Notatki ze szkolenia w formie warsztatów w siedzibie firmy Sii we Wrocławiu

- Poziom średno zaawansowany, 8h
- 27.05.2018
- Przygotował: Grzegorz Gwóźdź

Lambda, map, filter, reduce

Lambda

Wyrażenie lambda. Funkcja nienazwana.

In [18]:

```
double = lambda x : x * 2
double(2)
```

Out[18]:

```
4
```

In [19]:

```
def make_doubler():
    return lambda x : x * 2
doubler = make_doubler()
doubler(3)
```

Out[19]:

```
6
```

In [20]:

```
def make_adder(a):
    return lambda x : x + a
add_two = make_adder(2)
add_two(11)
```

Out[20]:

```
13
```

`map()`

"Nakładanie" funkcji na kolekcję.

`map(funkcja, lista)`

In [21]:

```
numbers = [1, 2, 3, 4, 5]
def double(x):
    return x * 2
list(map(double, numbers))
```

Out[21]:

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

In [22]:

```
pairs = [('a', 1), ('u', 4), ('h', 3)]
def test(xy):
    x, y = xy
    return f'{x}_{y}'

result = map(test, pairs)
print(list(result))

for i in map(test, pairs):
    print(i)
```

```
['a_1', 'u_4', 'h_3']
a_1
u_4
h_3
```

In [23]:

```
numbers = [1, 2, 3, 4, 5]
list(map(lambda x : x + 13, numbers))
```

Out[23]:

```
[14, 15, 16, 17, 18]
```

`map() + lambda`

filter()

Zwraca listę elementów, dla których predykat jest prawdziwy.

In [24]:

```
numbers = list(range(20))
print(numbers)
numbers = list(filter(lambda x : x % 2 == 0, numbers))
print(numbers)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

In [25]:

```
numbers = list(range(20))

numbers = filter(lambda x : x > 10, numbers)
numbers = list(numbers)

print(numbers)
```

```
[11, 12, 13, 14, 15, 16, 17, 18, 19]
```

reduce()

Dokonuje pewnych obliczeń na liście elementów i zwraca wynik.

In [26]:

```
from functools import reduce
numbers = list(range(10))
print(numbers)
suma = reduce(lambda x, y: x + y, numbers)

# x y
#
# 0 1 → 1
# 1 2 → 3
# 3 3 → 6
# 6 4 → 10
# ....

print(suma)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
45
```

In [27]:

```
from functools import reduce
numbers = list(range(1, 20))

product = reduce(lambda x, y: x * y, numbers)
print(product)
```

```
121645100408832000
```

ZADANIE ```def suma(4, 12)``` niech wypisze: **4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 = 72**

In [28]:

```
from functools import reduce

def suma(a, b):
    numbers = range(a, b + 1)
    result = reduce(lambda x, y: str(x) + ' + ' + str(y), numbers)
    return f'{result} = {sum(numbers)}'

print(suma(4, 12))
```

```
4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 = 72
```

Lambda + map() + filter() + reduce()

In [29]:

```
# analiza: 01/map_filter_reduce.py

from functools import reduce

numbers = range(100)

is_odd = lambda x : x % 2 == 1
doubler = lambda x : x * 2
adder = lambda x, y : x + y

print(reduce(adder, map(doubler, filter(is_odd, numbers))))
```

```
5000
```

Przeciążanie operatorów

In [30]:

```
help(int.__add__)
```

```
Help on wrapper_descriptor:

__add__(self, value, /)
    Return self+value.
```

In [31]:

```
class Cat(object):
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return f'Cat: {self.name} {id(self)}'

cat = Cat('Mruczek')
print(cat)
```

```
Cat: Mruczek 140433888252312
```

Własny typ *Liczba*

In [32]:

```
class Cat: pass

class Liczba:
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        if not isinstance(other, type(self)):
            raise Exception('Nie mozna dodac kota do liczby')
        return self.value + other.value

a = Liczba(4)
b = Liczba(5)
cat = Cat()

a + b
# Exception!
# a + cat
```

Out[32]:

```
9
```

ZADANIE a = Liczba(4) b = Liczba(5) a + b 9

Dekoratory

Są to "opakowania" dla innych funkcji. Modyfikują rezultat dekorowanych funkcji.

Wszystko jest obiektem

In [33]:

```
def hi(name='Grzesiek'):
    return 'Hi ' + name + '!'

hi()
```

Out[33]:

```
'Hi Grzesiek!'
```

In [34]:

```
my_hi = hi
my_hi()
```

Out[34]:

