



Szybkie wprowadzenie do Pythona

Dawid Kosiński

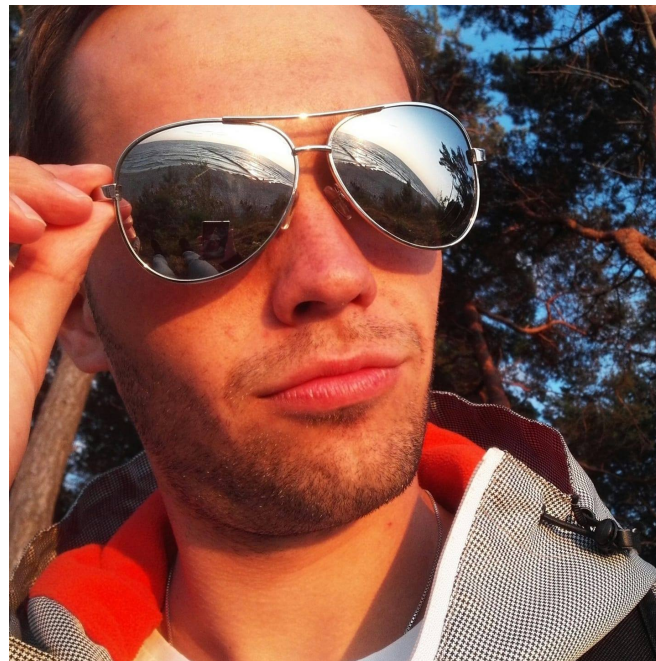


Agenda

1. Co to Python?
2. Co to są dokumentacje i dlaczego są takie ważne?
3. Podstawowe typy zmiennych
4. Pętle
5. Warunki
6. Stackoverflow jako twój drugi wujek
7. A co jak nie działa? Czyli podstawowa umiejętność czytania błędów oraz łapanie wyjątków
8. Łapanie wyjątków
9. Funkcje

O mnie

- Dawid Kosiński
- Zainteresowany programowaniem od 2008r.
- Mgr. inż z informatyki (politechnika białostocka - 2018r.)
- Zainteresowany Python'em od 2015r.
- Inne znane języki: C, C++, Java
- Aktualna praca: Optimo development (od 2018r.)
- Inne zainteresowania: sport, psychologia, matematyka
- Socials:
 - strona firmowa: <https://webnaq.pl/>
 - facebook: <https://www.facebook.com/itsDKey/>
 - instagram: <https://www.instagram.com/itsdkey/?hl=pl>
 - Github: <https://github.com/itsdkey>





Co to Python

- Język wysokiego poziomu programowania
- Język Open-source
- Oficjalna dokumentacja: <https://docs.python.org/3/>
- Kod organizowany za pomocą wcięć a nie klamer
- Brak średników



Co to dokumentacja i dlaczego jest ważna?

- Opisuje co dokładnie dana paczka robi i do czego można ją zastosować
- Podaje parę przykładów zastosowania
- Pokrywa najczęściej popełniane błędy (nie zawsze to jest zawarte)
- Jest to swego rodzaju ścieżka jak wykorzystać daną paczkę w swoim projekcie.



Podstawowe typy zmiennych

- “pustka”:
 - None
- Prawda/Falsz:
 - True/False
- znakowe:
 - string, np. ‘test’, ‘to jest dłuższe zdanie’, ‘t’
- liczbowe:
 - całkowite - integer (skrót: int), np. 1
 - zmiennoprzecinkowe:
 - float, np. 1.1, 2.2 -> $1.1 + 2.2 = 3.30000000000000000003$
 - Decimal, np. Decimal(‘1.1’) -> $\text{Decimal(‘1.1’)} + \text{Decimal(‘2.2’)} = \text{Decimal(‘3.3’)}$
- “zbiorowe”:
 - listy, np. [1, 2, 3, 4, 5], [[1, 2], [3, 4]], [‘a’, ‘b’, ‘c’], [1, 2, 3, ‘test’], [None, 1, 2, True]
 - słowniki: {‘nazwa’: 12, ‘klucz’: ‘wartość’}



