
pyladies

pyladies

4.6

Wielka powtórka!
(i nie tylko)

pyladies

4.6

Kompendium wiedzy z Pythona.

pyladies

Directed by: Jan Śliski

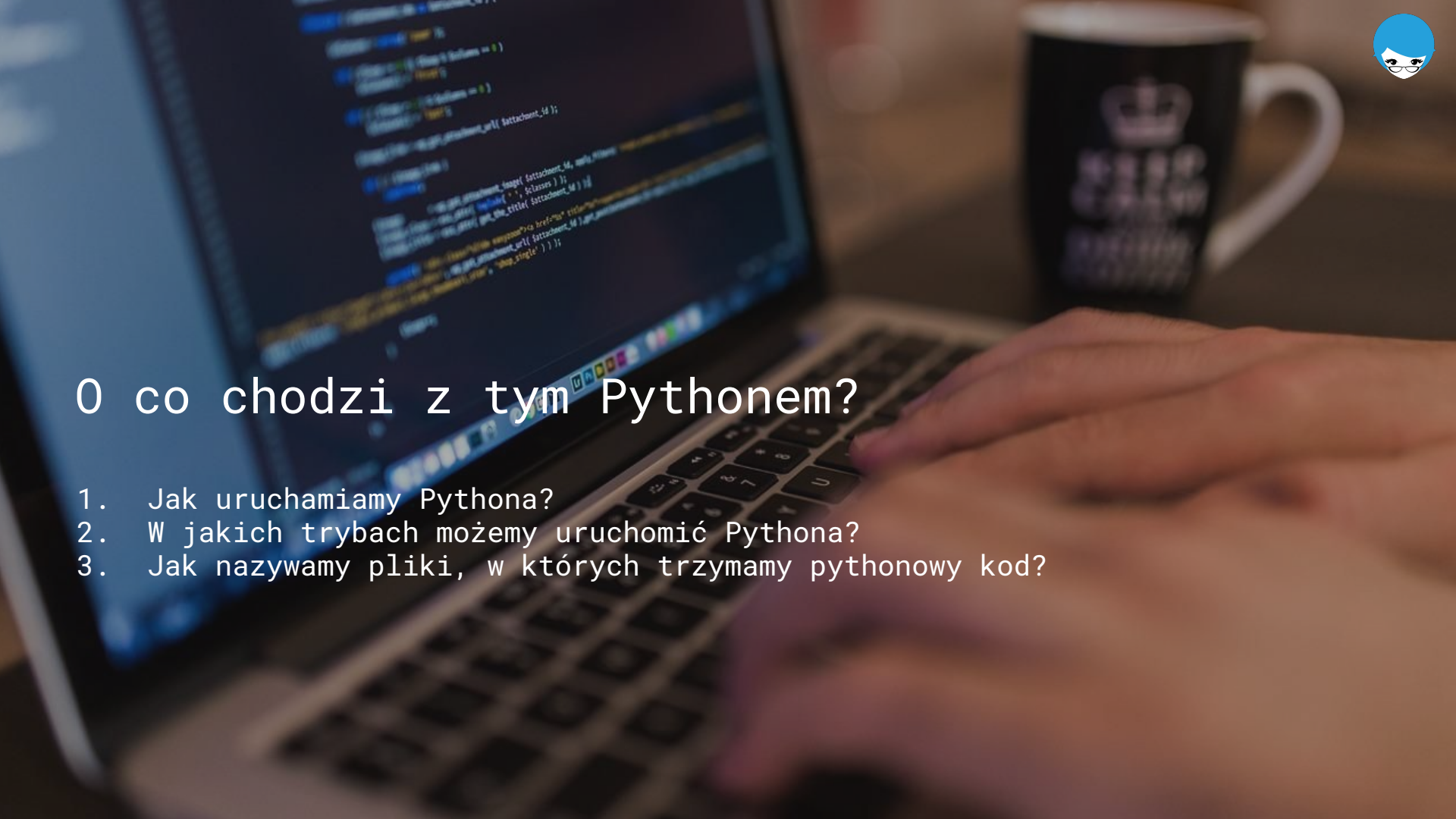
03.01.2018

pyladies

WIFI:
PUT-events-WiFi
Login: user_69211
Hasło: 9my3ci6kaZI

SPIS TREŚCI

1. Python jako program -> 7 - 11
2. Wirtualne środowiska -> 12 - 16
3. Zmienne -> 17 - 22
4. Typy i stałe wbudowane -> 23 - 31
5. Operatory -> 32 - 63
6. Wyrażenie if-else -> 66 - 73
7. Pętle (for, while) -> 74 - 82
8. Listy -> 82 - 92
9. ((NAWIASY)) i wcięcia -> 93 - 97



O co chodzi z tym Pythonem?

1. Jak uruchamiamy Pythona?
2. W jakich trybach możemy uruchomić Pythona?
3. Jak nazywamy pliki, w których trzymamy pythonowy kod?

Ale najpierw… terminal

Jak otworzyć terminal?

Windows:
- wciśnij: Windows + S, lub wybierz lupę (lewy dolny róg)
- wpisz: cmd
- kliknij: Wiersz polecenia/Command Prompt

Linux:
- wciśnij: Ctrl + Alt + T

MacOS:
- wybierz: lupę (Spotlight, prawy górny róg)
- wpisz: terminal
- kliknij: Terminal

Gdzie żyją pytony?

python(.exe) to nazwa programu, który w dużym skrócie odpowiada za wykonanie naszego kodu.

JAK GO ODSZUKAĆ? (wpisz w terminalu)

Windows:

```
where python
# przykładowy wynik:
C:\Users\Janusz\fotki_z_mielna\python.exe
```

Linux + MacOS:

```
which python lub which python3
# przykładowy wynik: /usr/bin/python
```

Czym nakarmić pytona?

Po wpisaniu w terminalu: `python`
uruchomiony zostanie tzw. tryb interaktywny, w którym
wszystko co wpiszemy zostanie natychmiast zinterpretowane.
(Aby wyjść wpisz: `exit()`)

Pythona można jednak nakarmić plikami!
Wystarczy wpisać:

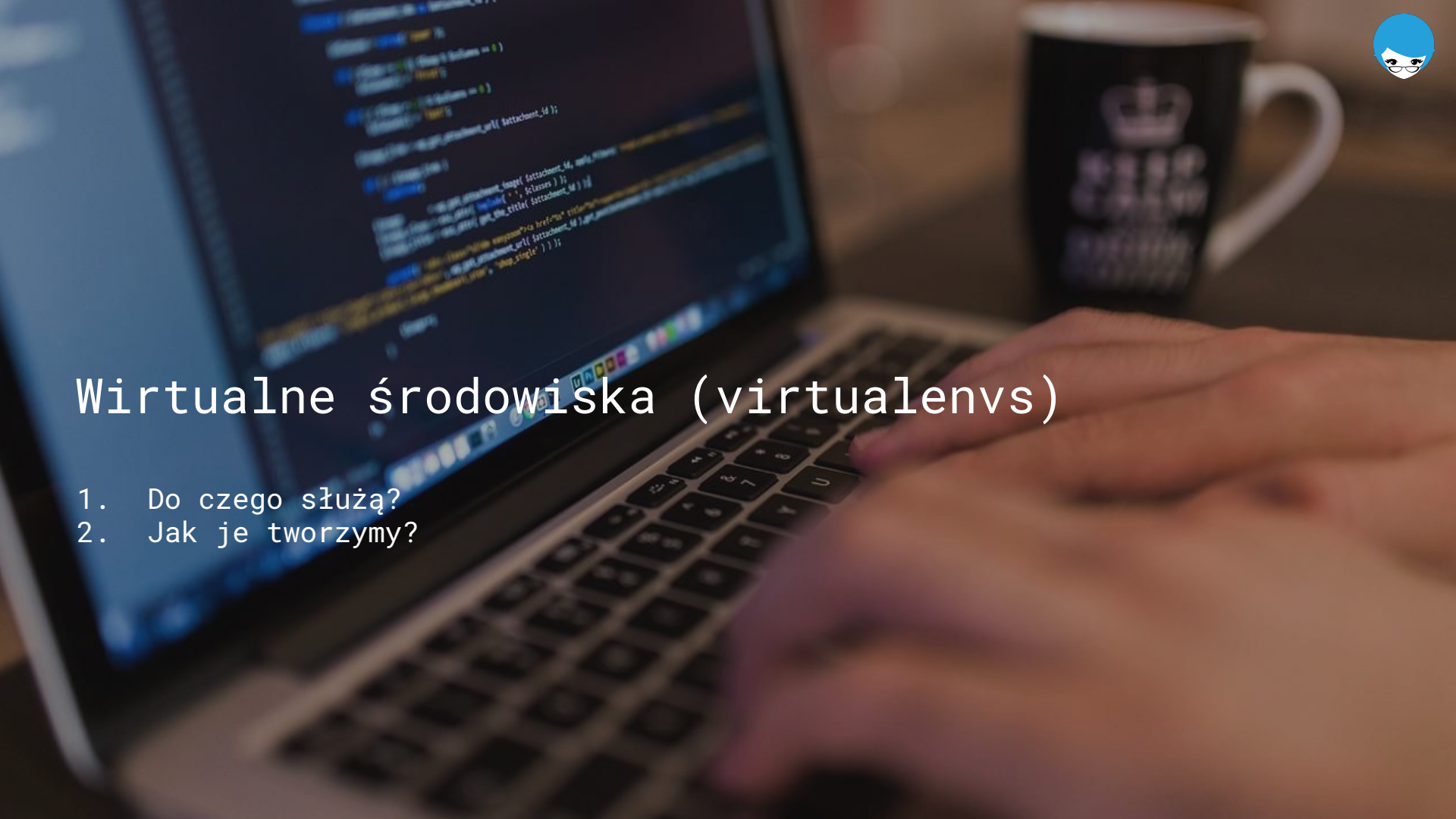
```
python /ścieżka/do/pliku/ze/skryptem.py
```

a python spróbuje wykonać kod który znajduje się we
wskazanym pliku. Jest to tzw. tryb skryptowy.

```
# Windows: python C:\Users\Janusz\Desktop\skrypt.py
# Linux:   python /home/janusz/skrypty/program.py
# Każdy:   python moj_skrypt.py
```

DOBRA PRAKTYKA!

