

# Kurs rozszerzony języka Python

## Wykład 6.

Marcin Młotkowski

10 listopada 2017

# Plan wykładu

## 1 Callable objects

## 2 Wątki

- Wprowadzenie
- Dzielenie się zasobami
- Zmienne warunkowe

## 3 Inne biblioteki

# Plan wykładu

## 1 Callable objects

## 2 Wątki

- Wprowadzenie
- Dzielenie się zasobami
- Zmienne warunkowe

## 3 Inne biblioteki

Wszystko jest obiektem.

Wszystko jest obiektem.

A funkcje?

# Przykład

```
def foo(x):  
    return 2*x
```

```
dir(foo)
```

# Przykład

```
def foo(x):  
    return 2*x
```

```
dir(foo)
```

```
["__call__", "__class__", "__closure__", "__code__", "__defaults__",  
 "__name__", ...]
```

## Elementy wykonywalne (ang. *callable*)

Są to te elementy języka Python, które można wywoływać jak funkcję.



## Elementy wykonywalne (ang. *callable*)

Są to te elementy języka Python, które można wywoływać jak funkcję.

Przykłady:

- funkcje i metody wbudowane;
- funkcje zdefiniowane przez użytkownika;
- metody obiektu;
- klasy (tworzenie nowego obiektu);
- obiekty implementujące metodę `__call__`.

# Przykład obiektu wykonywalnego

```
class Potrojenie:  
    def __call__(self, n):  
        return self.podwojenie(n) + n  
    def podwojenie(self, n):  
        return n + n
```

```
trojka = Potrojenie()  
for n in range(4):  
    print("{} 3*{} = {}".format(n, trojka(n), trojka(n)))
```

# Własny licznik

## Potrzebuję licznika

```
licznik() # zwraca 1  
licznik() # zwraca 2  
licznik() # zwraca 3
```

# Własny licznik

## Potrzebuję licznika

```
licznik() # zwraca 1  
licznik() # zwraca 2  
licznik() # zwraca 3
```

```
class Licznik:  
    def __init__(self):  
        self.licznik = 0  
    def __call__(self):  
        self.licznik += 1  
        return self.licznik
```

```
licznik = Licznik()
```

# Plan wykładu

## 1 Callable objects

## 2 Wątki

- Wprowadzenie
- Dzielenie się zasobami
- Zmienne warunkowe

## 3 Inne biblioteki

# Wstęp

## Z Wikipedii:

Wątek (ang. thread) — to jednostka wykonawcza w obrębie jednego procesu, będąca kolejnym ciągiem instrukcji wykonywanym w obrębie tych samych danych (w tej samej przestrzeni adresowej).

Wątki tego samego procesu korzystają ze wspólnego kodu i danych, mają jednak oddzielne stosy.

# Po co używać wątków

- zrównoleglenie wolnych operacji wejścia/wyjścia (ściągnięcie pliku/obsługa interfejsu)
- jednoczesna obsługa wielu operacji, np. serwery WWW

# Przykładowe obliczenie programu dwuwątkowego

## Przykład 1.

Wątek I	Wątek II
$i = i + 1$	
<code>print i</code>	
$i = i + 1$	
<code>print i</code>	
$i = i + 1$	
<code>print i</code>	
	$i = i + 1$
	<code>print i</code>
	$i = i + 1$
	<code>print i</code>
	$i = i + 1$
	<code>print i</code>

## Przykład 2.

Wątek I	Wątek II
$i = i + 1$	
<code>print i</code>	$i = i + 1$
	<code>print i</code>
$i = i + 1$	
	$i = i + 1$
<code>print i</code>	
	<code>print i</code>
$i = i + 1$	
	$i = i + 1$
<code>print i</code>	
	<code>print i</code>



# Moduły wątków w Pythonie

- `thread` (3.\*: `_thread`): niskopoziomowa biblioteka
- `threading`: wysokopoziomowa biblioteka, korzysta z `thread`;

# Moduły wątków w Pythonie

- `thread (3.*: _thread)`: niskopoziomowa biblioteka
- `threading`: wysokopoziomowa biblioteka, korzysta z `thread`;
- `dummy_thread`
- `dummy_threading`
- `multiprocessing`
- `concurrent.futures`

# Jak korzystać z wątków

moduł `threading`

`class Thread:`

```
def run(self):  
    """Operacje wykonywane w wątku"""
```

```
def start(self):  
    """Wystartowanie obliczeń w wątku"""
```

# Przykładowe zadanie

Zasymulowanie za pomocą wątków biegaczy w maratonie.