Pętle

For:

```
for ... in zbiór :  
    działanie()
```

While:

```
while <warunek> :  
    print(1)
```

Przykłady:

- **for** value **in** [1,2,3,4] :
 print(value)
- **while** True :
 print('nieskończona pętla')
- **while** x<10 :
 print(x)
 x += 1



Instrukcje warunkowe

- if-elif-else (jeśli - lub jeśli - w przeciwnym wypadku):

- **if** <wyrażenie> :
 <działanie>
elif <wyrażenie> :
 <działanie>
else:
 <działanie>

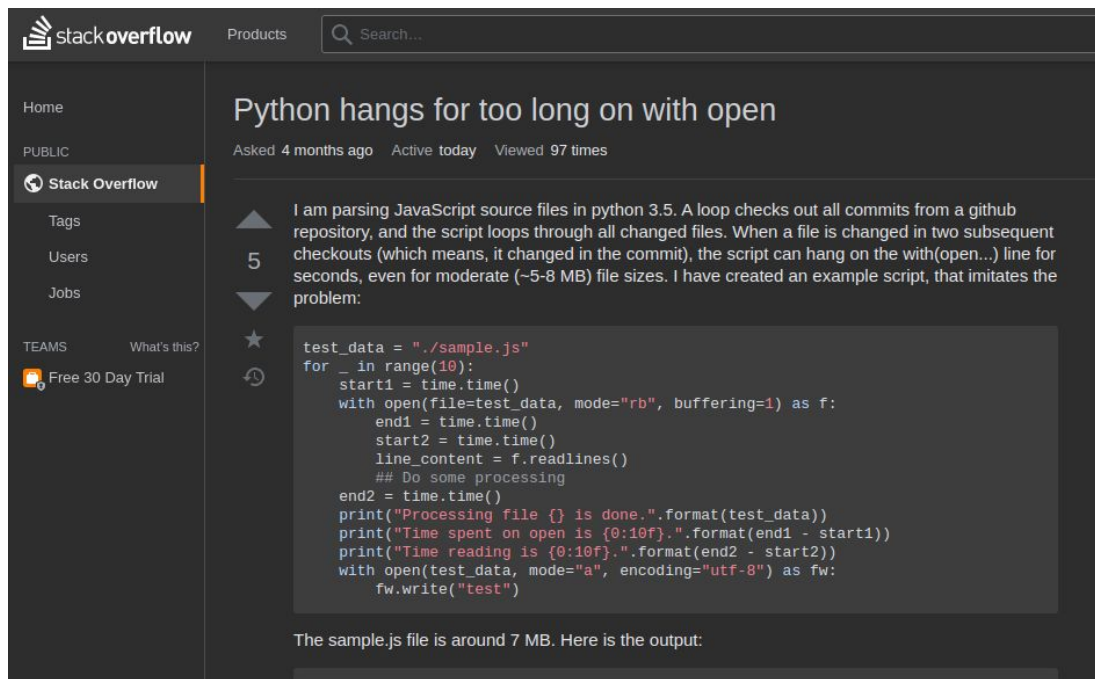
- przykład:

- **if** a > 0 :
 print('liczba dodatnia')
elif a < 0 :
 print('liczba ujemna')
else :
 print('zero')

- logika matematyczna:

- and (pol. "i")
 - 1 and 1 => 1
 - 1 and 0 => 0
 - 0 and 1 => 0
 - 0 and 0 => 0
- or (pol. "lub")
 - 1 or 1 => 1
 - 1 or 0 => 1
 - 0 or 1 => 1
 - 0 or 0 => 0

Stackoverflow



The screenshot shows a Stack Overflow page for a question titled "Python hangs for too long on with open". The page is in a dark theme. On the left is a sidebar with navigation links: Home, PUBLIC, Stack Overflow (selected), Tags, Users, Jobs, TEAMS, What's this?, and a Free 30 Day Trial offer. The main content area shows the question title, its metadata (Asked 4 months ago, Active today, Viewed 97 times), and the question body. The question body contains a paragraph describing a problem with parsing JavaScript source files in Python 3.5, followed by a code block containing a Python script. The script uses a loop to process files, and the user reports that the script hangs on the 'with open' line for seconds, even for moderate file sizes. Below the code block, the user mentions that the sample.js file is around 7 MB and shows the output.

stackoverflow Products Search...