Pythonowi najbardziej smakują pliki z rozszerzeniem **.py**! Chociaż bez problemu poradzi sobie z każdym innym plikiem, jeśli w środku będzie poprawny, pythonowy kod, to dobrą praktyką jest upewnienie się, że nazwa pliku kończy się na ‘.py’.



Wirtualne środowiska (virtualenvs)

1. Do czego służą?
2. Jak je tworzymy?

Z czym to się je?

Wirtualne środowiska, to nic innego jak pliki wykonywalne Pythona (python.exe). Każdy z nich znajduje się w osobnym folderze i posiada kopię wymaganych przez siebie bibliotek (czyli innych plików z kodem niezbędnych do działania projektu).

Wirtualne środowiska zapewniają izolację projektu, i sprawiają, że jego uruchomienie odbywa się w takich samych warunkach w jakich został napisany. Czyli, że będzie działać :)

Eee, okej?

Masz zainstalowane:
- Pythona w wersji 3.6.3
- Bibliotekę NumPy* w wersji 1.14

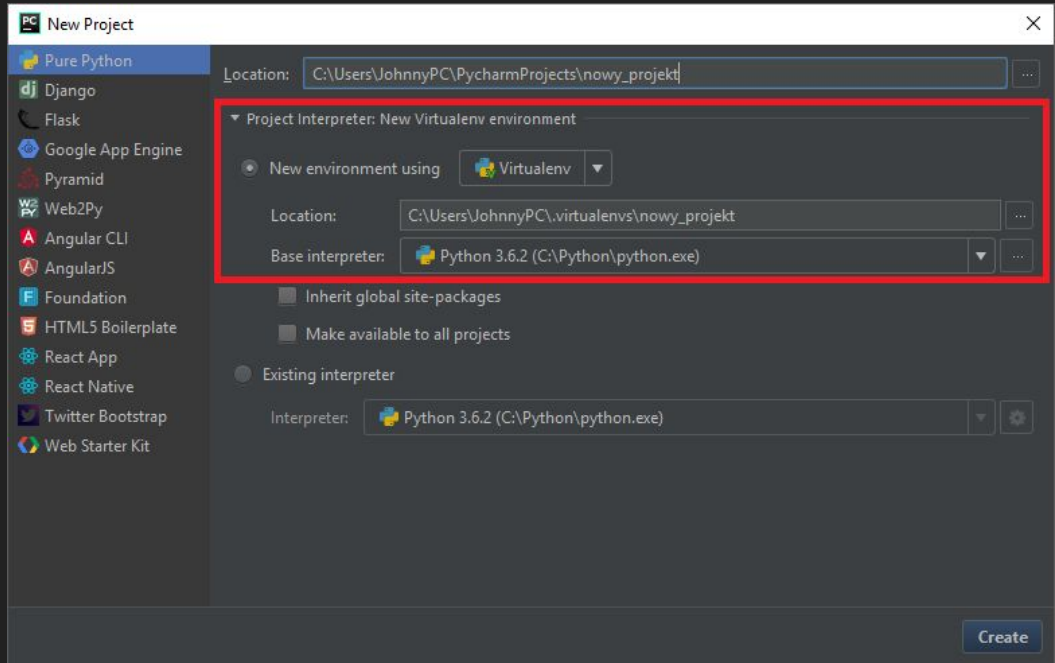
Potrzebujesz w projekcie:
- Pythona 2.6
- NumPy 1.01

Czy w takiej sytuacji musisz reinstalować Pythona? Nie! Tworzysz nowe wirtualne środowisko, które żyje sobie obok Twojego “głównego” Pythona i jest wykorzystywane tylko w projekcie, który tego potrzebuje.

*Biblioteka zawierająca mnóstwo przydatnych rzeczy do obliczeń naukowych.

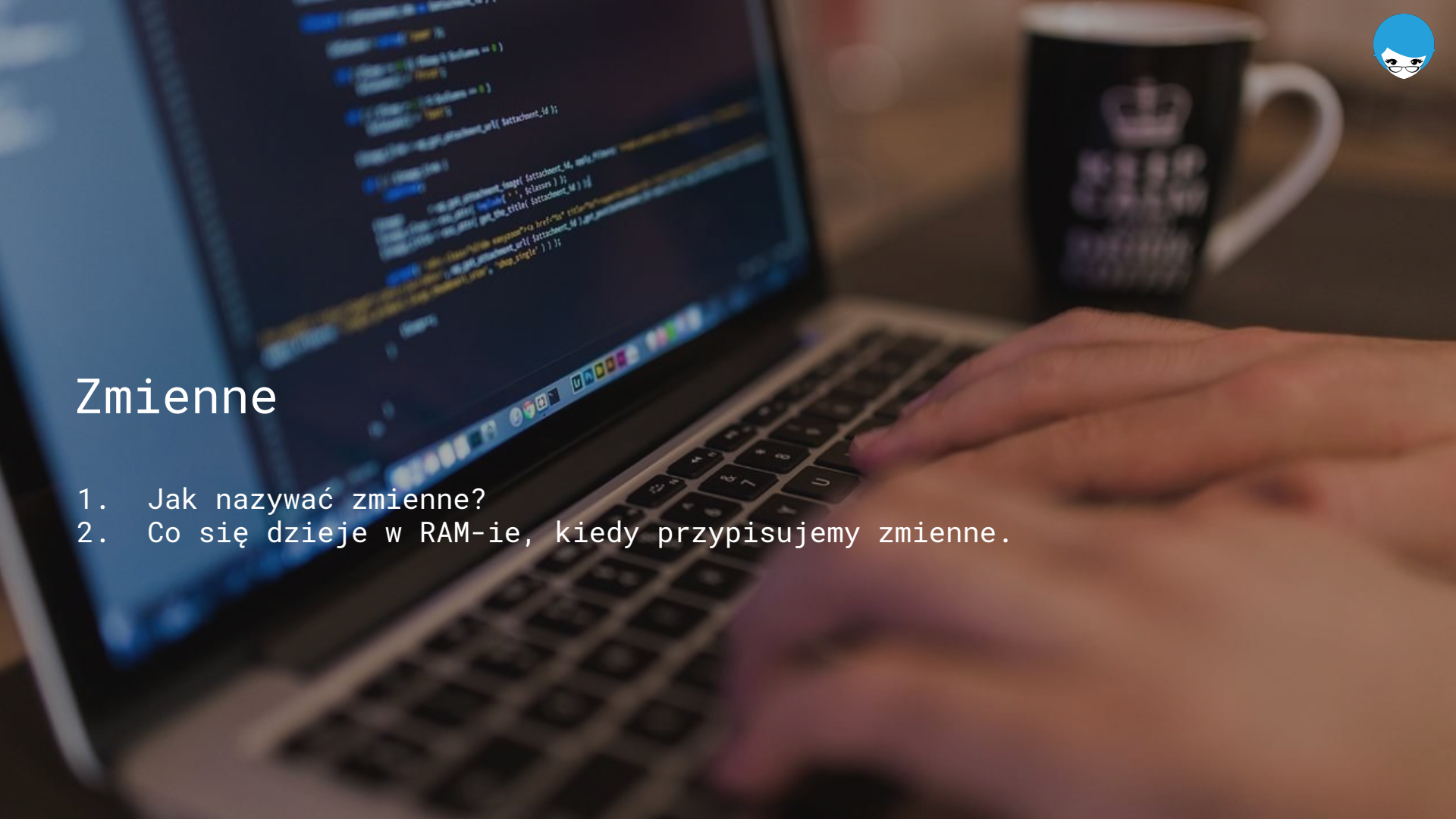
Tworzenie wirtualnego środowiska w PyCharm

PyCharm, dla każdego nowego projektu domyślnie tworzy nowe wirtualne środowisko w wybranym miejscu (Location) kopiując tam plik wykonywalny Pythona we wskazanej wersji (Base interpreter).



Zmiana wirtualnego środowiska dla projektu w PyCharm

1. Wejdź w ustawienia PyCharm (File -> Settings/Ctrl + Alt + S)
2. W pasku wyszukiwania wpisz: Project Interpreter
3. Z rozwijanej list po prawej stronie wybierz jedno z istniejących wirtualnych środowisk.
4. Albo stwórz nowe:
   kliknij: kółko zębate (prawa strona listy).
   wybierz: Add Local
   wskaż: lokalizację w której nowe środowisko ma zostać stworzone (Location).
   wskaż: wersję Pythona, na której środowisko ma bazować (Base interpreter).