```
'Hi Grzesiek!'
```

In [35]:

```
del hi
my_hi()
```

Out[35]:

```
'Hi Grzesiek!'
```

Funkcje zagnieżdżone

Funkcja w funkcji.

In [36]:

```
def hi(name='Grzesiek'):
    print('inside hi')

    def welcome():
        return 'inside welcome'

    hi.welcome = welcome

    print(welcome())

hi()
hi.welcome()
# print(dir(hi))
```

```
inside hi
inside welcome
```

Out[36]:

```
'inside welcome'
```

Zwracanie funkcji z funkcji

In [37]:

```
def hi(name='Grzesiek'):

    def welcome():
        return 'inside welcome'

    def bye():
        return 'inside bye'

    if name == 'Grzesiek':
        return welcome
    else:
        return bye

a = hi()
print(a)
a()
```

```
<function hi.<locals>.welcome at 0x7fb950059e18>
```

Out[37]:

```
'inside welcome'
```

Przekazywanie funkcji jako argumentu

In [38]:

```
def hi(name='Grzesiek'):
    return 'Hi ' + name + '!'

def before(func):
    print('before func execution')
    print(func())

before(hi)
```

```
before func execution
Hi Grzesiek!
```

Dekorator

In [39]:

```
def my_decorator(func):
    def wrap_the_func():
        print('before func execution')
        func()
        print('after func execution')

    return wrap_the_func

def hi(name='Grzesiek'):
    print('Hi ' + name + '!')

hi = my_decorator(hi)
hi()
```

```
before func execution
Hi Grzesiek!
after func execution
```

Za pomocą dekoracji z @

In [40]:

```
def my_decorator(func):
    def wrap_the_func():
        print('before func execution')
        func()
        print('after func execution')

    return wrap_the_func

@my_decorator
def hi(name='Grzesiek'):
    print('Hi ' + name + '!')

hi()
```

```
before func execution
Hi Grzesiek!
after func execution
```

Przykład: Tagi HTML.

In [41]:

```
def bold(func):
    def wrapper():
        print('<b>', end='')
        func()
        print('</b>')

    return wrapper

def underline(func):
    def wrapper():
        print('<u>', end='')
        func()
        print('</u>')

    return wrapper

@bold
@underline
def hi(name='Grzesiek'):
    print('Hi ' + name + '!', end='')

hi()
print(dir(hi))
```

```
<b><u>Hi Grzesiek!</u>
</b>
['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__defaults__', '__dela
ttr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__get__', '__getat
tribute__', '__globals__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__kwdefaults
__', '__le__', '__lt__', '__module__', '__name__', '__ne__', '__new__', '__qualname__', '__redu
ce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

W biblitoce standardowej możemy znaleźć wiele dekoratorów. W dokumentacji przed ich nazwą stoi znak `@`. Na przykład dla functools.wraps (https://docs.python.org/3/library/functools.html#functools.wraps)

```
@functools.wraps(wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)
```

Problem z dekoratorem

In [42]:

```
def my_decorator(func):
    '''
    doc: my_decorator
    '''
    def wrapper():
        '''
        doc: wrapper
        '''
        print('before func execution')
        func()
        print('after func execution')

    return wrapper

@my_decorator
def hi(name='Grzesiek'):
    '''
    doc: hi
    '''
    print('Hi ' + name + '!')

print(hi.__name__)
print(hi.__doc__)
```

```
wrapper

        doc: wrapper

```

Można to naprawić korzystając z modułu `functools`

In [43]:

```python
from functools import wraps

def my_decorator(func):
    '''
    doc: my_decorator
    '''
    @wraps(func)
    def wrapper():
        '''
        doc: wrapper
        '''
        print('before func execution')
        func()
        print('after func execution')

    return wrapper

@my_decorator
def hi(name='Grzesiek'):
    '''
    doc: hi
    '''
    print('Hi ' + name + '!')

print(hi.__name__)
print(hi.__doc__)
```

```
hi

    doc: hi
    
```

Zastosowanie

In [44]:

```python
def logit(f):
    def wrapper(a, b):
        print('executing:', f.__name__, 'with args:', a, b)
        return f(a, b)
    return wrapper

@logit
def add(a, b):
    return a + b

x = add(4, 9)
print(x)
```

```
executing: add with args: 4 9
13
```

Pojawiły się dwa argumenty. Co gdy będzie więcej?

