Python: podstawy programowania

Dzień 3 | v. 1.1.1

Coders Lab
SZKOŁA PROGRAMOWANIA

Virtualenv

virtualenv

Czym jest środowisko wirtualne?

Narzędzie pip instaluje biblioteki w systemie. Co, jeśli pracujemy nad dwoma projektami i w każdym z nich używamy różnych wersji danej biblioteki?

W tym celu należy użyć **środowiska wirtualnego**.

Środowisko wirtualne, to wydzielona „piaskownica”, w której trzymamy interpreter Pythona i biblioteki, które on widzi.

Można mieć zainstalowanych wiele środowisk wirtualnych, w każdym możemy mieć wiele bibliotek.

Robimy to, żeby odseparować od siebie projekty, które mogą używać wzajemnie wykluczających się bibliotek.

Środowiska wirtualne zakłada się i uruchamia przy pomocy programu **virtualenv**. Ten program należy zainstalować:

```
sudo apt install virtualenv
```

Po wykonaniu tej komendy i znainstalowaniu programu virtualenv, możemy już zakładać i pracować na wirtualnych środowiskach.

virtualenv

Zakładanie środowiska wirtualnego

Aby założyć środowisko wirtualne, musimy wydać komendę:

Python 2:

```
virtualenv <nazwa-katalogu>
```

Python 3:

```
virtualenv -p python3 <nazwa-katalogu>
```

np:

```
virtualenv -p python3 env
```

Po wydaniu komendy z przykładu, w katalogu env, mamy założone środowisko wirutalne. Znajduje się tam interpreter Pythona, program pip i tam instalowane będą zewnętrzne biblioteki.

Aby móc używać środowiska wirtualnego, należy je aktywować:

```
source <nazwa-katalogu>/bin/activate
```

Czyli w naszym przykładzie będzie to:

```
source env/bin/activate
```

Zamiast komendy **source** można użyć skrótu: znaku kropki, czyli nasz przykład może wyglądać tak:
`. env/bin/activate`

virtualenv

Korzystanie ze środowiska wirtualnego:

Po aktywowaniu środowiska, możemy instalować biblioteki programem **pip**. Python w środowisku wirtualnym widzi tylko biblioteki zainstalowane w tym środowisku!

Aby deaktywować środowisko wirtualne i wrócić do ustawień systemowych, w oknie terminala, należy wydać komendę:

```
deactivate
```

…albo, zwyczajnie, zamknąć okno. ;-)

W każdym momencie możemy wrócić do środowiska wirtualnego, włączając je ponownie sposobem podanym na poprzednich slajdach.

Obsługa wyjątków

Wyjątki i błędy

Podczas działania programu bardzo często napotykane są sytuacje, które nie powinny się wydarzyć.

Może być to np.:

- Próba użycia nieistniejącej zmiennej,
- Próba odczytu z nieistniejącego pliku,
- Próba matematycznego dodania liczby do tekstu,
- Sięgniecie do nieistniejącego elementu listy,
- I jeszcze wiele innych.

Takie sytuacje w programie nazywane są **wyjątkami**. Python reaguje na wyjątek zgłoszeniem błędu. Weźmy np. następujący przypadek:

```
print("Answer is: " + 42)
```

Wynikiem działania tej linijki kodu będzie następujący błąd:

```
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    print("Answer is: " + 42)
TypeError: Can't convert 'int' object to
str implicitly
```

Wyjątki i błędy

Komunikaty błędów mogą być bardzo rozbudowane. Wszystko zależy od tego, jak „głęboko” był program w momencie wystąpienia błędu.

Weźmy pod uwagę następujący program:

error.py

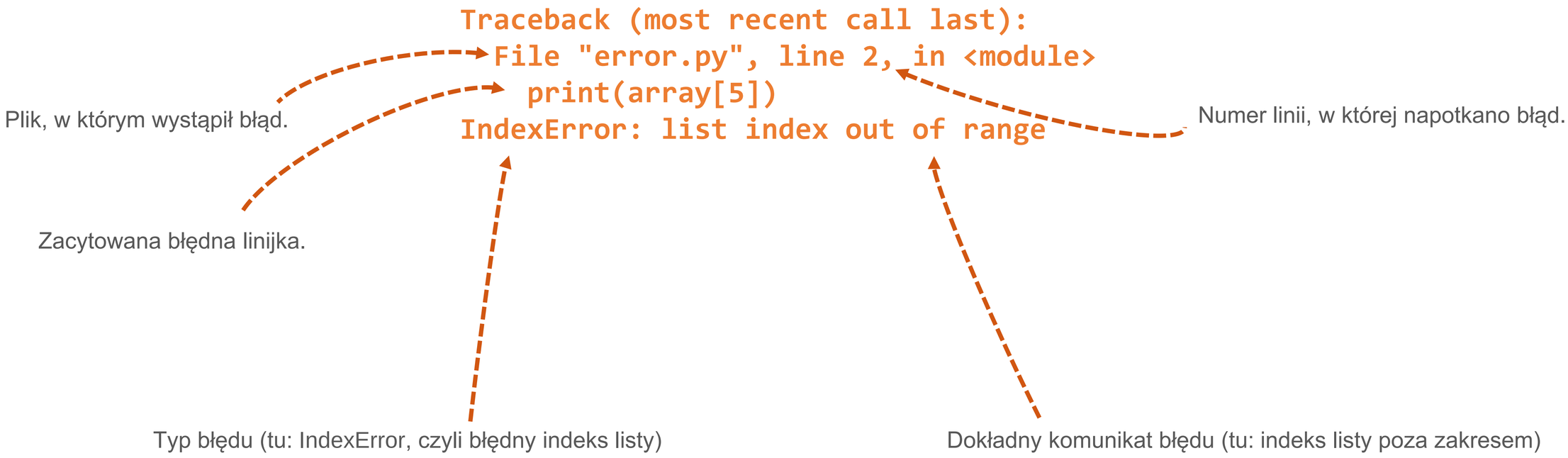
```
array = [1, 2]
print(array[5])
```

Wynik:

```
Traceback (most recent call last):
  File "error.py", line 2, in <module>
    print(array[5])
IndexError: list index out of range
```

Z komunikatu błędu (bardzo krótki w tym przypadku), możemy się dowiedzieć wielu przydatnych informacji na temat natury problemu, napotkanego przez Pythona.

Wyjątki i błędy



Wyjątki i błędy

Wyjątek powoduje przerwanie działania programu!

Aby temu zapobiec możemy **przechwycić wyjątek**. W tym celu musimy użyć pary komend **try – except**:

```
try:
    # tutaj komendy, które mogą
    # spowodować błąd
except <typ-błędu>:
    # tutaj komendy, które reagują
    # na błąd
```

W bloku **try** umieszczamy komendy, co do których spodziewamy się, że mogą spowodować błąd (np. otwieranie pliku z dysku),

W bloku **except** umieszczamy kod, który ma zareagować na wyjątkową sytuację (np. wyświetlić komunikat „plik nie istnieje”. W tej sytuacji program nie zostanie przerwany.

W miejscu **<typ-błędu>** umieszczamy typ błędu, którego się spodziewamy. Jeśli nie chcemy precyzować rodzaju błędu, umieszczamy po prostu **Exception**.

Wyjątki i błędy

Przykłady:

```
array = [1, 2]
try:
    print(array[5])
except Exception:  # Dowolny błąd!
    print("Element nie istnieje.")
```

Wynik:

```
Element nie istnieje!
```

```
array = [1, 2]
try:
    print(array[5])
except KeyError:  # Niewłaściwy typ!
    print("Element nie istnieje.")
```

Wynik:

```
Traceback (most recent call last):
  File "error.py", line 2, in <module>
    print(array[5])
IndexError: list index out of range
```

Wyjątki i błędy

Przykłady:

```
heroes = ["Harry", "Ron", "Hermione"]

while True:
    try:
        x = input("Wprowadź indeks: ")
        index = int(x)
        print(heroes[index])
        break

    except ValueError:
        print("Ups! To nie jest liczba! "
              "Spróbuj ponownie.")
```

W tym przykładzie sprawdzamy, czy nie wystąpił **ValueError**, czyli błąd który może powstać przy próbie konwersji napisu (wprowadzonego z klawiatury) do wartości liczbowej.