Zmienne

1. Jak nazywać zmienne?
2. Co się dzieje w RAM-ie, kiedy przypisujemy zmienne.

Jak nazywać zmienne? - odkryj 3 proste triki!

1. Tak, żeby było wiadomo co w zmiennej się znajduje!

```
# Źle nazwana zmienna:
a = ((4/23) + math.pi**4) / math.e ** (1/7))
# Dobrze nazwana zmienna:
average_bat_lifespan = ((4/23) + math.pi**4) / math.e ** (1/7))
```

Nadal nie wiemy co się dzieje w równaniu po prawej, ale wnioskując po nazwie zmiennej - obliczana jest średnia długość życia nietoperza.

2. Używając snake_case!

```
some_variable_name # super! 100% pythona w pythonie
someVariableName   # tzw. camel case, w pythonie nie jest używany.
sOmEvArIaBlEnAmE   # nie, nie, nie, nie, nie! NIE!
```

Jak nazywać zmienne? - ciąg dalszy

3. Używając języka angielskiego.

   Chociaż temat ten budzi kontrowersje, to faktem jest, że język angielski zdominował środowisko programistyczne.
   Dokumentacja, fora, publikacje naukowe, czy wreszcie sam kod pisany przez innych programistów do użytku publicznego - wszystko to jest przede wszystkim tworzone w wyżej wymienionym języku.

   Kod pisany w jednym języku także czyta się prościej, bo nie musimy co chwila “przełączać się” w głowie pomiędzy polskim a angielskim.

   Korzystanie z większych zasobów wiedzy pozwoli wam szybciej znaleźć rozwiązanie waszych problemów! Przecież nie chodzi o to, żeby się niepotrzebnie męczyć “dla zasady” ;)

Jak działa pamięć RAM?

Co się dzieje w RAM-ie, kiedy przypiszemy wartość do zmiennej?

Usiądźcie wygodnie, weźcie kubek ulubionego napoju i przeczytajcie!

PS Jak się domyślacie, jest to historia uproszczona. Numery pokojów są zupełnie przypadkowe i przy każdym uruchomieniu programu będą się różnić.

Dawno temu w RAM-ie.

Wyobraź sobie, że pamięć RAM to bardzo, bardzo, bardzo długi korytarz hotelowy, który tylko z jednej strony ma rosnąco numerowane drzwi, a drzwi te prowadzą do jednakowych pokoi.

Kiedy uruchamiamy nasz program pythonowy, System Operacyjny (jest szefem tego hotelu, więc ostrożnie!), przydziela nam część pokoi do wykorzystania przez nasz program, np: pokoje o numerach od 200 do 550 oraz pokoje o numerach od 1245 do 1733. Wspominałem już, że to długi korytarz?

W momencie kiedy do zmiennej przypiszemy jakąś wartość, np:

```
my_favourite_number = 888
```

w hotelu zaczynają się dziać rzeczy, w które jeszcze niedawno nikt by nie uwierzył…

Ciąg dalszy nastąpi.

Dawno temu w RAM-ie 2.

Na czym skończyliśmy? Na przypisaniu do zmiennej wartości, o tak:

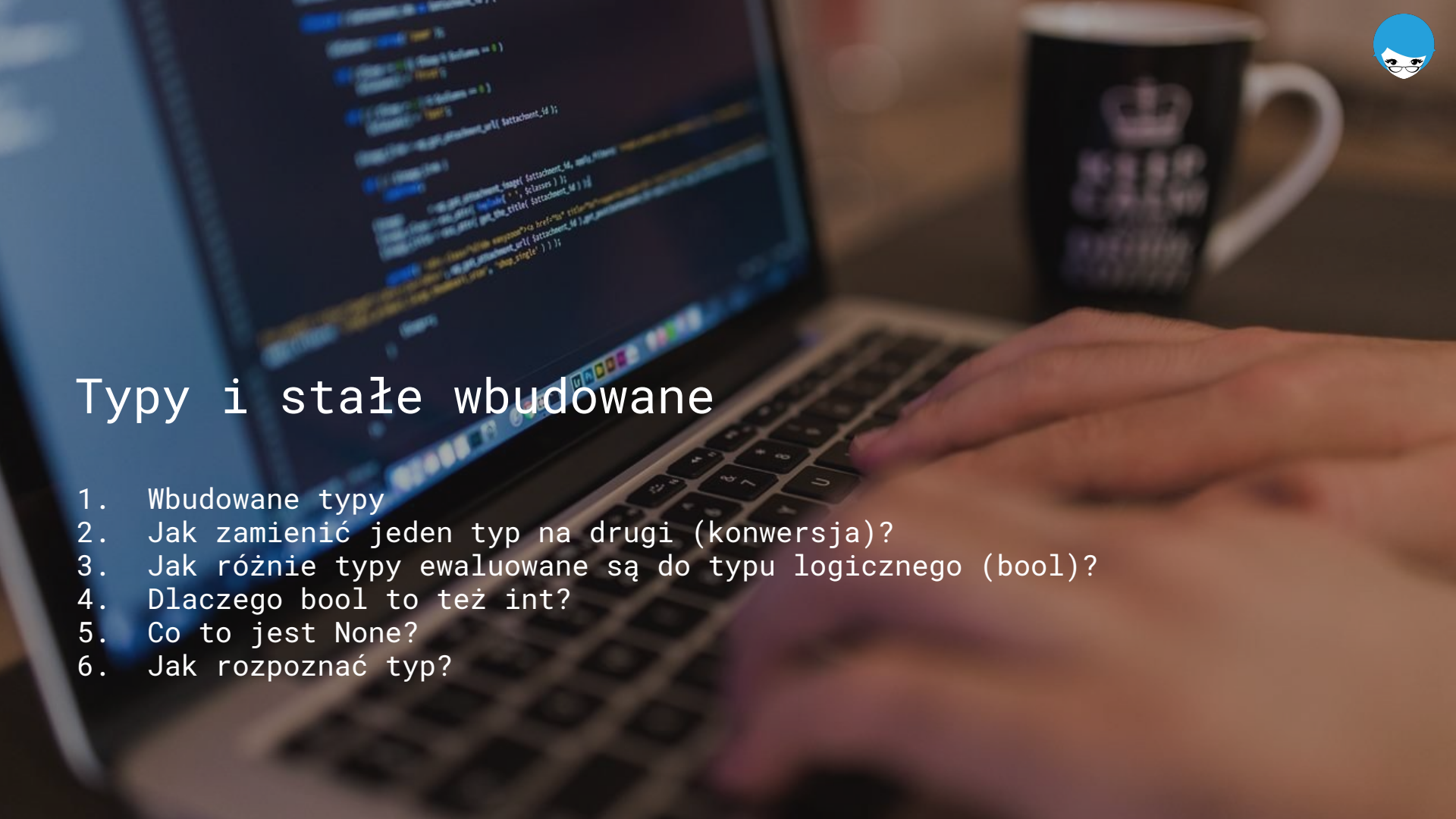
```
my_favourite_number = 8888
```

W czasie takiej operacji, z puli dostępnych pokoi wybierany jest jeden, wolny pokój, do którego wprowadza się wartość. Załóżmy, że wybrany został pokój numer 1564:

```
Widok z góry:   [      ][ 8888 ][      ][      ]
Numer pokoju:     1553    1564    1565    1566
```

Zmienna `my_favourite_number` nie przetrzymuje “w sobie” wartości `8888`, a jedynie numer pokoju (tzw referencję), w którym ta zmienna zagościła.
Aby sprawdzić numer pokoju, w którym znajduje się wartość przypisana do naszej zmiennej, możemy użyć funkcji `id`:

```
id(my_favourite_number) # 1564
```



Typy i stałe wbudowane

1. Wbudowane typy
2. Jak zamienić jeden typ na drugi (konwersja)?
3. Jak różnie typy ewaluowane są do typu logicznego (bool)?
4. Dlaczego bool to też int?
5. Co to jest None?
6. Jak rozpoznać typ?

Co za typ!

Python posiada kilkanaście wbudowanych typów, spośród których poznaliśmy:

```
int (Integer) - liczba całkowita
# -100, 0, 24

float (Floating Point) - liczba rzeczywista (zmiennoprzecinkowa)
# -15.4, 0.0, 14.1

bool (Boolean) - wartość logiczna
# True, False

str (String) - łańcuch znaków
# ‘Ala ma kota’, “Ala ma kota”, “””Ala
                                  ma
                                  kota”””
```

Co za typ!

Kolejne z wbudowanych typów należą do tzw. typów sekwencyjnych, spośród nich poznaliśmy:

listy - do definiowania list używamy nawiasów kwadratowych - []

```
# [1, ‘kotek’, [‘Ala’, ‘ma’, ‘kota’]], []
```

range - używana do generowania sekwencji liczb.

```
# range(5), range(1, 20), range(1, 20, 2)
```

Konwersja typów

Aby przekonwertować jeden typ na drugi, wystarczy wziąć nazwę typu docelowego i w nawiasach okrągłych podać wartość którą chcemy skonwertować, np:

```
str na int:   int('42')      # 42
str na float: float('1.4')   # 1.4

int na float: float(13)      # 13.0
float na int: int(567.7)     # 567 (tzw. podłoga)

int na str:  str(42)         # '42'
list na str: str([1, 2, 3]) # '[1, 2, 3]'
bool na str: str(True)       # 'True'
```

Wszystko jest fałszem (albo prawdą)

Wszystkie pozostałe typy mogą zostać przekonwertowane do wartości prawda/fałsz.