# Implementacja klasy biegaczy

```
import threading

total_distance = 0

class runner(threading.Thread):
    def __init__(self, nr_startowy):
        self.numer = nr_startowy
        threading.Thread.__init__(self)
```

# Implementacja biegu

```
class runner, cd
def run(self):
    global total_distance
    dystans = 42195
    while dystans > 0:
        dystans = dystans - 1
        total_distance = total_distance + 1
        if dystans % 10000 == 0:
            print ("Zawodnik nr {0}".format(self.numer))
    print ("Zawodnik {0} na mecie".format(self.numer))
```

# Rozpoczęcie biegu

```
r1 = runner(1)
r2 = runner(2)

r1.start()
r2.start()

r1.join()
r2.join()

print ("koniec wyścigu, dystans {}".format(total_distance))
```

# Rola `.join`

- Główny program to też wątek, więc po wywołaniu

`r1.start()`

są dwa wątki

- `r1.join()` oznacza, że wątek nadrzędny będzie czekał na zakończenie wątku `r1`



# Tworzenie wątków

## Podsumowanie

Wątki tworzymy dziedzicząc po klasie Thread.

# Inny sposób tworzenia wątków

```
wątek = Thread(target=callable, args=sekwencja)
```

## Inny sposób tworzenia wątków

```
wątek = Thread(target=callable, args=sekwencja)
```

```
import threading  
wątek = threading.Thread(target=pow, args=(2, 10))
```

# Dostęp do wspólnej zmiennej wątków

## Przypomnienie

```
total_distance = 0
```

```
class runner(threading.Thread):
```

```
    ...
```

```
        total_distance = total_distance + 1
```

```
print total_distance
```

# Zagadka

Jaka jest wartość zmiennej `total_distance`?

# Zagadka

Jaka jest wartość zmiennej `total_distance`?

Teoria

$$2 * 42195 = 84390$$

# Zagadka

Jaka jest wartość zmiennej `total_distance`?

Teoria

$2 * 42195 = 84390$

Praktyka

54390

74390

83464

...

# Operacje atomowe?

```
i = i + 1
```

```
LOADFAST 0
```

```
LOAD_CONST 1
```

```
BINARY_ADD
```

```
STORE_FAST 0
```



# Operacje atomowe?

$i = i + 1$

```
LOADFAST 0  
LOAD_CONST 1  
BINARY_ADD  
STORE_FAST 0
```

$i = i + 1$

```
LOADFAST 0  
LOAD_CONST 1  
BINARY_ADD  
STORE_FAST 0
```

# Blokady

## Klasa Lock

```
lock = Lock()

def run(self):
    global lock
    ...
    lock.acquire()
    total_distance = total_distance + 1
    lock.release()
```

# Inne blokady

## RLock

Wątek może założyć blokadę dowolną liczbę razy, i tyleż razy musi ją zwolnić. Bardzo spowalnia program.

## Semaphore

Blokadę można założyć ustaloną liczbę razy:

```
sem = Semaphore(3)
sem.acquire()
sem.acquire()
sem.acquire()
sem.acquire() # blokada
```

# Czekanie na zasób

Jeden wątek (barman) nalewa mleko do szklanki, drugi (klient) czeka na napełnienie szklanki do pełna i wypija mleko.

# Implementacja picia mleka

```
lck = Lock()
```

## Nalewanie

```
lck.acquire()  
for i in range(5):  
    szklanka_mleka = szklanka_mleka + 1  
lck.release()
```

## Wypijanie

```
while szklanka_mleka != 5: pass  
lck.acquire()  
while szklanka_mleka > 0:  
    szklanka_mleka = szklanka_mleka - 1  
lck.release()
```

# Implementacja picia mleka

```
lck = Lock()
```

## Nalewanie

```
lck.acquire()  
for i in range(5):  
    szklanka_mleka = szklanka_mleka + 1  
lck.release()
```

## Wypijanie

```
while szklanka_mleka != 5: pass  
lck.acquire()  
while szklanka_mleka > 0:  
    szklanka_mleka = szklanka_mleka - 1  
lck.release()
```

# Zmienne warunkowe

Mechanizm który pozwala na usypianie i budzenie wątków.