Home

PUBLIC

Stack Overflow

Tags

Users

Jobs

TEAMS What's this?

Free 30 Day Trial

Python hangs for too long on with open

Asked 4 months ago Active today Viewed 97 times

I am parsing JavaScript source files in python 3.5. A loop checks out all commits from a github repository, and the script loops through all changed files. When a file is changed in two subsequent checkouts (which means, it changed in the commit), the script can hang on the with(open...) line for seconds, even for moderate (~5-8 MB) file sizes. I have created an example script, that imitates the problem:

```
test_data = "./sample.js"
for _ in range(10):
    start1 = time.time()
    with open(file=test_data, mode="rb", buffering=1) as f:
        end1 = time.time()
        start2 = time.time()
        line_content = f.readlines()
        ## Do some processing
    end2 = time.time()
    print("Processing file {} is done.".format(test_data))
    print("Time spent on open is {0:10f}.".format(end1 - start1))
    print("Time reading is {0:10f}.".format(end2 - start2))
    with open(test_data, mode="a", encoding="utf-8") as fw:
        fw.write("test")
```

The sample.js file is around 7 MB. Here is the output:



A co jeśli nie działa?

- Należy wpierw przeczytać co program zwrócił za błąd
- Dzięki przeczytaniu jesteśmy w stanie zlokalizować gdzie nastąpił błąd (interpreter zwróci nam, w której linii nastąpiła usterka)
- Dzięki stack trace'owi (stos wywołań) jesteśmy w stanie określić co wykonywało się po sobie (jaka metoda wykonała inna metoda, takie trochę "kopanie głębiej")
- Wraz z doświadczeniem, czytanie błędów stanie się bardziej klarowne



Łapanie wyjątków

- **try:**
 wyrażenie
except <wyjątek> :
 co_zrobić_gdy_wystąpił_wyjątek
else:
 co_zrobić_gdy_nie_było_wyjątku
finally:
 zawsze_wykonaj_cos()

Przykład:

- **try:**
 variable = int(value)
except ValueError :
 print('to nie jest liczba')
else:
 variable += 1



f-string

- składnia:
 - `value = 10`
 - `value = f"to jest juz f-string co wyświetli zmienna value: {value}"`
 - wynik: to jest już f-string co wyświetli zmienną value: 10

F-string (skrót od 'formatted string') jest odmianą zwykłego string'a. Umożliwia nam 'wstrzykiwanie' zmiennych. Aby to zrobić należy zmienną umieścić pomiędzy nawiasami klamrowymi { ... }. Podczas działania programu te wystąpienia zostaną zastąpione wartościami kryjącymi się pod tymi zmiennymi. Takie konstrukcja umożliwia prostsze budowanie ciągów tekstowych, jest czytelniejsze niż inne dostępne metody, np. konstrukcja `%s, '{}'.format(...)`

Więcej na ten temat: <https://www.python.org/dev/peps/pep-0498/>



funkcje

- budowa:
 - **def** nazwa(parametr1, parametr2, *args, **kwargs):
....

Funkcje Pythonowe są obiektami umożliwiającymi ponowne wykorzystanie już istniejącego kodu. Jej deklaracja zaczyna się od słowa kluczowego “def”. Następnie podajemy nazwę funkcji. Nazwa powinna jednoznacznie określać co dany fragment kodu robi (dobra konwencja). Po nazwie, w zwykłych nawiasach, deklarujemy parametry tej funkcji. Są to tzw. zmienne lokalne które będą wykorzystywane w danej funkcji. Deklaracja parametrów umożliwia nam przekazanie zmiennych z zewnątrz do środka funkcji. Ilość parametrów jest opcjonalna - może być zero, wtedy po prostu mamy puste nawiasy - (), lub wiele.



Eat, sleep, code, repeat