Dla typów liczbowych każda wartość inna niż 0 jest interpretowana jako prawda!

```
bool(0)   # False        bool(0.1)  # True
bool(-1)  # True         bool(100)  # True
```

Dla łańcuchów znaków (str), każdy niepusty łańcuch jest prawdą!

```
bool('')  # False        bool('a')   # True
bool(' ') # True         bool('kot') # True
```

Dla list, każda niepusta lista jest prawdą!

```
bool([])  # False        bool([1])   # True
```

Konwersja typów - tak się nie da!

Jeśli konwersja się nie uda, Python zwróci nam odpowiedni komunikat:

```
int('kotek')
ValueError: invalid literal for int() with base 10: 'kotek'

x = range([1, 2, 3]) # Nie rób tak, to się nigdy nie uda :(
TypeError: 'list' object cannot be interpreted as an integer
```

… i w trybie skryptowym nasz program przestanie działać :(

Na szczęście można temu zapobiec!
Ale o tym powiemy sobie przy innej okazji ;)

Prawda to, czy fałsz? Bool

Typ bool jest typem logicznym reprezentowany jest przez stałe True oraz False, oznaczające logiczną prawdę lub fałsz.

W Pythonie bool jest także takim ‘podzbiorem typu int’ składającym się z dwóch elementów: {0, 1} gdzie stała True odpowiada 1, a False - 0

Ta powyższa właściwość pozwala używać stałych logicznych w operacjach arytmetycznych. Chociaż nie powinno się tego robić, bo jest to mało czytelne, tutaj przykład:

```
player_speed = 75                # Prędkość gracza
should_keep_moving = False       # True - gracz może dalej się
                                 przemieszczać po mapie z daną
                                 prędkością; False - zatrzymaj gracza
                          0
player_speed *= should_keep_moving
print(player_speed) # 0
```

None - nic tu nie ma!

None, podobnie jak True oraz False jest stałą wbudowaną, a używa się jej, gdy chce się zasygnalizować ‘brak wartości’, np:

```
name = None

# tutaj wyobraźcie sobie kod!
# mnóstwo operacji, które
# mogą przypisać do zmiennej
# name wartość (ale nie muszą).

if name is not None:
    print(f‘Hello, {name}’)
```

None ewaluowane jest do logicznego fałszu:

```
    bool(None) # False
```

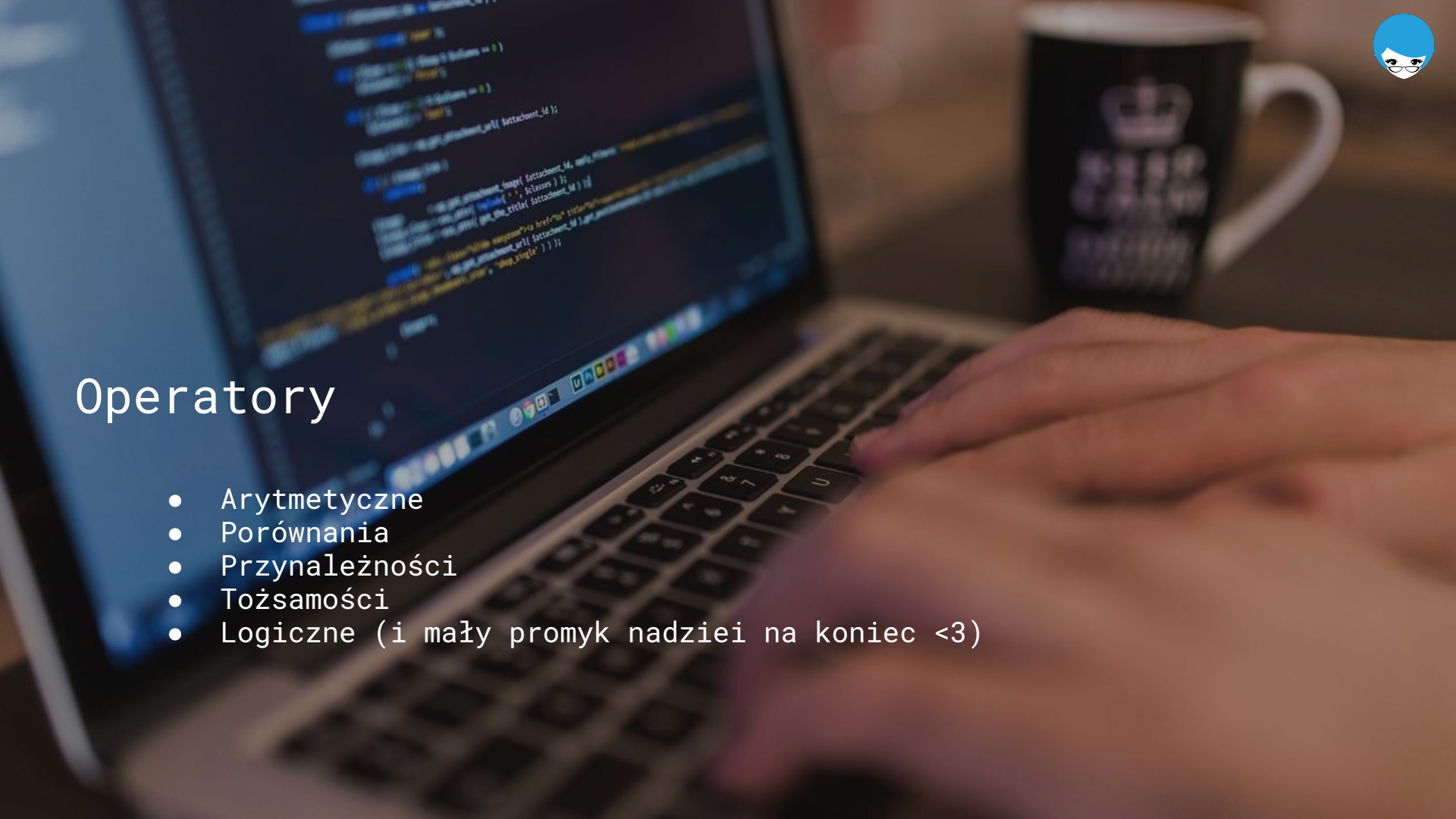
Proszę się przedstawić!

Jeśli chcemy sprawdzić, czy dana zmienna jest określonego typu, możemy to zrobić za pomocą isinstance w następujący:

```
x = 14
y = 0.5
z = False

isinstance(x, int)   # True
isinstance(y, float) # True
isinstance(x, str)   # False
isinstance(y, bool)  # False

isinstance(z, int)   # True, typ logiczny to ‘podzbiór’ typu int
```



Operatory

- Arytmetyczne
- Porównania
- Przynależności
- Tożsamości
- Logiczne (i mały promyk nadziei na koniec <3)

Podstawowe operatory - przypisanie

Symbol: =

Znaczenie: do zmiennej po lewej stronie symbolu przypisz wartość wyrażenia po prawej stronie.

Przykłady:

```
x = 4

y = x * 2

a_list = [1, ‘Ala’, [‘spam’, ‘eggs’]]

another_list = a_list

is_less = x < 10

savings = None
```

Podstawowe operatory - dodawanie

Symbol: +

```
x = 4
y = 6
x + y # 10 (int)
```

```
x = 4
y = 2.5
x + y   # 6.5 (float)
```

```
x = 4
y = True
x + y    # 5 (int)
```

```
x = 4.0
y = True
x + y    # 5.0 (float)
```

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list1 + list2 # [1, 2, 3, 4, 5, 6]
list2 + list1 # [4, 5, 6, 1, 2, 3]
#Tutaj kolejność ma znaczenie ;)
```

Podstawowe operatory - dodawanie

```
x = 4   # int
y = '4' # str
x + y # TypeError: unsupported operand type(s) for +: 'int' and 'str'
y + x # TypeError: must be str, not int
```

Jeśli oczekujemy liczby w łańcuchu znaków:

```
    x + int(y) # 8
```

Jeśli chcemy połączyć dwa łańcuchy:

```
    y + str(x) # ‘44’
```

Podstawowe operatory - odejmowanie

Symbol: -

```
x = 4
y = 6
x - y # -2 (int)
```

```
x = 4
y = 2.5
x - y   # 1.5 (float)
```

```
x = 4
y = True
x - y # 3 (int)
```

```
x = 4.0
y = True
x - y # 3.0 (float)
```

```
# I tyle :(
# Odejmowanie jest nudne!
```

Podstawowe operatory - mnożenie

Symbol: *

```
x = 4 (int)
y = 6 (int)
x * y # 24 (int)
```

```
x = 4 (int)
y = 2.3 (float)
x * y # 9.2 (float)
```

```
x = 2
y = 'Ala'
x * y # 'AlaAla'
```

```
x = False
y = 'Ala'
x * y # ''
```

```
x = 2
y = [1, 2, 3]
x * y # [1, 2, 3, 1, 2, 3]
```

```
x = False
y = [1, 2, 3]
x * y # []
```

Podstawowe operatory - dzielenie

Symbol: /

