6          return 'hello world'
7
8  print(MyClass.f(None))    # wypisuje hello world
9  x = MyClass() # konstruktor tworzy obiekt danej klasy
10 MyClass.i = 5 # teraz wszystkie obiekty maja i=5
11 print(x.i)     # wypisuje 5
12 x.i = 8
13 MyClass.i = 1 # x.i nadal wynosi 8
14 print(x.i)
```

Iteratory – przypomnienie z wykładu 2

- Iterator pozwala znaleźć następny element, każdy iterowalny obiekt go ma
- Można uzyskać iterator funkcją iter, np.
  myit=iter('pyton')
- Można dostać wartość następnego elementu, zarazem iterując, funkcją next, np.
  print(next(myit)) # wypisuje p
  print(next(myit)) # wypisuje y
- Uwaga, po 'n' wyrzuci błąd StopIteration, dodaj do next domyślną wartość aby tak nie robił, np.
  print(next(myit,None))

Jak naprawdę działa pętla for?

- Wywołuje metodę iter() na obiekcie po którym iteruje. Metoda zwraca iterator
- Pętla for używa funkcji next() na tym iteratorze dopóki nie dojdzie do wyjątku StopIteration, który łapie i wtedy pętla się kończy

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<str_iterator object at 0x10c90e650>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

Własny iterator

- Iteratorem może być dowolny obiekt który posiada metodę __next__(self), zwracającą odpowiednie dane lub zgłaszającą wyjątek StopIteration
- Obiektem iterowalnym może być dowolny obiekt który posiada metodę __iter__(self), która zwraca iterator
- Jeśli obiekt ma metodę __next__, wystarczy jeśli __iter__ zwróci self

Własny iterator - przykład

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

Własny iterator - przykład

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

Generatory

- Zwykła metoda __next__ musi pamiętać indeks (jak w przykładzie) albo coś podobnego – na pewno da się to zrobić jakoś prościej…
- Generatory to funkcje które zwracają coś więcej niż raz – zamiast return mają słowo specjalne yield
- Wywołanie generatora **automatycznie** tworzy iterator, metodę __next__ i w dodatku na koniec samo zgłasza wyjątek StopIteration
- Stan generatora jest **pamiętany** pomiędzy wywołaniami

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Wyrażenia generatorowe

- Kiedy chcemy od razu użyć obiektu w funkcji, nie musimy tworzyć go wyrażeniem listowym (które tworzy obiekt), tylko generatorowym (które tworzy iterator od razu wykorzystywany przez funkcję, co oszczędza pamięć)

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> unique_words = set(word for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

Zadanie domowe – exceptiongroup

- ExceptionGroup jest domyślne dopiero od Pythona 3.11, aby używać we wcześniejszym trzeba zrobić from exceptiongroup import ExceptionGroup
- Jeśli nie ma biblioteki exceptiongroup, trzeba ją zainstalować:
  pip install exceptiongroup
- Konstrukcja except* i tak nie będzie działać we wcześniejszych wersjach

Zadanie domowe – prosty sposób

- Aby sprawdzić czy plik istnieje, można spróbować tak (ten sposób nie jest idealny, bo próbuje otworzyć plik, przez co zmienia datę odczytu):

main.py

```
1  try:
2      f = open('pracownicy.pkl','rb')
3  except FileNotFoundError:
4      czyIstnieje = False
5  except Exception as e:
6      print("Plik",f,"istnieje i nie można go otworzyć")
7      czyIstnieje = True
8  else:
9      czyIstnieje = True
10     f.close()
```

Zadanie domowe – dobry sposób

- Filozofia Pythona: **do wszystkiego ma być odpowiednia biblioteka**
- os.getcwd() zwraca string current working directory
- moduł os.path zawiera funkcję exists która sprawdza czy plik istnieje

main.py

```
1  path=os.getcwd()+'\pracownicy.pkl'
2  czyIstnieje = os.path.exists(path)
3
4
5  # albo jeszcze krócej:
6  czyIstnieje = os.path.isfile('pracownicy.pkl')
7
8
9
10
```

Biblioteka os

- Zawsze używaj import os, a nie from os import *
  bo funkcja os.open() nadpisze pythonowe open() itd.
- chdir zmienia nasze położenie, system wywołuje polecenie w konsoli
- nazwy funkcji jak zawsze w dir(os), pomoc w help(os)

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python311'
>>> os.chdir('/server/accesslogs')   # Change current working directory
>>> os.system('mkdir today')   # Run the command mkdir in the system shell
0
```

Biblioteka os

- Zwykle zwraca wyjątki OSError (inna nazwa to os.error) kiedy jest zła ścieżka itp.
- os.name to nazwa systemu operacyjnego, os.uname() ma więcej szczegółów
- os.environ to obiekt mapowalny (taki słownik), gdzie klucze to zmienne środowiskowe (HOME, PATH itd.), a wartości to ich wartości **w momencie uruchamiania Pythona albo import os** (jeśli później się zmienią to environ zostanie taki jak był). Uwaga: modyfikowanie os.environ na MacOS, FreeBSD i innych może powodować wycieki pamięci
- os.getlogin() zwraca nazwę użytkownika konsoli w której wywołano Pythona
- uid, pid, gid itd. też tam są

Biblioteki shutil oraz glob

- shutil to wysokopoziomowa alternatywa dla os

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

- glob pozwala wyszukiwać w katalogu przy użyciu *

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

Biblioteka shutil

- shutil.copyfileobj(fsrc,fdst[, length]) kopiuje zawartość obiektu plikowego fsrc **od aktualnego położenia "kursora" do końca pliku** do obiektu plikowego fdst, rozmiar buforu to length (ujemne length kopiuje plik na raz)
- shutil.copy2(src, dst, *, follow_symlinks=True) stara się przekopiować plik ze **ścieżki** src do pliku lub katalogu ze **ścieżki** dst, starając się zachować (copy się nie stara) metadane takie jak właściciel, czas powstania i modyfikacji itd.
- shutil.copystat(src, dst, *, follow_symlinks=True) kopiuje flagi, uprawnienia, czas ostatniej modyfikacji i dostępu, nie ruszając zawartości, grupy i właściciela
- shutil.copytree(src, dst, symlinks=False, ignore=None, copy_function=copy2, ignore_dangling_symlinks=False, dirs_exist_ok=False) kopiuje cały katalog

Biblioteka sys

- sys.exception() zwraca obecnie rozpatrywany wyjątek (ten złapany w najbardziej wewnętrznym bloku except w którym aktualnie jesteśmy), od wersji 3.11
- sys.flags to tupla zawierająca flagi z jakimi został odpalony interpreter
- sys.getfilesystemencoding() zwraca kodowanie systemu plików
- sys.getsizeof(object[, default]) zwraca rozmiar w bajtach dowolnego obiektu
- sys.version to wersja Pythona
- Aby bezwarunkowo wyjść ze skryptu: sys.exit()

Biblioteka sys - wejście, wyjście i błędy

- Wiele konsol pozwala przekazywać programom wejście stdin, np. echo "test"|python program.py
- Funkcja print() wypisuje na standardowe wyjście, stdout
- Błędy są w stderr
- Wszystkie 3 są w sys (oryginalne wartości to np. sys.__stdout__)

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

Biblioteka argparse - alternatywa dla sys.argv

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

python top.py --lines=5 alpha.txt beta.txt
wtedy args.lines wynosi 5, a args.filenames wynosi ['alpha.txt', 'beta.txt'].

Biblioteka argparse -h w pakiecie

```
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

- --help jest wbudowany, --verbose już nie

```
$ python3 prog.py
$ python3 prog.py --help
usage: prog.py [-h]

options:
  -h, --help  show this help message and exit
$ python3 prog.py --verbose
usage: prog.py [-h]
prog.py: error: unrecognized arguments: --verbose
$ python3 prog.py foo
usage: prog.py [-h]
prog.py: error: unrecognized arguments: foo
```

Biblioteka argparse – argumenty pozycyjne

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo", help="echo the string you use here")
args = parser.parse_args()
print(args.echo)
```

- Działanie:
  python3 prog.py
  usage: prog.py [-h] echo
  prog.py: error: the following arguments are required: echo
  python3 prog.py foo
  foo
  python3 prog.py –h
  usage: prog.py [-h] echo
  positional arguments:
  echo echo the string you use here
  options:
  -h, --help show this help message and exit

Biblioteka argparse – argumenty opcjonalne

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

- Działanie:

```
$ python3 prog.py --verbose
verbosity turned on
$ python3 prog.py --verbose 1
usage: prog.py [-h] [--verbose]
prog.py: error: unrecognized arguments: 1
$ python3 prog.py --help
usage: prog.py [-h] [--verbose]

options:
  -h, --help  show this help message and exit
  --verbose   increase output verbosity
```

Biblioteka argparse – typy argumentów

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number",
                    type=int)
args = parser.parse_args()
print(args.square**2)
```

- Działanie:

```
$ python3 prog.py 4
16
$ python3 prog.py four
usage: prog.py [-h] square
prog.py: error: argument square: invalid int value: 'four'
```

Biblioteka argparse – podsumowanie

- choices pozwala ograniczyć dostępne wartości argumentów
- argument może mieć więcej niż jedną nazwę (-v i –-verbosity to to samo)

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int, choices=[0, 1, 2],
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

Następnym razem: biblioteka re dla wyrażeń regularnych

- Biblioteka re pozwala używać wyrażeń regularnych:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

- Ale proste rzeczy można zrobić też bez niej:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```