Gdy wprowadzimy z klawiatury jakąś wartość, która nie jest liczbą całkowitą, wystąpi ten błąd.

Ale wciąż nie zabezpieczamy się przed wpisaniem poprawnej liczby, która nie jest poprawnym indeksem listy **heroes**! W takim przypadku wystąpi **IndexError**!

Wyjątki i błędy

Przykłady:

```
heroes = ["Harry", "Ron", "Hermione"]

while True:
    try:
        x = input("Wprowadź indeks: ")
        index = int(x)
        print(heroes[index])
        break

    except ValueError:
        print("Ups! To nie jest liczba! "
              "Spróbuj ponownie.")

    except IndexError:
        print("W tablicy heroes nie ma "
              "elementu o tym indeksie!")
```

Aby temu zapobiec, dodajemy kolejny blok kodu, który zabezpieczy nas przed błędnym indeksem listy.

Wynik (przykładowy):

```
Wprowadź indeks: I am Lord Voldemort!
Ups! To nie jest liczba! Spróbuj
ponownie.
Wprowadź indeks: 55
W tablicy heroes nie ma elementu o tym
indeksie!
Wprowadź indeks: 0
Harry
```

Wyjątki i błędy

Czasami chcemy wyrzucić jakiś błąd: np. wtedy, gdy piszemy jakiś moduł, który dostaje dane od użytkownika i jeśli te dane są błędne, chcemy o tym poinformować.

W takiej sytuacji należy użyć polecenia **raise**.

```
raise KeyError
```

Wynik:

```
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    raise KeyError
KeyError
```

Poprzedni komunikat nie przekazuje zbyt wiele treści: wiadomo, że jest to jakiś błąd z jakimś kluczem jakiegoś słownika.

Aby podać więcej informacji, możemy do typu błędu dodać więcej informacji:

```
raise KeyError("Key does not exist!")
```

Wynik:

```
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    raise KeyError("Key does not exist!")
KeyError: 'Key does not exist!'
```

Dekoratory

Dekoratory

Dekorator to wzorzec projektowy, który pozwala nam opakować funkcję w inną funkcję. Funkcja nadrzędna rozszerza możliwości funkcji podstawowej.

Przykład:

- Zamiast sprawdzać w każdej funkcji, czy użytkownik jest zalogowany, piszemy dekorator, który to zrobi i w razie czego nie pozwoli wykonać funkcji.

W Pythonie funkcje są obiektami (wszystko jest obiektem, szczegóły w module „Zaawansowane programowanie w Pythonie"). Zatem można przekazać ciało funkcji jako parametr do innej funkcji, można też zwrócić ciało funkcji jako wynik działania innej.

W skrócie – dekoratory pozwalają na modyfikację wyników różnych funkcji / metod „w locie".

Dekoratory

Wywołanie dekoratora

Dekorator wywołujemy jak zwykłą funkcję. Jako parametr tej funkcji podajemy funkcję, którą chcemy udekorować:

Załóżmy, że chcemy udekorować funkcję **fun**, dekoratorem **decor:**

```
def fun():
    result = None
    # tutaj działanie funkcji
    return result

fun = decor(fun)
```

Tak wygląda wywołanie dekoratora **decor** na funkcji **fun()**

Od tej pory, wyniki funkcji **fun()**, będą modyfikowane w locie przez funkcję **decor()**.

Lukier składniowy

Konstrukcję

```
fun = decor(fun)
```

możemy zastąpić czytelniejszą formą, która ułatwia rozumienie kodu:

```
@decor
def fun():
    result = None
    # tutaj działanie funkcji
    return result
```

W środowisku programistów Pythona taka składnia nazywana jest **lukrem składniowym** (syntactic sugar). Nie wnosi niczego do merytoryki, jedynie ułatwia czytanie kodu.

Dekoratory

Jeśli chcesz dowiedzieć się, jak tworzyć własne dekoratory, zajrzyj do poniższych linków:

https://www.python.org/dev/peps/pep-0318/

http://thecodeship.com/patterns/guide-to-python-function-decorators/

http://simeonfranklin.com/blog/2012/jul/1/python-decorators-in-12-steps/

https://www.codeschool.com/blog/2016/05/12/a-guide-to-python-decorators/