```
x = 4 (int)
y = 8 (int)
x / y # 0.5 (float)
```

```
x = 4 (int)
y = 2.3 (float)
x / y # 1.7391304... (float)
```

```
x = 2
y = 'Ala'
x / y # TypeError: unsupported operand type(s) for /: 'int' and 'str'
y / x # TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

```
x = 2
y = [1, 2, 3]
x / y # TypeError: unsupported operand type(s) for /: 'int' and 'list'
y / x # TypeError: unsupported operand type(s) for /: 'list' and 'int'
```

:(

Inne podstawowe operatory arytmetyczne

```
Symbol: %
Znaczenie: modulo (reszta z dzielenia)
Przykłady: 16 % 3 # 1 | 12 % 4 # 0 | 23.3 % 5.7 # 0.5


Symbol: //
Znaczenie: podłoga z dzielenia (wynik: float)
Przykłady: 16 // 3.5 # 4.0 | 12.7 // 5 # 2.0


Symbol: **
Znaczenie: potęgowanie (wynik: int lub float)
Przykłady: 2 ** 4 # 16 | 4.5 ** 2 # 20.25 | 4 ** 2.5 # 32.0
           4 ** 0.5 # 2.0
```

Podstawowe operatory arytmetyczne - na skróty

Niektóre operacje można zapisać w formie skróconej, bo jest szybciej i równie czytelnie ;)

```
x = 4 # wartość początkowa dla każdego przykładu

x = x + 1       ->  x += 1      # 5
x = x - 1       ->  x -= 1      # 3
x = x * 2       ->  x *= 2      # 8
x = x / 2       ->  x /= 2      # 2
x = x // 2.5    ->  x //= 2.5   # 1.0
x = x ** 2      ->  x **= 2     # 16
x = x % 3       ->  x %= 3      # 1
```

Zapis skrótowy jest generalnie preferowany!

Operatory porównania

Operatory porównania, jak sama nazwa wskazuje - służą do porównywania dwóch **wartości** (czyli tego co jest w pokoju), a wynikiem ich działania jest stała logiczna True lub False.

```
x = 4
y = 3

Symbol: ==
Znaczenie: równy
Przykład: x == y # False


Symbol: !=
Znaczenie: różny
Przykład: x != y # True
```

Operatory porównania - ciąg dalszy

```
x = 4
y = 3
```

```
Symbol: <, >
Znaczenie: mniejszy od, większy od
Przykłady: x < y # False
           x > y # True
```

```
Symbol: <=, >=
Znaczenie: mniejszy lub równy, większy lub równy
Przykłady: x <= y # False
           x >= y # True
```

Ciekawy przypadek pana stringa.

Operatorów porównania możemy używać nie tylko na liczbach, ale także na innych typach danych. W przypadku łańcuchów znaków może to posłużyć np. do ustalenia porządku alfabetycznego.

```
x = 'apple'
y = 'banana'

x < y # True
```

Z czego to wynika? Stringi składają się ze znaków, takich jak `'A'`, `'x'`, `'9'`, `'-'`. Każdy znak ma przypisany swój kod w tabeli znaków UNICODE. Na przykład litery od `A` do `Z` mają przyporządkowane kody od `65` do `90`, a litery od `a` do `z` - odpowiednio od `97` do `122`. Kody te są liczbami całkowitymi (int).

Ciekawy przypadek pana stringa.

Kiedy więc porównujemy ze sobą dwa łańcuchy znaków, Python dla każdego kolejnego znaku z porównywanych stringów bierze jego reprezentację numeryczną (kod) i porównuje je ze sobą, tak długo, aż znajdzie nierówność.

'aardvak' vs 'aaron':

```
Indeks:  0  |  1  |  2  |  3
Znak:   'a' | 'a' | 'r' | 'd' S
Kod:     97 |  97 | 114 | 100 T
Status:  == |  == |  == |  <  O
Kod      97 |  97 | 114 | 111 P
Znak:   'a' | 'a' | 'r' | 'o'

'aardvak' < 'aaron' # True
   'd'    <    'o'  # True
```

Ciekawy przypadek pana stringa.

Aby sprawdzić kod dla danego znaku, można użyć funkcji ord

```
ord('Q') # 81
```

Aby sprawdzić, jaka liczba koduje dany znak, można użyć funkcji chr

```
chr(115) # 's'
```

Ile jest takich znaków? Bardzo dużo. Kilkając tutaj możecie zobaczyć jak wygląda pierwsze 256 kodów.

Operatory porównania - listy

Jeśli dwie listy posiadają takie same elementy w takiej samej kolejności operator równości zwróci stałą True.

```
list1 = [1, 2, 3]
list2 = [1, 2, 3]

list1 == list2 # True
```

```
list1 = [1, 2, 3]
list2 = [1, 3, 2]

list1 == list2 # False
```

Użycie na liście operatora mniejszości (<), zadziała podobnie jak w przypadku łańcucha tekstowego, porównywane będą wartości kolejnych elementów list.

```
list1 = [1, 2, 3]
list2 = [1, 3, 2]

list1 < list2 # True
```

Operator przynależności - in

Aby sprawdzić, czy sekwencja zawiera w sobie daną wartość możemy użyć operatora przynależności in, lub not in aby sprawdzić warunek przeciwny. Zwracane jest True lub False:

```
a_list = [1, 2, 3, 4]
1 in a_list # True              1 not in a_list # False
5 in a_list # False             5 not in a_list # True


sentence = 'Ala ma kota'
'kot' in sentence   # True      'kot' not in sentence  # False
'Jacek' in sentence # False     'koty' not in sentence # True


sentences = ['Ala ma kota', 'Mam 5 lat']
'Ala' in sentences          # False
'Ala ma kota' in sentences # True
```

Operator tożsamości - is

Operator tożsamości `is` zwraca `True` jeśli dwie porównywane zmienne wskazują na ten sam obiekt w pamięci (czyli na ten sam pokój na naszym korytarzu). Jego odwrotnością jest `is not`:

```
x = 320
y = 320
x is y # False
```

```
x = 320
y = x
x is y # True
```

```
lista1 = [1, 2, 3]
lista2 = [1, 2, 3]
lista1 is not lista2 # True
```

```
lista1 = [1, 2, 3]
lista2 = lista1
lista1 is not lista2 # False
```

Kiedy używać is, a kiedy ==

Pamiętacie jeszcze ‘Dawno temu w RAM-ie’?

Najprościej mówiąc, is użyjemy, jeśli będziemy chcieli sprawdzić, czy nasze dwie zmienne wskazują na te same pokoje, a == jeśli będziemy chcieli sprawdzić czy to co znajduje się w pokojach, ma taką samą wartość.

True, False oraz None są tak zwanymi stałymi wbudowanymi. Oznacza, to, że kiedy program dostanie od kierownika hotelu pokoje do przydzielenia, jedną z pierwszych rzeczy jaką zrobi, będzie umieszczenie w 3 osobnych pokojach wartości dla każdej z tych stałych.

Każda z nich będzie więc przez cały czas trwania programu wskazywać zawsze jeden, ten sam pokój! Dlatego porównując możemy użyć is:

```
x = None                                    y = 1 < 2
x == None    <- Tak nie porównuj! ->        y == True
x is None    <-   Tak porównuj!   ->        y is True
```

Operatory logiczne - and, or oraz not

Operatory te pozwalają nam łączyć prostsze wyrażenia logiczne w bardziej skomplikowane, np. w blokach if:

```
number_of_wheels = 4
has_doors = True
if has_doors and number_of_wheels == 4:
    print('To samochód! Chyba...')

barks = True
woofs = True
if barks or woofs:
    print('To musi być pies!')

if not (barks or woofs):
    print('To może być kot!')
```

Co się dzieje pod maską?

Pierwszy krok w dół - poznajcie wyrażenie

Pod maską dzieje się bardzo dużo! Na warsztat weźmy tego if'a:

```
    if has_doors and number_of_wheels == 4:
       \                                  /
        \                                /
         \     to jest: wyrażenie       /
```

Semantycznie (znaczeniowo):

```
    if has_doors and number_of_wheels == 4:
```

równy jest takiemu zapisowi:

```
    if bool(has_doors and number_of_wheels == 4) is True:
```

Czyli sprawdzamy, czy nasze wyrażenie może zostać zinterpretowane jako prawda w kontekście logicznym i jeśli tak, wykonujemy wcięty blok kodu:

```
    if True:
        # wykonuje wcięty blok kodu
```

Drugi krok w dół - poznajcie wyrażenie

Jeśli nasze wyrażenie to jedna zmienna lub wartość - nie ma sprawy! Zostanie bez problemu zinterpretowana do typu logicznego.

```
if 2:   # 2 interpretowane jest jako True
```

Jeśli nasze wyrażenie wygląda tak: `has_doors and number_of_wheels == 4` musi się tutaj zadziać trochę magii.

Zwróćcie uwagę, że operator `and` ma coś po swojej lewej i prawej stronie! Wyrażenie po prawej (`number_of_wheels == 4`) jest równe `True` (bo taką wartość, czyli `4`, przypisaliśmy do tej zmiennej 2 slajdy temu).

Mamy więc taką sytuację: `has_doors and True`

Gotowi na magię?

Trzeci krok w dół - poznajcie AND

Operator and działa od lewej do prawej! A co do dokładnie w tym chodzi?