# Implementacja

```
lck = threading.Condition()
```

## Konsumpcja

```
lck.acquire()  
while szklanka_mleka != 5:  
    lck.wait()  
while szklanka_mleka > 0: szklanka_mleka = szklanka_mleka - 1  
lck.release()
```

## Nalewanie

```
lck.acquire()  
for i in range(5):  
    szklanka_mleka = szklanka_mleka + 1  
lck.notify()  
lck.release()
```



# Zmienne warunkowe

- Zmienne warunkowe są zmiennymi działającymi jak blokady (`acquire()`, `release()`);
- metoda `wait()` zwalnia blokadę i usypia bieżący wątek;
- metoda `notify()` budzi jeden z uśpionych wątków (na tej zmiennej warunkowej), `notifyAll()` budzi wszystkie uśpione wątki.

# Wady takiego mechanizmu

- jest tylko jedna szklanka, można do niej tylko nalewać albo tylko z niej pić;
- barman nie może nalać więcej szklanek na zapas i iść do domu

# Bezpieczne struktury

## Thread-safety

Struktura danych jest *thread-safe*, jeśli może być bezpiecznie używana w środowisku wielowątkowym.

# Struktury danych do programów wielowątkowych

## Klasa Queue:

- Jest to kolejka FIFO, thread-safe;
- Konstruktor: `Queue(rozmiar)`
- pobranie elementu (z usunięciem): `.get()`; gdy kolejka jest pusta zgłasza wyjątek `Empty`
- `.get(True)`: gdy kolejka jest pusta, wątek jest usypiany;
- umieszczenie elementu: `.put(element)`, gdy kolejka jest pełna to zgłaszany jest wyjątek `Full`;
- umieszczenie elementu: `.put(element, True)`, gdy kolejka jest pełna wątek jest usypiany;
- `.full()`, `.empty()`

# Warianty klasy Queue

- LifoQueue
- PriorityQueue

## Bar mleczny: inne rozwiązanie

```
def mlekopij(q):  
    while True:  
        szklanka_mleka = q.get()  
        q.task_done()
```

```
q = queue.Queue()  
m = threading.Thread(target=mlekopij, args=(q))  
m.start()
```

```
for mleczko in bar_mleczny:  
    q.put(mleczko)  
  
q.join()  
m.join()
```

# Plan wykładu

## 1 Callable objects

## 2 Wątki

- Wprowadzenie
- Dzielenie się zasobami
- Zmienne warunkowe

## 3 Inne biblioteki







Źródło: Wikimedia

# Efektywność standardowych wątków

## Global Interpreter Lock (GIL)

Tylko jeden wątek ma dostęp do bytencodu.

## Operacje I/O

GIL jest zwalniany podczas czekania na operacje We/Wy.

# Biblioteka multiprocessing

- podobna do `threading`;
- oparta o procesy, nie o wątki; więc nie powinno być problemu z GIL'em.

# Tworzenie procesów

```
import multiprocessing
```

```
p = multiprocessing.Process(target=callable, args=sequence)
```

# Tworzenie procesów

```
import multiprocessing  
  
p = multiprocessing.Process(target=callable, args=sequence)
```

Mi nie zadziało w Pythonie 3.1.2 :-)

# Tworzenie procesów

```
import multiprocessing  
  
p = multiprocessing.Process(target=callable, args=sequence)
```

Mi nie zadziało w Pythonie 3.1.2 :-)

Ale zadziało w 3.2.3 :-)

# Process

```
pr = Process(target=foo, args=(1,2,3))  
pr.start()  
pr.join()
```

# Co mi też zadziało

Pule wątków

Biblioteka Pool



# Co mi też zadziało

Pule wątków

Biblioteka Pool

Liczby Fibonacciego

Algorytm rekurencyjny, pierwsze wywołanie dzieli na dwa procesy.

# Implementacja wieloprocessorowa

```
def fib(n):  
    if n < 2: return 1  
    return fib(n - 1) + fib(n - 2)
```

```
from multiprocessing import Pool
```

```
def pfib(n):  
    if n < 2: return 1  
    p = Pool(2)  
    result = p.map(fib, [n-1, n-2])  
    return sum(result)
```

# Wymiana informacji między procesami

`multiprocessing.Value`

```
val = Value("i", 0)
```

```
...
```

```
val.value = 512
```

# Wymiana informacji między procesami

## multiprocessing.Value

```
val = Value("i", 0)
...
val.value = 512
```

## multiprocessing.Queue

```
q = Queue()
...
q.put(wartosc)
q.get()
```

# Komunikacja synchroniczna

```
par_conn, child_conn = Pipe()
...
child_conn.send([1, "dwa", 3.0])
...
print(par_conn.recv())
```

# I jeszcze jedna biblioteka

## `concurrent.futures`

- automatyczny wybór między wątkami a procesami;
- Od wersji 3.2