In [45]:

```
def logit_args2(f):
    def wrapper(a, b):
        print('executing:', f.__name__, 'with args:', a, b)
        return f(a, b)
    return wrapper

def logit_args3(f):
    def wrapper(a, b, c):
        print('executing:', f.__name__, 'with args:', a, b, c)
        return f(a, b, c)
    return wrapper

@logit_args2
def add2(a, b):
    return a + b

@logit_args3
def add3(a, b, c):
    return a + b + c

x = add2(4, 9)
print(x)
y = add3(1, 2, 3)
print(y)
```

```
executing: add2 with args: 4 9
13
executing: add3 with args: 1 2 3
6
```

Czy da się zgrabniej? Da się.

In [46]:

```
def log_it(f):
    def wrapper(*args):
        print('executing:', f.__name__, 'with args:', args)
        return f(*args)
    return wrapper

@log_it
def add2(a, b):
    return a + b

@log_it
def add3(a, b, c):
    return a + b + c

x = add2(4, 9)
print(x)
y = add3(1, 2, 3)
print(y)
```

```
executing: add2 with args: (4, 9)
13
executing: add3 with args: (1, 2, 3)
6
```

Czy da się sparametryzować dekorator? Da się.

In [47]:

```
from time import time

def logit(logfile):
    def logging_decorator(func):
        def wrapper(*args):
            log_string = '[{}] executing {} with args: {}'.format(int(time()), func.__name__, args)
            print(log_string)
            with open(logfile, 'at') as f:
                f.write(log_string)
                f.write('\n')
            return func(*args)
        return wrapper
    return logging_decorator

@logit('/tmp/log.txt')
def add2(a, b):
    return a + b

@logit('/tmp/log.txt')
def add3(a, b, c):
    return a + b + c

x = add2(4, 9)
print(x)
y = add3(1, 2, 3)
print(y)
```

```
[1527446334] executing add2 with args: (4, 9)
13
[1527446334] executing add3 with args: (1, 2, 3)
6
```

ZADANIE Dekorator, który zaloguje czas trwania wykonania funkcji.

In [48]:

```
def checkfile(f):
    def wrapper(filename, data):
        try:
            f(filename, data)
        except FileNotFoundError:
            print('Brak pliku!')
    return wrapper

@checkfile
def save_data(filename, data):
    with open(filename, 'rt') as f:
        f.write(data)

save_data('/tmp/data.dat', b'dandanadnadndnandandna')
```

```
Brak pliku!
```

Iteratory i generatory

iterable - zdolny do bycia powtórzonym

In [49]:

```
for i in [1, 2, 3, 4]:
    print(i)
```

```
1
2
3
4
```

In [50]:

```
for logline in open('/tmp/log.txt'):
    print(logline)
```

```
[1527444670] executing add2 with args: (4, 9)

[1527444670] executing add3 with args: (1, 2, 3)

[1527446334] executing add2 with args: (4, 9)

[1527446334] executing add3 with args: (1, 2, 3)
```

In [51]:

```
'---'.join(['Ala', 'ma', 'kota'])
```

Out[51]:

```
'Ala---ma---kota'
```

In [52]:

```
list('Python')
```

Out[52]:

```
['P', 'y', 't', 'h', 'o', 'n']
```

Iterator

Wbudowana funkcja `iter()` przyjmuje jako argument obiekt **iterable** i zwraca do niego iterator. Obiekt powinien posiadać metodę `__iter__()` lub `__getitem__()`

In [53]:

```
numbers = [1, 2, 3, 4]
it = iter(numbers)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

```
1
2
3
4

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-53-1035bffd95c8> in <module>()
      5 print(next(it))
      6 print(next(it))
----> 7 print(next(it))

StopIteration: 
```

Drugi (opcjonalny) parametr `iter()` - **sentinel**, określa koniec sekwencji.

In [1]:

```
def ask_password():
    user_input = input('Password:')
    return user_input

for ask in iter(ask_password, 'haha'):
    print('Bad password!')
```

```
Password:hello
Bad password!
Password:passowrd
Bad password!
Password:haha
```

Własny obiekt *iterable*

In [2]:

```
class zakres:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __iter__(self):
        return self

    def __next__(self):
        if self.a < self.b:
            a = self.a
            self.a += 1
            return a
        else:
            raise StopIteration()

for i in zakres(1, 5):
    print(i)


print('sum(zakres(0, 10)) =', sum(zakres(0, 10)))
```

```
1
2
3
4
sum(zakres(0, 10)) = 45
```

Generator

Uproszczony sposób na iterator. Jest to funkcja produkująca sekwencję wyników zamiast jednej wartości.

In [3]:

```
def zakres(a, b):
    while a < b:
        yield a
        a += 1

for i in zakres(3, 8):
    print(i)
```

```
3
4
5
6
7
```

In [4]:

```
def week():
    yield 'PN'
    yield 'Wt'
    yield 'se'
    yield 'asdasd'

for i, day in enumerate(week()):
    print(i, day)

it = iter(week())
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))

print(list(week()))
```

```
0 PN
1 Wt
2 se
3 asdasd
PN
Wt
se
asdasd

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-4-7e3e79db5a2b> in <module>()
     13 print(next(it))
     14 print(next(it))
---> 15 print(next(it))
     16 
     17 print(list(week()))

StopIteration: 
```

In [5]:

```
def integers():
    i = 0
    while True:
        yield i
        i += 1

def squares():
    for i in integers():
        yield i * i

def take(n, seq):
    it = iter(seq)
    result = []
    try:
        for i in range(n):
            result.append(next(it))
    except StopIteration:
        pass
    return result

print(take(10, squares()))
```

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

Generator, a *List Comprehension*

In [6]:

```
lista = [x for x in range(100)]
generator = (x for x in range(100))
print('lista =', lista)
print('generator =', generator)

print(sum(lista))
print(sum(generator))
print(sum(lista))
print(sum(generator))
```

```
lista = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,
96, 97, 98, 99]
generator = <generator object <genexpr> at 0x7f35d4faf518>
4950
4950
4950
0
```

Można pominąć parę nawiasów gdy generator jest jedynym argumentem funkcji.

In [7]:

```
sum(x for x in range(4))
```

Out[7]:

```
6
```

itertools

chain

In [8]:

```
from itertools import chain
it1 = iter([1, 2, 3])
it2 = iter([4, 5, 6])
it3 = iter([7, '8', 9, 10])
it = chain(it1, it2, it3)
list(it)
```

Out[8]:

```
[1, 2, 3, 4, 5, 6, 7, '8', 9, 10]
```

accumulate

In [9]:

```
from itertools import accumulate
from operator import add
accum = accumulate([1, 1, 1, 1, 2, 10], add)
list(accum)
```

Out[9]:

```
[1, 2, 3, 4, 6, 16]
```

combinations, permuations

In [10]:

```
from itertools import combinations as comb
from itertools import combinations_with_replacement as combr
from itertools import permutations as perm
print(list(comb('ABC', 2)))
print(list(combr('ABC', 2)))
print(list(perm('ABC', 2)))
```

```
[('A', 'B'), ('A', 'C'), ('B', 'C')]
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C', 'C')]
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
```

cycle

In [11]:

```
from itertools import cycle
it = cycle('Python')
for i in range(20):
    print(next(it))
```

```
P
y
t
h
o
n
P
y
t
h
o
n
P
y
t
h
o
n
P
y
```

repeat

In [12]:

```
from itertools import repeat
it = repeat('Linux', 6)
'_'.join(it)
```

Out[12]:

```
'Linux_Linux_Linux_Linux_Linux_Linux'
```

In [13]:

```
from itertools import repeat
from math import pow
list(map(pow, range(10), repeat(2)))
```

Out[13]:

```
[0.0, 1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0]
```

In [14]:

```
from itertools import cycle
from math import pow
list(map(pow, range(10), cycle([1, 2, 3])))
```

Out[14]:

```
[0.0, 1.0, 8.0, 3.0, 16.0, 125.0, 6.0, 49.0, 512.0, 9.0]
```

tee

In [15]:

```
from itertools import tee
it1, it2, it3 = tee('Ala ma kota', 3)
print(next(it1))  # A
print(next(it2))  # A
print(next(it1))  # l
print(next(it1))  # a
print(next(it3))  # A
```

```
A
A
l
a
A
```

Receptury (https://docs.python.org/3/library/itertools.html#itertools-recipes)

take

In [16]:

```
from itertools import islice
it = islice('Ala ma kota', 2, 6)
print(list(it))
```

```
['a', ' ', 'm', 'a']
```

In [17]:

```
from itertools import islice

def take(n, iterable):
    return list(islice(iterable, n))

take(3, 'Python')
```

Out[17]:

```
['P', 'y', 't']
```

flatten

`[[1, 2, 3], [4, 5], [6, 7]]` → `[1, 2, 3, 4, 5, 6, 7]`

In [18]:

```
from itertools import chain

def flatten(list_of_lists):
    return chain.from_iterable(list_of_lists)


list(flatten([[1, 2, 3], [4, 5], [6, 7]]))
```

Out[18]:

```
[1, 2, 3, 4, 5, 6, 7]
```

Menedżer kontekstu with

In [19]:

```
with open('/tmp/log.txt') as f:
    print(f.read())
```

```
[1527444670] executing add2 with args: (4, 9)
[1527444670] executing add3 with args: (1, 2, 3)
[1527446334] executing add2 with args: (4, 9)
[1527446334] executing add3 with args: (1, 2, 3)
```

Można napisać własny *context manager*.

In [20]:

```
class File:
    def __init__(self, filename, mode):
        self.file_obj = open(filename, mode)

    def __enter__(self):
        return self.file_obj

    def __exit__(self, exc_type, exc_value, traceback):
        # print('exc_type:', exc_type)
        # print('exc_value:', exc_value)
        # print('traceback:', traceback)
        self.file_obj.close()
        return True

with File('/tmp/log.txt', 'r') as f:
    print(f.read())
```

```
[1527444670] executing add2 with args: (4, 9)
[1527444670] executing add3 with args: (1, 2, 3)
[1527446334] executing add2 with args: (4, 9)
[1527446334] executing add3 with args: (1, 2, 3)
```

In [ ]:

```
class pobierz:
    def __init__(self, url):
        self.url = url

    def __enter__(self):
        return self

    def get(self):
        import urllib.request
        response = urllib.request.urlopen(self.url)
        data = response.read()
        text = data.decode('utf-8')
        return text

    def __exit__(self, exc_type, exc_value, traceback):
        return False

with pobierz('http://gwoozdz.vot.pl/pan-tadeusz.txt') as data:
    print(data.get())
```

Jeśli `__exit__()` zwraca True, to znaczy, że wyjątek został obsłużony.

In [22]:

```
class Liczba:
    def __enter__(self):
        return 2
    def __exit__(self, *args):
        return True

with Liczba() as liczba:
    print(liczba)
```

```
2
```

Context Manager można utworzyć za pomocą dekoratora i generatora.

In [23]:

```
from contextlib import contextmanager

@contextmanager
def File(filename, mode):
    file_obj = open(filename, mode)
    yield file_obj
    file_obj.close

with File('/tmp/log.txt', 'r') as f:
    print(f.read())
```

```
[1527444670] executing add2 with args: (4, 9)
[1527444670] executing add3 with args: (1, 2, 3)
[1527446334] executing add2 with args: (4, 9)
[1527446334] executing add3 with args: (1, 2, 3)
```

In [24]:

```
from contextlib import contextmanager

@contextmanager
def singleuse():
    print('before')
    yield
    print('after')

cm = singleuse()

with cm:
    pass

# with cm:
#     pass

with singleuse():
    pass

with singleuse():
    pass
```

```
before
after
before
after
before
after
```

Serializacja

Jest to konwersja danych do takiego formatu, który pozwoli je przechować by móc je potem odtworzyć. Tymi danymi mogą być instancje klas.

Zapis

In [25]:

```
import pickle

people = {'Grzesiek': 50, 'Bob': 23, 'Ala': 17}

serial_people = pickle.dumps(people)

print(serial_people)

with open('/tmp/baza.bin', 'wb') as f:
    f.write(serial_people)
```

```
b'\x80\x03}q\x00(X\x08\x00\x00\x00Grzesiekq\x01K2X\x03\x00\x00\x00Bobq\x02K\x17X\x03\x00\x00\x0
0Alaq\x03K\x11u.'
```

Odczyt

In [26]:

```
import pickle

with open('/tmp/baza.bin', 'rb') as f:
    serial_data = f.read()
    data = pickle.loads(serial_data)
    print(data)
```

```
{'Grzesiek': 50, 'Bob': 23, 'Ala': 17}
```

Marynowanie własnego obiektu

In [27]:

```
import pickle

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return '{}({})'.format(self.name, self.age)

ala = Person('Ala', 20)
print(ala)

serial_ala = pickle.dumps(ala)
print(serial_ala)
print(len(serial_ala))
```

```
Ala(20)
b'\x80\x03c__main__\nPerson\nq\x00)\x81q\x01}q\x02(X\x04\x00\x00\x00nameq\x03X\x03\x00\x00\x00A
laq\x04X\x03\x00\x00\x00ageq\x05K\x14ub.'
65
```

65 to dużo, jak na opis jednej osoby. Możemy określić własny sposób na "piklowanie" obiektu.

In [28]:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return '{}({})'.format(self.name, self.age)

    def __getstate__(self):
        return '{}_{}'.format(self.name, self.age)

ala = Person('Ala', 20)

serial_ala = pickle.dumps(ala)
print(serial_ala)
print(len(serial_ala))

with open('/tmp/ala.bin', 'wb') as f:
    f.write(serial_ala)
```

```
b'\x80\x03c__main__\nPerson\nq\x00)\x81q\x01X\x06\x00\x00\x00Ala_20q\x02b.'
40
```

40 to już lepiej, ale teraz nie można "odpiklować".

In [29]:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return '{}({})'.format(self.name, self.age)

    def __getstate__(self):
        return '{}_{}'.format(self.name, self.age)

with open('/tmp/ala.bin', 'rb') as f:
    serial_data = f.read()
    data = pickle.loads(serial_data)
```

```
---------------------------------------------------------------------------
UnpicklingError                           Traceback (most recent call last)
<ipython-input-29-fb4cedfdf293> in <module>()
     12 with open('/tmp/ala.bin', 'rb') as f:
     13     serial_data = f.read()
---> 14     data = pickle.loads(serial_data)

UnpicklingError: state is not a dictionary
```

Zatem trzeba funkcję `pickle.loads` tego nauczyć.

In [ ]:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return '{}({})'.format(self.name, self.age)

    def __getstate__(self):
        return '{}_{}'.format(self.name, self.age)

    def __setstate__(self, state):
        name, age = state.split('_')
        self.name = name
        self.age = int(age)

with open('/tmp/ala.bin', 'rb') as f:
    serial_data = f.read()
    data = pickle.loads(serial_data)
    print(data)
    print(data.__dict__)
```

`json`

In [ ]:

```
import json
people = {'Grzesiek': 50, 'Bob': 23, 'Ala': 17}

json.dumps(people)

with open('/tmp/bazka.json', 'wt') as f:
    json.dump(people, f)
```

`-m pickle`

Debugowanie

pdb (https://docs.python.org/3/library/pdb.html)

`python3 -m pdb myscript.py`

Komendy w trybie interaktywnym (https://docs.python.org/3/library/pdb.html?highlight=pdb#debugger-commands)

Najczęściej `list`, `next`, `step`, `continue`, `p locals()`

In [30]:

```
def bad_loop(n):
    for i in range(n):
        if i == 5:
            1/0

def run_bad_loop(x):
    bad_loop(x)

if __name__ == '__main__':
    run_bad_loop(10)
    print('Bye bye')
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-30-cb2e7a2c9c24> in <module>()
      8
      9 if __name__ == '__main__':
---> 10     run_bad_loop(10)
     11     print('Bye bye')

<ipython-input-30-cb2e7a2c9c24> in run_bad_loop(x)
      5
      6 def run_bad_loop(x):
----> 7     bad_loop(x)
      8
      9 if __name__ == '__main__':

<ipython-input-30-cb2e7a2c9c24> in bad_loop(n)
      2     for i in range(n):
      3         if i == 5:
----> 4             1/0
      5
      6 def run_bad_loop(x):

ZeroDivisionError: division by zero
```

Stop! Debug time!

Możemy wstawić w dowolnym miejscu w programie by go tam zatrzymać i zacząć debugować.
`import pdb; pdb.set_trace()`

Ciekawostka: Od Pythona 3.7 będzie komenda `breakpoint()` (PEP-553 (https://www.python.org/dev/peps/pep-0553/)).

ZADANIE dekorator "debugowy"

Testowanie

doctest

In [31]:

```
def add(a, b):
    '''
    >>> add(2, 3)
    5
    >>> add(-4, 6)
    2
    >>> add('3', 1)
    Traceback (most recent call last):
    ...
    TypeError: must be str, not int
    '''
    return a + b

import doctest
doctest.testmod()
```

Out[31]:

```
TestResults(failed=0, attempted=3)
```

- `python add.py` - Jeśli wszystko będzie w porządku, to nic nie zostanie wypisane.
- `python add.py -v` - Dostaniemy podsumowanie mimo poprawności testowanych funkcji.

Działa również na plikach tekstowych. Przykład 08/docs_add.txt

```
python3 -m doctest docs_add.txt
```

```
TREŚĆ PLIKU docs_add.txt:

Cześć Pracowniku,

Wymyśliłem sobie funkcję, która dodaje dwie liczby. Używałoby się jej tak:
Najpierw import:

    >>> from add import add

Potem tak używamy:

    >>> add(3, 42)
    7

    >>> add(1, 1)
    2

Można też ujemne liczby dodawać:

    >>> add(-4, -9)
    -13

Co myślisz o tym?

Pozdrawiam,
Twój Szef
```

unittest

In [32]:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

# unittest.main()
```

1. Dziedziczenie po unittest.TestCase
2. Metody zaczynają się od `test_`

- `self.assertEqual`
- `self.assertNotEqual`
- `self.assertTrue`
- `self.assertFalse`
- `self.assertRaises`
- `self.assertAlmostEqual`
- `self.assertNotAlmostEqual`
- `self.assertRegex`
- `self.assertNotRegex`
- `self.assertRaisesRegex`

Metody `setUp` oraz `tearDown` zostaną wykonane przed i po każdym teście.

In [33]:

```
import unittest

class TestZeroDivision(unittest.TestCase):

    def setUp(self):
        print('setUp')

    def tearDown(self):
        print('tearDown')

    def test_division(self):
        with self.assertRaises(ZeroDivisionError):
            1/0

# unittest.main()
```

Zamiast `unittest.main()` można za pomocą `python3 -m unittest mytests.py`

Bez argumentu (`python3 -m unittest`) uruchamia się TestDiscovery (https://docs.python.org/3/library/unittest.html#test-discovery).

Współbieżność

threading.Thread (https://docs.python.org/3/library/threading.html#threading.Thread)

Reprezentuje pewną aktywność, która jest wykonywana współbieżnie w obrębie jednego procesu. Są dwa sposoby sprecyzowania takich aktywności:

- Poprzez przekazanie wykonywalnego obiektu
- lub przez przesłonięcie metody `run()`

UWAGA Można przesłonić tylko ```__init__()``` oraz ```run()```.

Start za pomocą `start()`. Uruchomiona zostanie metoda `run()` w osobnym wątku.

Po wystartowaniu, `Thread` jest `alive` tak długo jak działa metoda `run()`. Można sprawdzić za pomocą `is_alive()`.

Wywołanie `join()` na wątku powoduje oczekiwanie na jego zakończenie.

Wątek może zostać oznaczony jako daemon. Gdy w programie zostaną wyłącznie wątki oznaczone tą flagą - program wtedy kończy działanie, a owe demony zostają nagle unicestwione.

UWAGA Najpewniej nie zwolnią zasobów.

In [34]:

```
import threading
import time

def hello10():
    thread = threading.current_thread()
    print('Worker name', thread.name)
    for i in range(10):
        print(i)
        time.sleep(0.1)

thread = threading.Thread(target=hello10)

print('start')
thread.name = 'countdown'
thread.start()
thread.join(timeout=0.3)
print('is_alive', thread.is_alive())
print('after join')
```

```
start
Worker name countdown
0
1
2
is_alive True
after join
3
4
5
6
7
8
9
```

`start()` (https://docs.python.org/3/library/threading.html#threading.Thread.start) - Można wywołać tylko raz. Inaczej rzuci wyjątek `RuntimeError` (https://docs.python.org/3/library/exceptions.html#RuntimeError).

`join(timeout=None)` (https://docs.python.org/3/library/threading.html#threading.Thread.join) - Zwraca zawsze `None`. Trzeba potem sprawdzić `is_alive()`.

name

GIL (https://docs.python.org/3/glossary.html#term-global-interpreter-lock)

threading.Lock() (https://docs.python.org/3/library/threading.html#threading.Lock)

acquire(blocking=True, timeout=-1)

release()

with

In [35]:

```
import threading
import time

lock = threading.Lock()

def hello(n):
    # lock.acquire()
    with lock:
        for i in range(n):
            print(i)
            time.sleep(0.1)
            if i == 3:
                1/0
    # lock.release()

thread1 = threading.Thread(target=hello, args=(10,))
thread2 = threading.Thread(target=hello, args=(10,))

thread1.start()
thread2.start()
```

```
0
1
2
```

In [36]:

```
import threading
import time

lock = threading.Lock()

def hello(n):
    lock.acquire()

    try:
        for i in range(n):
            print(i)
            time.sleep(0.01)
            if i == 5:
                1/0
    except:
        print('wyjatek!')
        return
    else:
        print('bez wyjatku!')
    finally:
        print('finally')
        lock.release()

thread1 = threading.Thread(target=hello, args=(10,))
thread2 = threading.Thread(target=hello, args=(10,))


thread1.start()
thread2.start()
```

```
0
1
2
3
3
4
5
wyjatek!
finally
0
1
2
3
4
5
wyjatek!
finally
0
1

Exception in thread Thread-6:
Traceback (most recent call last):
  File "/usr/lib/python3.6/threading.py", line 916, in _bootstrap_inner
    self.run()
  File "/usr/lib/python3.6/threading.py", line 864, in run
    self._target(*self._args, **self._kwargs)
  File "<ipython-input-35-0f0efbc0e212>", line 13, in hello
    1/0
ZeroDivisionError: division by zero