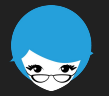
```
wyrażenie1 and wyrażenie2

# Kroki algorytmu:
Weź wyrażenie po lewej stronie operatora and (czyli wyrażenie1)
i zinterpretuj je w kontekście logicznym.

    -> Jeśli otrzymana wartość logiczna jest prawdą (True)
        -> weź wyrażenie po prawej stronie operatora and (wyrażenie2)
           i logicznie je zinterpretuj.
            -> jeśli otrzymana wartość logiczna jest prawdą (True)
                -> zwróć wartość tego wyrażenia
            -> w przeciwnym wypadku                    > WYKONA SIĘ TO SAMO?**
                -> zwróć wartość tego wyrażenia

    -> W przeciwnym wypadku:
        -> zwróć wartość tego wyrażenia
```

AND - czarna magia



**Tak, wykona się to samo! A dlaczego?

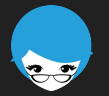
W przypadku operatora `and`:
    jeżeli wartość po jego lewej stronie
    zostanie zinterpretowana jako `True`:
        ZAWSZE zostanie zwrócona wartość po prawej stronie.

Bardziej sensownie działanie `and` można opisać w ten sposób:
1. Jeśli wszystkie wyrażenia są prawdą w kontekście logicznym, zwróć wartość ostatniego wyrażenia.
2. Jeśli któreś z wyrażeń jest fałszem, zwróć wartość pierwszego fałszywego wyrażenia.

A jeszcze bardziej sensownie:
Dla: `x and y`
Jeśli `x` jest fałszem, zwróć jego wartość, w przeciwnym wypadku zwróć wartość `y`.

Zaczynamy od lewej strony!

AND - Czas na przykład!



has_doors and True (w takim stanie zostawiliśmy slajd 52)

has_doors ma wartość True (slajd 50), czyli mamy:

True and True

Najpierw lewa strona: True # True jest logicznie prawdą!
Skoro lewa strona jest prawdą w kontekście logicznym, możemy sprawdzić drugą stronę: True # Też jest prawdą

W takim przypadku (wszystkie wyrażenia są prawdą) zwracamy wartość ostatniego, czyli True.

Cofając się o 5 slajdów, nasze:

```
    if has_doors and number_of_wheels == 4:
```

Zamienia się w:

```
    if True: # Warunek jest spełniony! Hurra!
```

AND - czarna magia

A co z takim przykładem?

```
    has_doors and number_of_wheels
       True              4          <- wartości powyższych zmiennych
```

Sprawdźmy!

```
True and 4
```

1. True to logiczna prawda więc przechodzimy na drugą stronę operatora
2. 4 zostanie zinterpretowane jako True, więc co zwracamy?

WARTOŚĆ, czyli 4

W ten sposób nasz blok:

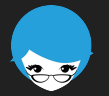
```
    if has_doors and number_of_wheels:
```

Wygląda tak:

```
    if 4: # 4 ponownie zostanie zinterpretowane jako True

    if True: # Warunek spełniony!
```

Czas na OR!

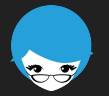


Operator or działa od lewej do prawej tak samo jak and! Bez rozpisywania kolejnych kroków wygląda to tak:

1. Jeśli któreś wyrażenie jest prawdą or zwraca jego wartość natychmiast.
2. Jeśli wszystkie wyrażenia są fałszem or zwraca wartość ostatniego wyrażenia.

A w wersji najkrótszej:
Dla: x or y
Jeśli x jest prawdą w kontekście logicznym, zwróć jego wartość, w przeciwnym razie zwróć wartość y.

Czas na OR!



Przykład (sprzed 8 slajdów):

```
if barks or woofs:
   True     True

True or True
```

Czy True jest logiczną prawdą? Tak, zwracamy!

Mamy więc:

```
    if True:
        # ten kod się wykona!
```

Czas na OR!

Inny przykład:

```
if False or [‘a’, ‘b’, ‘c’]:
```

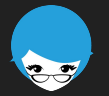
```
False or [‘a’, ‘b’, ‘c’]
```

`False` nie jest logiczną prawdą, przechodzimy na drugą stronę operatora
`[‘a’, ‘b’, ‘c’]` - niepusta lista jest interpretowana jak logiczna prawda,
zwrócona zostanie lista!

Mamy więc:

```
    if [‘a’, ‘b’, ‘c’]:
        # [‘a’, ‘b’, ‘c’] is True, więc warunek jest spełniony!
```

Schody! - przykład z dwoma operatorami!



```
variable = 'Potter' or [] and ['a', 'b', 'c']
```

Po pierwsze - nie panikujemy!

```
'Potter' or [] and ['a', 'b', 'c']
```

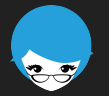
Tak samo jak w matematyce pierwszeństwo ma mnożenie przed dodawaniem, tak w Pythonie pierwszeństwo ma operator and. Czyli najpierw rozpatrujemy:

```
[] and ['a', 'b', 'c']

Interpretujemy (logicznie) []       # False, nie sprawdzamy dalej,
                                      zwracamy []!

'Potter' or [] and ['a', 'b', 'c'] # [] or ['a', 'b', 'c'] zwraca []. []
           \                     / # staje się teraz prawą wartością
'Potter' or  \       []       /  # kolejnego operatora:
```

Schodów ciąg dalszy!



Zostaliśmy z:

```
    'Potter' or []
```

Interpretujemy logicznie lewą wartość:

```
'Potter'          # True, zwracamy!
```

A nasze początkowe:

```
variable = 'Potter' or [] and ['a', 'b', 'c']
```

Zmieniło się w:

```
variable = 'Potter'
```

A to już chyba wygląda znajomo? ;)

Magia and magia!

```
'' and True and [1, 2, 3]   # ''
'kot' and True or [1, 2, 3] # True
'' or True or []            # True
'' or True and []           # []
[1] or [] and ' '           # [1]
True and False and ()       # False
```

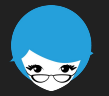
Przykład użycia w kodzie:

```
DOMYSLNY_PROMIEN_KM = 25
wiadomosc = ('Podaj, w jakim promieniu (km) chcesz szukac dopasowań '
             '(wpisz 0 aby wybrać domyślną wartość - 25 km):')
podany_zasieg = abs(int(input(wiadomosc)))

zasieg = podany_zasieg or DOMYSLNY_PROMIEN_KM # Jeśli użytkownik wpisze 0,
do zmiennej zasieg zostanie przypisana wartość 25.


BONUS - co robi: podany_zasieg = abs(int(input(wiadomosc))) ?
Zaczynamy od najbardziej zagnieżdżonego wyrażenia:
input(wiadomosc)      # wyświetla użytkownikowi wartość zmiennej wiadomosc
                        i zwraca wprowadzoną przez niego wartość jako str.
int(input(wiadomosc)) # przyjmuje wartosc (w tym przypadku str) i próbuje
                        dokonać konwersji na typ int. Zwraca wartość
                        liczbową - wynik konwersji - jeśli się to uda.
abs(int(input(wiadomosc))) # przyjmuje liczbę i zwraca jej wartość
                             absolutną.
```

Czy to na serio musi być takie skomplikowane?



Nie! Jeśli mieliście logikę na studiach/w liceum to takie wyrażenie:

```
    if x and y:
```

możecie rozumieć jako:

“Jeśli x ORAZ y są logiczną prawdą”

A takie:

```
    if x or y:
```

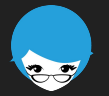
jako:

“Jeśli x jest prawdą LUB y jest prawdą”

Po czym możecie po prostu zinterpretować wartość logiczne x i y, bez bawienia się w te wszystkie kroki.

TO PO CO W OGÓLE MI TO WSZYSTKOOOO???!!!!111oneone

Czy to na serio musi być takie skomplikowane?



Odp:
Bo w Pythonie dużo dzieje się pod maską! Bardzo dużo i co więcej, działa to inaczej niż w innych językach programowania, z którymi zapewne kiedyś się zetkniecie.
W nich, w przeciwieństwie do Pythona w if-ach i elsach nie można używać innych typów niż logiczne (np list, stringów), a zwracana jest wartość logiczna (True/False w danym języku), a nie wartość zmiennej (czyli na przykład: [‘ODJANIEPAWLAĆ’, 2,’xd’], True, lub ‘MAKOWIEC’).

NO DOBRA, ALE TAKIE COŚ:

```
port = 403
protocol = port == 403 and ‘https//:’ or ‘http://’
```

JEST NADAL MAŁO CZYTELNE!!!

Odp. No jest xd Tym niemniej jest to jak najbardziej poprawny zapis, z którym możecie się spotkać, więc warto wiedzieć co się tu dzieje. Jeśli chcecie się dowiedzieć jak takie wyrażenie zapisać w sposób dużo bardziej elegancki (i równie krótki), rzućcie okiem na slajdy 71-72!

IF - ELSE

1. Do czego służą bloki if-else?
2. Jak je definiujemy?
3. Bloki if-else w linii.

IF - ELSE

`if`, `elif`, `else` używamy w celu kontroli przepływu sterowania w programie - tzn. co ma się wykonać, jeśli jakiś warunek jest spełniony lub nie.

```
if warunek:
    # jeśli warunek jest prawdą
    # w kontekście logicznym, to
    # wykonaj ten, KONIECZNIE WCIĘTY kod
elif inny_warunek:
    # jeśli inny_warunek jest
    # prawdą, to wykonaj ten kod
else:
    # jeśli żaden z powyższych warunków
    # nie został spełniony, wykonaj ten kod
```

Warunki są sprawdzane po kolei od góry, jeśli któryś z nich jest prawdą, to te, które pozostały nie zostaną sprawdzone.

IF - ELSE

Obowiązkowy jest tylko blok if, bloki elif, oraz else, są opcjonalne.

```
x = int(input('Podaj liczbę od 1 do 100'))

if x == 1:                          # Blok if - obowiązkowy
    print('Wpisałeś: 1')
elif x == 2:                        # Dowolna ilość opcjonalnych
    print('Wpisałeś: 2')              bloków elif (ale nie warto
.                                     przesadzać z ich ilością)
.
.
elif x == 100:
    print('Wpisałeś: 100')
else:                               # Opcjonalny blok else, jeśli
    print(':(')                       chcemy obsłużyć każdy inny
                                      przypadek.
```

PS Nie piszcie takich programów :(

IF - ELSE, przykład

```
allowed_ids = [‘123491’, ‘98273166’, ‘8128737’, ‘91737712’]
user_id = input(‘Please, provide your ID:’)

if user_id not in allowed_ids:
    print(‘ACCESS DENIED!’)
else:
    print(‘Welcome to Illuminati Secret Program!’)
    option = input(‘Press 1 to take control over the world, ’
                   ‘or press 2 to check your upcoming appointments:’)
    if option == ‘1’:
        print(‘Taking control over the world… This might take a while.’)
    elif option == ‘2’:
        print(‘No upcoming appointments.’)
    else:
        print(‘Invalid option specified.’)
```

IF - ELSE, przykład

```
import random
number = random.choice(range(1, 101))

if 1 < number <= 5:
    print('Congrats, you have won a car')
elif 5 < number <= 20:
    print('Nice, you have won a TV!')
else:
    print('Sorry, you have won nothing this time :(')
```

```
# random.choice(range(1, 101)) zwróci losowo wybraną wartość z sekwencji
liczb od 1 do 100. Aby takie wyrażenie zadziałało musimy jednak najpierw,
najlepiej na samej górze skryptu wpisać:
    import random
```

IF - ELSE jako operator warunkowy

```
wiek = int(input('Podaj swój wiek:'))

kategoria_wiekowa = 'dziecko' if wiek < 18 else 'dorosły'
```

Takie wyrażenie (tzw operator warunkowy) ma strukturę:

```
    wartość1 if warunek else wartość2
```

Jeśli zostanie spełniony warunek, zwracana jest wartość po lewej stronie (wartość1), w przeciwnym razie - wartość po prawej stronie. W przypadku operatora warunkowego blok else jest wymagany.

IF - ELSE jako operator warunkowy

```
wiek = int(input('Podaj swój wiek:'))
kategoria_wiekowa = 'dziecko' if wiek < 18 else 'dorosły'
```

Zapis powyżej (2 linijka) jest równoznaczny z takim zapisem:

```
if wiek < 18:
    kategoria_wiekowa = 'dziecko'
else:
    kategoria_wiekowa = 'dorosły'
```

Z operatora warunkowego warto korzystać, bo skraca on kod. Jednak jeśli nasz warunek jest długi/skomplikowany, lub chcemy wykonać więcej niż jedną operację, klasyczna forma if-else będzie lepszym rozwiązaniem.

Pętle

1. Pętle for
2. Pętle while

Pętle

Pętle, tak jak blok if-else służą do kontroli przepływu sterowania w naszym programie. Dzięki pętlom możemy sprawić, że określony kawałek kodu wykona się określoną liczbę razy.

W Pythonie występują 2 rodzaje pętli :

- pętle for
- pętle while

Pętle for

Pętla for wykorzystywana jest najczęściej do przejścia po wszystkich elementach sekwencji.

Składnia:

```
for element in sekwencja:
    # co ma się zrobić dla każdego wykonania pętli
```

Gdzie:

for, in to wymagane słowa kluczowe
element - nazwa zmiennej, w której przechowywane będą kolejne wartości zmiennych z…
sekwencja - nazwa zmiennej przechowującej listę, lub range.

Pętle for - przykłady

```
ppl_invited_to_party = [‘Marta’, ‘Piotr’, ‘Monika’, ‘Krzyś’]

for name in ppl_invited_to_party:
    print(f‘Hey {name}, join me on Friday, 7PM at my place!’)
```

```
countries_with_capitals = [[‘Poland’, ‘Warsaw’],
                           [‘Germany’, ‘Berlin’],
                           [‘France’, ‘Paris’]]

# Tzw rozpakowywanie wartości
for country, capital in countries_with_capitals:
    print(f’{capital} is the capital city of {country}’)
```

Pętle for - przykłady

```
for x in range(5):
    print(f'{x} do potęgi to: {x ** 2}')


for _ in range(10):
    print('Hello there!')
    # print(_) <- zadziała
```

Zmienna _ (podkreślnik) jest zmienna specjalną - jeśli coś do niej przypiszemy, jest to znak dla nas i innych ludzi czytających ten kod, że w gruncie rzeczy nie jest do niczego wykorzystywana.

Pętle for - przykłady

```
prime_numbers = [2, 3, 5, 7, 11, 13, 17]

for prime_number in prime_numbers:
    prime_number **= 2

print(prime_numbers) # [2, 3, 5, 7, 11, 13, 17]   <- BEZ ZMIAN :(
```

Zmienna, z której korzystamy w pętli, tak naprawdę ma numery pokojów do kopii kolejnych wartości z listy. Oznacza to, że modyfikując zmienną, nie zmieniamy wartości w tablicy.

Żeby zmodyfikować elementy w tablicy musimy odwołać się do nich po ich indeksie. Można to zrobić za pomocą funkcji enumerate.

Pętle for - enumerate

```
prime_numbers = [2, 3, 5, 7, 11, 13, 17]

for index, prime_number in enumerate(prime_numbers):
    prime_number[index] = prime_number ** 2

print(prime_numbers) # [4, 9, 25, 49, 121, 169, 289]
```

Przy każdym wywołaniu pętli zwracana będzie para wartości, gdzie:

wartość pierwsza - kolejna liczba z rosnącej sekwencji, generowanej przez `enumerate` (domyślnie - rosnąca od 0). W tym przykładzie trafia do zmiennej index.

wartość druga - kolejny element z sekwencji podanej w okrągłych nawiasach za `enumerate`. W tym przykładzie trafia do zmiennej prime_number.

Pętle while

Pętla while wygląda następująco:

```
while warunek:
    # wykonaj coś w pętli!
    # Najlepiej coś co sprawi,
    # że zmienią się wartości
    # zmiennych ewaluowanych
    # w warunku.
```

Przykład pętli nieskończonej:

```
while True:
    print('Daleko jeszcze?')
```

Przykład pętli która wykona się 10X:

```
  index = 0
  while index < 10:
      print('Daleko jeszcze')
      index += 1
```

Pętle while - przykłady

```python
import random
should_end = False

while not should_end:
    random_number = random.randint(1, 10)
    should_end = True if random_number % 2 == 1 else False


# Co tu się dzieje?
random.randint(1, 10)  # Losowy int z przedziału od 1 do 10 włącznie.

should_end = True if random_number % 2 == 1 else False # robi to samo co:

if random_number % 2 == 1:  # Jeśli reszta z dzielenia przez 2 wynosi 1
    should_end = True
else:                       # W przeciwnym wypadku (reszta równa 0)
    should_end = False
```


Listy

Listy

Lista to typ sekwencyjny. Może zawierać w sobie wszystkie inne typy, takie jak na przykład int, float, bool, lub inne listy.

```
eu_countries = [‘Poland’, ‘Germany’, ‘France’]

a_list_with_everything = [1, ‘kotek’, False, [‘a’, ‘b’, ‘c’],
    [‘kawa’, ‘herbata’], eu_countries]
```

Operacje na listach

```
# Tworzenie pustej listy
my_friends = []

# Tworzenie listy z początkowymi wartościami
polish_cities = [‘Poznań’, ‘Warszawa’, ‘Wrocław’]

# Tworzenie listy za pomocą metody str.split()
words = ‘Ala,ma,kota’.split(‘,’) # [‘Ala’, ‘ma’, ‘kota’]

# Tworzenie listy ze zmiennych
name = ‘Jan’
gender = ‘mężczyzna’
occupation = ‘mentor @ PyLadies’
personal_data = [name, gender, occupation]
```

Operacje na listach

```
a_girls_list = [‘Cersei’, ‘The Hound’, ‘Meryn Trant’]

# Dodanie elementu do listy
a_girls_list.append(‘Joffrey’)
# [‘Cersei’, ‘The Hound’, ‘Meryn Trant’, ‘Joffrey’]

# Dodanie kilku elementów do listy (wydłużenie)
a_girls_list.extend([‘The Mountain’, ‘Tywin Lannister’])
# [‘Cersei’, ‘The Hound’, ‘Meryn Trant’, ‘Joffrey’, ‘The Mountain’, ‘Tywin Lannister’]

# Usunięcie elementu z listy - sposób pierwszy (po wartości)
a_girls_list.remove(‘Joffrey’)
# [‘Cersei’, ‘The Hound’, ‘Meryn Trant’, ‘The Mountain’, ‘Tywin Lannister’]

# Usunięcie po indeksie
del a_girls_list[-1]
# [‘Cersei’, ‘The Hound’, ‘Meryn Trant’, ‘The Mountain’]
```