2
```

threading.Timer (https://docs.python.org/3/library/threading.html#timer-objects)

In [37]:

```
import threading

def hello():
    print('Hello')

t = threading.Timer(2.0, hello)
t.start()
```

```
3

Exception in thread Thread-7:
Traceback (most recent call last):
  File "/usr/lib/python3.6/threading.py", line 916, in _bootstrap_inner
    self.run()
  File "/usr/lib/python3.6/threading.py", line 864, in run
    self._target(*self._args, **self._kwargs)
  File "<ipython-input-35-0f0efbc0e212>", line 13, in hello
    1/0
ZeroDivisionError: division by zero
```

threading.Barrier (https://docs.python.org/3/library/threading.html#barrier-objects)

In [44]:

```
import threading
import time

barrier = threading.Barrier(2)

def hello(n, wait_for):

    time.sleep(wait_for)
    print('I am waiting for others')
    barrier.wait()
    print('OK! Let\'s go!')

    for i in range(n):
        print(i)
        time.sleep(0.1)

thread1 = threading.Thread(target=hello, args=(10, 0.0))
thread2 = threading.Thread(target=hello, args=(10, 4.0))

thread1.start()
thread2.start()
```

```
I am waiting for others
I am waiting for others
OK! Let's go!
0
OK! Let's go!
0
1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
9
9
```

`multiprocessing` (https://docs.python.org/3/library/multiprocessing.html)

Procesy zamiast wątków. Więcej używanych zasobów, ale lepsza korzyść z kilku procesorów.

In [39]:

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(15) as p:
        print(p.map(f, [1, 2, 3]))
```

```
[1, 4, 9]
```

In [40]:

```
import os
from multiprocessing import Process
import multiprocessing
import time

def odd_to_one(x):
    pid = os.getpid()
    time.sleep(0.1 * x)
    cp = multiprocessing.current_process()
    print('{} → {}: {} → {}'.format(cp.name, pid, x, x % 2))

numbers = list(range(10))

procs = []

for n in numbers:
    process = Process(target=odd_to_one, args=(n,))
    process.name = 'Proces({})'.format(n)
    process.start()
    procs.append(process)



for p in procs:
    p.join()
    print(p.exitcode)
```

```
Proces(0) → 1580: 0 → 0
Proces(1) → 1581: 1 → 1
Proces(2) → 1584: 2 → 0
0
0
0
Proces(3) → 1585: 3 → 1
Proces(4) → 1586: 4 → 0
0
Hello
0
Proces(5) → 1587: 5 → 1
Proces(6) → 1588: 6 → 0
0
0
Proces(7) → 1589: 7 → 1
Proces(8) → 1590: 8 → 0
0
0
Proces(9) → 1591: 9 → 1
0
```

`current_process()`

`join()`

terminate()

> Ważne jest by wykonać **join()** po **terminate()**. Dlaczego?

.exitcode

Logowanie

In [41]:

```
import multiprocessing
import logging
import sys
import time

def worker():
    print('Wykonuję pracę, bo jestem workerem!')
    sys.stdout.flush()
    time.sleep(0.1)

multiprocessing.log_to_stderr(logging.DEBUG)
process = multiprocessing.Process(target=worker)
process.start()
process.join()
```

```
[INFO/Process-26] child process calling self.run()
```

```
Wykonuję pracę, bo jestem workerem!
```

```
[INFO/Process-26] process shutting down
[DEBUG/Process-26] running all "atexit" finalizers with priority >= 0
[DEBUG/Process-26] running the remaining "atexit" finalizers
[INFO/Process-26] process exiting with exitcode 0
```

Zadanie: Brzuszki

In [43]:

```
# Zadanie 1: Przeanalizuj kod
# Zadanie 2: Wykorzystaj metodę str.count() do obliczenia *result*

def brzuszki(n):
    '''
    >>> brzuszki('1234567890')
    5
    >>> brzuszki('100')
    2
    >>> brzuszki('654')
    1
    >>> brzuszki('11111')
    0
    >>> brzuszki('4567')
    1
    '''
    b = {'6': 1, '8': 2, '9': 1, '0': 1}
    result = sum(map(lambda x: b.get(x, 0), n))
    return result

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```