Indeksacja list i tupli

Indeks od lewej strony jest rosnący i zaczyna się od 0 <- BARDZO WAŻNE
Indeks od prawej strony jest malejący i zaczyna się od -1

```
    -3          -2            -1        <- od prawej - maleje od -1
['Cersei', 'The Hound', 'Meryn Trant']
     0           1             2        <- od lewej - rośnie od 0
```

Jeśli przypiszemy powyższą listę do zmiennej a_girls_list, to:

```
a_girls_list[0]  # 'Cersei'
a_girls_list[2]  # 'Meryn Trant'
a_girls_list[-1] # 'Meryn Trant'
a_girls_list[-3] # 'Cersei'

a_girls_list[-3] = 'Joffrey' # ['Joffrey', 'The Hound', 'Meryn Trant']
a_girls_list[1] = 'Cersei'   # ['Joffrey', 'Cersei', 'Meryn Trant']
```

Slicing

```
a_girls_list = [‘Cersei’, ‘The Hound’, ‘Meryn Trant’, ‘Joffrey’]


# Weź wycinek listy od elementu o indeksie 1 do końca
a_girls_list[1:]
[‘The Hound’, ‘Meryn Trant’, ‘Joffrey’]


# Weź wycinek listy od początku do elementu o indeksie -1 (ale bez niego).
a_girls_list[:-1]
[‘Cersei’, ‘The Hound’, ‘Meryn Trant’]


# Weź co drugi element listy (zaczynając od indeksu 0)
a_girls_list[::2]
[‘Cersei’, ‘Meryn Trant’]
```

Slicing

```
a_girls_list = [‘Cersei’, ‘The Hound’, ‘Meryn Trant’, ‘Joffrey’]


# Weź co drugi element listy (zaczynając od indeksu 1)
a_girls_list[1::2]
[‘The Hound’, ‘Joffrey’]


# Weź co drugi element listy (zaczynając od końca)
a_girls_list[-1::-2]
[‘Joffrey’, ‘The Hound’]


# Odwróć listę
a_girls_list[-1::-1]
[‘Joffrey’, ‘Meryn Trant’, ‘The Hound’, ‘Cersei’]
```

Kopiowanie list

```
list1 = [1, 2, 3, 4]
list2 = list2

id(list1) == id(list2) # True, list1 i list2 wskazują na ten sam pokój

Skopiować listę można na 3 sposoby:

# Za pomocą list
list1 = [1, 2, 3]
list2 = list(list1)
# Ładne i całkiem szybkie!

# Za pomocą slicingu
list1 = [1, 2, 3]
list2 = list1[:]
# Brzydkie, ale za to szybkie!


# Za pomocą wbudowanej metody
list1 = [1, 2, 3]
list2 = list1.copy() # Ładne i szybkie


# Rzut oka na slajd 37 sprawi, że przypomnicie sobie o mnożeniu liczb i list.
# Tak - lista * 1, zwróci nową kopię. Ale tak nie róbcie ;)
```

Slicing - z tego kota zrobię konia!

```
word = [‘k’, ‘r’, ‘u’, ‘k’]

word[1:-1] = [‘o’, ‘t’, ‘e’] # [‘k’, ‘o’, ‘t’, ‘e’, ‘k’]
```

Co tu się stało? Slice [1:-1] oznacza - wszystkie elementu poza pierwszym i ostatnim. W tym wypadku wybieramy 2 i 3 element listy. Następuje przypisanie, które w wybrane miejsce (na miejsce elementów drugiego i trzeciego) “wstawia” nową tablicę.

```
# Na miejsce ostatnich 3 elementów wstaw ‘ń’:
word[-3:] = ‘ń’ # [‘k’, ‘o’, ‘ń’]
```

Slicing - trochę magii

Slicing jest bardzo popularny, jednak zapisywany tak jak na poprzednich slajdach może być ciężki do zrozumienia. Dlatego można używać obiektu `slice`, któremu możemy podać te same argumenty, tylko, że po przecinku.

```
my_nubmers = [1, 2, 3, 4, 5]

FIRST_THREE = slice(3)                  # równoważne z [:3]
my_numbers[FIRST_THREE]                 # [1, 2, 3]

ALL_EXCEPT_OUTER = slice(1, -1)         # równoważne z [1:-1]
my_numbers[ALL_EXCEPT_OUTER]            # [2, 3, 4]

EVERY_SECOND = slice(None, None, 2) # równoważne z [::2]
my_numbers[EVERY_SECOND]                # [1, 3, 5]
```

`None` odpowiada tutaj pominięciu wartości!
`# Jeśli w przykładzie pierwszym chcemy pominąć pierwsze 3 elementy -> slice(3, None)`

Slicing działa też na str!

```
‘Ala ma kota’[:3] # ‘Ala’
‘Ala ma kota’[5:7] # ‘ma’
```

Jednakże, każda próba modyfikacji zakończy się błędem:

```
‘Ala ma kota’ = ‘O’
# TypeError: 'str' object does not support item assignment

‘Ala ma kota’ = ‘Ania’
# TypeError: 'str' object does not support item assignment

sentence1 = ‘Ala lubi placki’
sentence2 = ‘Michał ma katar’
sentence3 = ‘Nakarm proszę kota’

sentence = sentence1[:3] + sentence2[5:7] + sentence3[-4:]
print(sentence) # ‘Alamakota’
```

((NAWIASY)) i wcięcia

1. Gdzie używamy nawiasów okrągłych?
2. Co sygnalizują wcięcia?

Ileż tych nawiasów!

Mała ściągawka - gdzie używamy nawiasów okrągłych:

```
print(‘Dzisiaj padał śnieg!’)
```
<- wywołanie funkcji

```
lista.append(1)
```
<- lub metody

```
(2 + 7) ** ((4 + 3) / 2)
```
<- zmiana kolejności wykonania operacji arytmetycznych

```
not (True and (‘’ or [1]))
```
<- zmiana kolejności wykonywania operacji logicznych

```
(‘Bardzo długie zdanie, ’
 ‘które nie mieści się w’
 ‘ jednej linii’)
```
<- rozbicie na kilka linii długiego łańcucha znaków. Kolejnych stringów nie nie musimy oddzielać żadnym znakiem.

Kiedy wcinamy kod?

Kiedy chcemy pogrupować dany kod na bloki, które powinny wykonać się razem np:

```
if x < 4:
    print('Ciekawe')          # ten kod wykona się tylko jeśli
    print('Fascynujące')      # wartość x będzie mniejsza od 10.

print('OK.')                  # ten kod wykona się niezależnie od
print('NUDY.')                # tego jaką wartość przyjmie x.
print('MOŻEMY JUŻ IŚĆ.')      # (czyli w tym przypadku - zawsze)
```

Jeśli poprzednia linijka zakończyła się dwukropkiem, to jest to sygnał, że następna linijka (albo linijki) powinny być wcięte.

Kiedy wcinamy kod?

Ponieważ wcięcia służą zmiany przepływu sterowania programu*, nie możemy ich wstawiać w losowe miejsca!

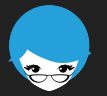
```
import time

print('Losuję przypadkową liczbę z zakresu 0-1000...')
time.sleep(5)                # 'uśpij' program na 5 sekund**
    losowa_liczba = 4        # Błąd! Nie możemy tak z czapy wstawiać sobie
print(losowa_liczba)         # wcięć w kodzie! Program się wywali :(

File "c:/Users/JohnnyPC/Desktop/test.py", line 5
    losowa_liczba = 4
    ^
IndentationError: unexpected indent
```

*czyli - co ma się wykonać zależnie od stanu i wartości zmiennych.
** udajemy, że dzieją się tutaj jakieś ciężkie obliczenia ;)

Kiedy wcinamy kod?



1. Jak bardzo wcinamy kod?
   Tak bardzo, jak chcemy! Najważniejsze, żeby była to zawsze stała liczba znaków. Nie może być tak, że w jednym miejscu kod będzie wcięty na 2 spacje, a w innym na 3. Python nie ogarnie!

2. Serio, na ile tylko chcę?
   Noooo, niby tak.. ale nie do końca!
   Powszechnie używane i rekomendowane jest wcięcie na 4 znaki i takiego warto się trzymać!
   Mniej znaków spowoduje, że kod będzie się zlewał i ciężko będzie ogarnąć, co tam się dzieje.
   Więcej znaków sprawi, że nasz kod będzie za bardzo rozlazły, i trzeba będzie machać głową w lewo i prawo. Wtedy też łatwo się pogubić :(

3. Czy muszę coś z tym robić?
   W sumie to nie! PyCharm sam zrobi wcięcie, jeśli zakończymy linijkę dwukropkiem i wciśniemy Enter. Kiedy wciśniemy Tab - automatycznie wstawi 4 spacje.


QUARTERS ONLY
25¢
1. DROP COIN IN SLOT.
2. PULL HANDLE DOWN ALL THE WAY-LET GO
3. TO CLEAR VISION - TURN RED KNOB.
2414

Q&A

Feedback

tinyurl.com/y8mvzlud

Stay in touch

- materiały i zadania: pyladiesstart.pl
- grupa na FB: goo.gl/GLiX1V
- fanpage: facebook.com/pyladiespoznan/

A STOJĄ ZA TYM: