

Podstawy Informatyki

Instytut Telekomunikacji, TI

dr inż. Jarosław Bułat (c)

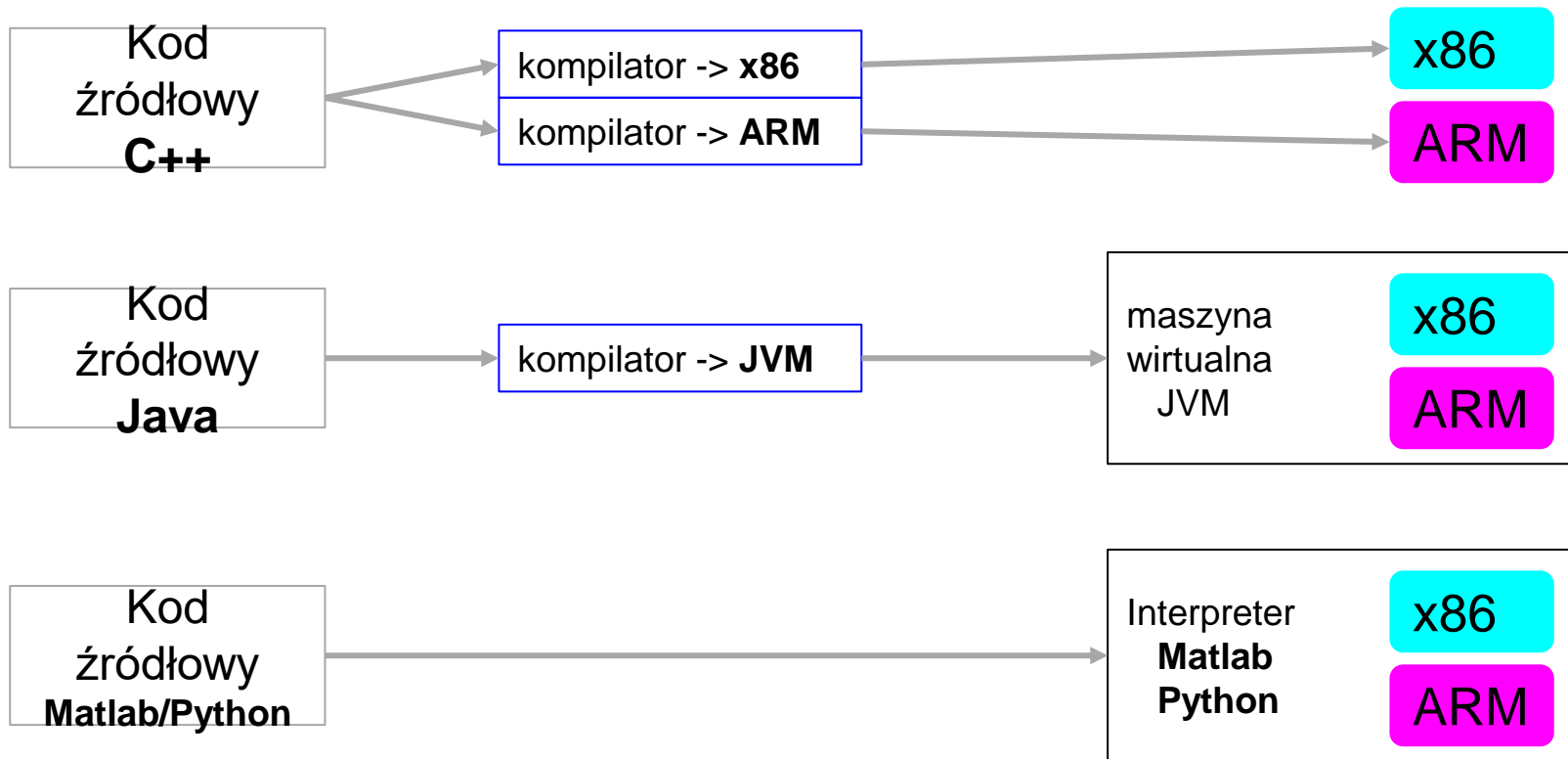
kwant@agh.edu.pl

Plan prezentacji

- » Kompilator vs Interpreter
- » Python - główne założenia
- » Jak uruchomić program
- » Typy danych, konwersje
- » Kolekcje
- » “Zmienne” w języku Python

Chciałbym napisać jeden program
na wiele platform
nawet na takie, które jeszcze nie istnieją

Kompilator vs Interpreter



Kompilator vs Interpreter

Kompilator

- max. szybkość działania programu (asm)
- nie jest wymagany "runtime"
- może działać bare metal
- kompilacja tylko raz
- nieprzenośne
- kompilacja wymaga znajomości docelowej arch.
- trudniej testować

Interpreter

- łatwość pisania i testowania kodu
- przenośność (OS/sprzęt)
- będzie działać na przyszłych architekturach
- mniejsza wydajność
- wymagane większe zasoby (energochłonność)
- trudniej sprzedać aplikację (ukryć algoryt - knowhow)

Python - popularność

- » Język zdobywający coraz większą popularność
- » Popularny jako front-end i back-end
- » Często używany jako “klej” do istniejących bibliotek
 - wrappery do “wszystkich” bibliotek
 - często API jest pisane w Pythonie
- » Chętnie używany przez duże firmy (obecnie 3.x)
- » Dobrze wspiera współczesne metodologie programistyczne: XML, JSON, RESTfull,
- » Wielowątkowość* (dużo funkcjonalności thread-safe) ale **GIL**
- » Świetne wsparcie dla przetwarzania plików
- » Dużo “lukru składniowego” ;-)

Python - wersje

- » W praktyce istnieją dwie wersje języka
 - stara 2.x (najpopularniejsza 2.7)
 - nowa 3.x
- » Nowa wersja (3.x) nie różni się bardzo od starej:
 - zawiera poprawki
 - ujednolicony interfejs (np. **print**)
 - nowe funkcje
- » Wersje 3.x i 2.x nie są kompatybilne, nie da się uruchomić programu 2.x przy użyciu interpretera 3.x
- » Nowa wersja zastępuje starą, problemem jest duża liczba bibliotek działających pod 2.x a nie działających pod 3.x
- » Komercyjny kod jest obecnie najczęściej wydawany jako 3.x

Python - cechy

- » Język programowania wysokiego poziomu, **ogólnego przeznaczenia**
- » Rozbudowany zestaw bibliotek standardowych
- » Kod źródłowy: **czytelny, przejrzysty, zwięzły**
- » Wieloparadygmatowy: obiektowy, imperatywny, strukturalny, funkcyjny (trochę)
- » Dynamiczny system typów (dynamicznie typowany)
- » Automatyczne zarządzanie pamięcią (garbage collector)
- » Licencja języka OS, wiele implementacji (OS) - nawet IoT
- » Często stosowany jako język skryptowy w systemach operacyjnych

Python - język obiektowy

- » W Pythonie (prawie) wszystko jest obiektem
- » Podstawowe typy liczbowe nie są obiektami ale można po nich dziedziczyć
- » Możliwe jest wielokrotne dziedziczenie
- » **Nie ma enkapsulacji** (hermetyzacji), czyli nie da się ukrywać składowych w klasach - **wszystkie składowe są public**

Jak uruchomić

- » Napisać kod w języku Python
- » Zapisać do pliku tekstowego *.py
- » Uruchomić na docelowej maszynie

Jak uruchomić

- » Napisać kod w języku Python
- » Zapisać do pliku tekstowego *.py
- » Uruchomić na docelowej maszynie

plik **ex01.py**

```
print('Hello world\n')  
~  
~  
<" 1L, 23C
```

konsola

```
>python ex01.py  
Hello world  
  
> █
```

Jak uruchomić

- » Napisać kod w języku Python
- » Zapisać do pliku tekstowego *.py
- » Uruchomić na docelowej maszynie

plik **ex01.py**

```
print('Hello world\n')  
~  
~  
<" 1L, 23C
```

konsola

```
>python ex01.py  
Hello world  
  
>|
```

- » Program - interpreter, który uruchamia **ex01.py**
- » **python3 ex01.py** uruchomi interpreter w wersji 3.x

Jak uruchomić

- » Napisać kod w języku Python
- » Zapisać do pliku tekstowego *.py
- » Uruchomić na docelowej maszynie

plik **ex01.py**

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
□
print('Hello world\n')
< zapisano      3,0-1
```

konsola

```
>chmod u+x ex01.py
>./ex01.py
Hello world
>□
```

- » Jeżeli dodam shebang

Jak uruchomić

- » Napisać kod w języku Python
- » Zapisać do pliku tekstowego *.py
- » Uruchomić na docelowej maszynie

plik **ex01.py**

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
print('Hello world\n')
< zapisano      3,0-1
```

konsola

```
>chmod u+x ex01.py
>./ex01.py
Hello world
>
```

- » Jeżeli dodam shebang

Jak uruchomić

- » Napisać kod w języku Python
- » Zapisać do pliku tekstowego *.py
- » Uruchomić na docelowej maszynie

plik **ex01.py**

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
print('Hello world\n')
< zapisano      3,0-1
```

konsola

```
>chmod u+x ex01.py
>./ex01.py
Hello world
>
```

- » Jeżeli dodam shebang i zmienię prawa dostępu

Jak uruchomić

- » Napisać kod w języku Python
- » Zapisać do pliku tekstowego *.py
- » Uruchomić na docelowej maszynie

plik **ex01.py**

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
print('Hello world\n')
< zapisano      3,0-1
```

konsola

```
>chmod u+x ex01.py
>./ex01.py
Hello world
>
```

- » Jeżeli dodam shebang i zmienię prawa dostępu
- » Będę mógł uruchomić program w jak zwykły skrypt

Jak uruchomić

- » Program można pisać w VIMie (jak w C++ to jest zwykły txt)
- » Można używać wiele środowisk IDE do Pythona:
 - PyCharm (drogie ale najlepsze, jest kilka tanich planów)
 - Spyder (scientific, MATLAB replacement*)
 - Jupyter
 - Python Tools for Visual Studio
 - PyDev (plugin do Eclipse)
- » IDE: podpowiada, koloruje, debuguje, uruchamia, integruje

Python == prosty program

```
#!/usr/bin/env python
```

```
# -*- coding:utf-8 -*-
```

```
dir=glob.glob(str(directory)+'/*.json')
```

```
filesNo=len(dir)
```

```
fileCounter=0
```

```
for file in dir:
```

```
    fileCounter+=1
```

```
    userID=file[:-5].rsplit('/',1)[-1]
```

```
    if os.path.isfile(str(directory)+'.txt'):
```

```
        print(' already exists')
```

```
        continue
```

```
print( 'finished as: '+api.me().name )
```

Python == prosty program

» Nie ma znaków końca instrukcji
(średniki w C/C++)

```
#!/usr/bin/env python
```

```
# -*- coding:utf-8 -*-
```

```
dir=glob.glob(str(directory)+'/*.json')
```

```
filesNo=len(dir)
```

```
fileCounter=0
```

```
for file in dir:
```

```
    fileCounter+=1
```

```
    userID=file[:-5].rsplit('/',1)[-1]
```

```
    if os.path.isfile(str(directory)+'.'+file[:-5].rsplit('/',1)[-1]+'.txt'):
```

```
        print(' already exists')
```

```
        continue
```

```
print( 'finished as: '+api.me().name )
```

Python == prosty program

```
#!/usr/bin/env python
```

```
# -*- coding:utf-8 -*-
```

```
dir=glob.glob(str(directory)+'/*.json')
```

```
filesNo=len(dir)
```

```
fileCounter=0
```

```
for file in dir:
```

```
    fileCounter+=1
```

```
    userID=file[:-5].rsplit('/',1)[-1]
```

```
    if os.path.isfile(str(directory)+'.'+userID+'.txt'):
```

```
        print(' already exists')
```

```
        continue
```

```
print( 'finished as: '+api.me().name )
```

- » Nie ma znaków końca instrukcji (średniki w C/C++)
- » Nie ma klamerek {}

Python == prosty program

```
#!/usr/bin/env python
```

```
# -*- coding:utf-8 -*-
```

```
dir=glob.glob(str(directory)+'/*.json')
```

```
filesNo=len(dir)
```

```
fileCounter=0
```

```
for file in dir:
```

```
    fileCounter+=1
```

```
    userID=file[:-5].rsplit('/',1)[-1]
```

```
    if os.path.isfile(str(directory)+'.'+userID+'.txt'):
```

```
        print(' already exists')
```

```
        continue
```

```
print( 'finished as: '+api.me().name )
```

- » Nie ma znaków końca instrukcji (średniki w C/C++)
- » Nie ma klamerek {}
- » Blok kodu (pętla, warunek, funkcja/metoda) jest określony indentacją !!!

Python == prosty program

```
#!/usr/bin/env python
```

```
# -*- coding:utf-8 -*-
```

```
dir=glob.glob(str(directory)+'/*.json')
```

```
filesNo=len(dir)
```

```
fileCounter=0
```

```
for file in dir:
```

```
    fileCounter+=1
```

```
    userID=file[:-5].rsplit('/',1)[-1]
```

```
    if os.path.isfile(str(directory)+'.'+userID+'.txt'):
```

```
        print(' already exists')
```

```
        continue
```

```
print( 'finished as: '+api.me().name )
```

- » Nie ma znaków końca instrukcji (średniki w C/C++)
- » Nie ma klamerek { }
- » Blok kodu (pętla, warunek, funkcja/metoda) jest określony indentacją !!!

Python == prosty program

```
#!/usr/bin/env python
```

```
# -*- coding:utf-8 -*-
```

```
dir=glob.glob(str(directory)+'/*.json')
```

```
filesNo=len(dir)
```

```
fileCounter=0
```

```
for file in dir:
```

```
    fileCounter+=1
```

```
    userID=file[:-5].rsplit('/',1)[-1]
```

```
    if os.path.isfile(str(directory)+'./'+userID+'.txt'):
```

```
        print(' already exists')
```

```
        continue
```

```
print( 'finished as: '+api.me().name )
```

- » Nie ma znaków końca instrukcji (średniki w C/C++)
- » Nie ma klamerek { }
- » Blok kodu (pętla, warunek, funkcja/metoda) jest określony indentacją !!!

Python == prosty program

```
#!/usr/bin/env python
```

```
# -*- coding:utf-8 -*-
```

```
dir=glob.glob(str(directory)+'/*.json')
```

```
filesNo=len(dir)
```

```
fileCounter=0
```

```
for file in dir:
```

```
    fileCounter+=1
```

```
    userID=file[:-5].rsplit('/',1)[-1]
```

```
    if os.path.isfile(str(directory)+'./txt'):
```

```
        print(' already exists')
```

```
        continue
```

```
print( 'finished as: '+api.me().name )
```

- » Nie ma znaków końca instrukcji (średniki w C/C++)
- » Nie ma klamerek {}
- » Blok kodu (pętla, warunek, funkcja/metoda) jest określony indentacją !!!
- » Nie da się napisać kodu źle sformatowanego - nie uruchomi się :-))))))))))

koniec niechlujnego kodu :-)

```
%PLL implementation
for n=2:length(Signal)
    vco(n)=conj(exp(j*(0+ phi_hat(n-1))));%Compute VCO
    phd_output(n)=imag(Signal(n)*vco(n));%Complex multiply VCO x
    e(n)=e(n-1)+(beta+alpha)*phd_output(n)-beta*phd_output(n-1);
    phi_hat(n)=phi_hat(n-1)+e(n);%Update VCO
end;
dif(1:25600)=0;
for x = 1:25600
    dif(x)=abs((real(Signal(x))-real(vco(x))));
end
f=sum(dif)
end
```

Python == prosty program

```
#!/usr/bin/env python
```

```
# -*- coding:utf-8 -*-
```

```
dir=glob.glob(str(directory)+'/*.json')
```

```
filesNo=len(dir)
```

```
fileCounter=0
```

```
for file in dir:
```

```
    fileCounter+=1
```

```
    userID=file[:-5].rsplit('/',1)[-1]
```

```
    if os.path.isfile(str(directory)+'./txt'):
```

```
        print(' already exists')
```

```
        continue
```

```
print( 'finished as: '+api.me().name )
```

- » Nie ma znaków końca instrukcji (średniki w C/C++)
- » Nie ma klamerek {}
- » Blok kodu (pętla, warunek, funkcja/metoda) jest określony indentacją !!!
- » Nie da się napisać kodu źle sformatowanego - nie uruchomi się :-))))))))))
- » Jedna linia - jedno wyrażenie
- » dwukropek rozpoczyna “body” funkcji, pętli, warunku
- » **kod jest zwięzły ale czytelny!**

W Pythonie **wartości** a nie zmienne posiadają **typy** język dynamicznie typowany

Typy zmiennych

a=1
b=1.02
c=2+1j*3
d=True
e='abc'

int
float
complex
bool
str

- » Język dynamicznie typowany
- » Nie ma deklaracji zmiennych
- » Zmienna sama zostanie utworzona podczas inicjalizacji, jej typ zostanie wydedukowany

Typy zmiennych

```
a=1
b=1.02
c=2+1j*3
d=True
e='abc'
```

```
# int
# float
# complex
# bool
# str
```

```
print(type(a))
print(type(b))
print(type(c))
print(type(d))
print(type(e))
```

- » Język dynamicznie typowany
- » Nie ma deklaracji zmiennych
- » Zmienna sama zostanie utworzona podczas inicjalizacji, jej typ zostanie wydedukowany
- » Wynik:
 - <type 'int'>
 - <type 'float'>
 - <type 'complex'>
 - <type 'bool'>
 - <type 'str'>
- » Funkcja `type(...)` zwraca typ zmiennej (obiektu)

Konwersja typów

a=1
b=1.92
c=a+b

int
float
float

» Konwersja automatyczna jeżeli
to oczywiste: float == int + float

Konwersja typów

```
a=1
b=1.92
c=a+b
d=int(c)
(truncating)
print(d)
```

```
# int
# float
# float
# int
# 2
```

- » Konwersja automatyczna jeżeli to oczywiste: float == int + float
- » Konwersja **jawna**

Konwersja typów

```
a=1
b=1.92
c=a+b
d=int(c)
(truncating)
print(d)

s=str(b)
print(b)
```

```
# int
# float
# float
# int
# 2
```

- » Konwersja automatyczna jeżeli to oczywiste: `float == int + float`
- » Konwersja jawna
- » Konwersja na typ `str` żeby wyświetlać obiekt jako tekst

Konwersja typów

```
a=1                # int
b=1.92             # float
c=a+b              # float
d=int(c)           # int
(truncating)
print(d)           # 2
```

```
s=str(b)
print(b)
```

```
z='abc'+s
print('text: '+str(c))
```

- » Konwersja automatyczna jeżeli to oczywiste: float == int + float
- » Konwersja jawna
- » Konwersja na typ str żeby wyświetlać obiekt jako tekst
- » Często używane aby utworzyć tekst z wartościami zmiennych

Konwersja typów

```
a=1                # int
b=1.92             # float
c=a+b              # float
d=int(c)           # int
(truncating)
print(d)           # 2
```

```
s=str(b)
print(b)
```

```
z='abc'+s
print('text: '+str(c))
```

```
print('text: {}'.format(c, b))
print(f'text: {c} ')
print(f'text: {c:03} ')
```

- » Konwersja automatyczna jeżeli to oczywiste: float == int + float
- » Konwersja jawna
- » Konwersja na typ str żeby wyświetlać obiekt jako tekst
- » Często używane aby utworzyć tekst z wartościami zmiennych

← mnóstwo lukru składniowego :-)

Konwersja typów

```
a=1                # int
b=1.92             # float
c=a+b              # float
d=int(c)           # int
(truncating)
print(d)           # 2
```

```
s=str(b)
print(b)
```

```
z='abc'+s
print('text: '+str(c))
```

```
z='abc'+"123"
print(z)
```

- » Konwersja automatyczna jeżeli to oczywiste: float == int + float
- » Konwersja jawna
- » Konwersja na typ str żeby wyświetlać obiekt jako tekst
- » Często używane aby utworzyć tekst z wartościami zmiennych
- » **Dodawanie** tekstu do siebie to łączenie (concatenation)
- » 'text' albo "123"

Konwersja typów

```
a=1                # int
b=1.92             # float
c=a+b              # float
d=int(c)           # int
(truncating)
print(d)           # 2

s=str(b)
print(b)

z='abc'+s
print('text: '+str(c))

cpx=2+1j*3
print(str(cpx)+' complex\n')
```

- » Konwersja automatyczna jeżeli to oczywiste: float == int + float
- » Konwersja jawna
- » Konwersja na typ str żeby wyświetlać obiekt jako tekst
- » Często używane aby utworzyć tekst z wartościami zmiennych
- » Dodawanie tekstu do siebie to łączenie (concatenation)
- » 'text' albo "123"
- » wbudowana konwersja, również złożonych typów, rezultat: (4-7j) complex

Kolekcje

czyli grupowanie danych

Kolekcje

- » W języku C/C++ do grupowania danych jest tablica
 - tablica dowolnych obiektów (liczby, struktury, ...)
 - większe możliwości za pomocą biblioteki std::

Kolekcje

- » W języku C/C++ do grupowania danych jest tablica
 - tablica dowolnych obiektów (liczby, struktury, ...)
 - większe możliwości za pomocą biblioteki std::
- » Python:
 - listy
 - krotki (tuple)
 - słownik (dict)
 - tablice (array, NumPy)
 - zbiory (set/frozenset)

Kolekcje - lista

```
colors = ['red', 'green']  
print(colors[0])
```

```
numbers = [-1, 0, 10, 1]  
print(numbers[1:])  
# [0, 10, 1]
```

```
print(numbers[1:3])  
# [0, 10]
```

```
print(numbers[:3])  
# [-1, 0, 10]
```

```
print(numbers[:4])  
print(numbers[:5])  
# [-1, 0, 10, 1]
```

- » Lista jest najpopularniejszym typem kolekcji
- » Elementami mogą być dowolne obiekty: napisy, liczby, obiekty klas
- » Indeksowanie []
- » Indeksowanie zakresami (od ... do) - slicing
- » Listę można rozszerzać dynamicznie
- » Nie jest tak efektywna jak tablica w C/C++ (duży narzut pamięci)

Kolekcje - lista

```
x = [1, 2, 3, 'c']  
print(x)  
print(len(x))
```

4

- » Lista może zawierać różne typy!
- » Podstawowe operacje na listach:
 - liczba elementów

Kolekcje - lista

```
x = [1, 2, 3, 'c']  
print(x)  
print(len(x))           # 4  
  
print(3 in x)           # True  
print(0 in x)           # False  
print('c' in x)         # True
```

- » Lista może zawierać różne typy!
- » Podstawowe operacje na listach:
 - liczba elementów
 - sprawdzenie czy zawiera element

Kolekcje - lista

```
x = [1, 2, 3, 'c']  
print(x)  
print(len(x))           # 4  
  
print(3 in x)           # True  
print(0 in x)           # False  
print('c' in x)         # True  
  
print(x*2)  
# [1, 2, 3, 'c', 1, 2, 3, 'c']
```

- » Lista może zawierać różne typy!
- » Podstawowe operacje na listach:
 - liczba elementów
 - sprawdzenie czy zawiera element
 - powielenie listy

Kolekcje - lista

```
x = [1, 2, 3, 'c']  
print(x)  
print(len(x))           # 4  
  
print(3 in x)           # True  
print(0 in x)           # False  
print('c' in x)         # True  
  
print(x*2)  
# [1, 2, 3, 'c', 1, 2, 3, 'c']
```

- » Lista może zawierać różne typy!
- » Podstawowe operacje na listach:
 - liczba elementów
 - sprawdzenie czy zawiera element
 - powielenie listy

często: `[0]*100`
do utworzenia 100-el. listy
wypełnionej zerami

Kolekcje - lista

```
x = [1, 2, 3, 'c']  
print(x)  
print(len(x))           # 4  
  
print(3 in x)            # True  
print(0 in x)            # False  
print('c' in x)          # True  
  
print(x*2)  
# [1, 2, 3, 'c', 1, 2, 3, 'c']  
  
print(x+[3, 2])  
# [1, 2, 3, 'c', 3, 2]
```

- » Lista może zawierać różne typy!
- » Podstawowe operacje na listach:
 - liczba elementów
 - sprawdzenie czy zawiera element
 - powielenie listy
 - łączenie listy (concatenation)

Kolekcje - lista

```
x = [1, 2, 3, 'c']
print(x)
print(len(x))          # 4

print(3 in x)           # True
print(0 in x)           # False
print('c' in x)         # True

print(x*2)
# [1, 2, 3, 'c', 1, 2, 3, 'c']

print(x+[3, 2])
# [1, 2, 3, 'c', 3, 2]

x[3]=[0,1]
print(x) # [1, 2, 3, [0, 1]]
```

- » Lista może zawierać różne typy!
- » Podstawowe operacje na listach:
 - liczba elementów
 - sprawdzenie czy zawiera element
 - powielenie listy
 - łączenie listy (concatenation)
 - modyfikacja elementu listy (element listy może być inną listą)

Kolekcje - krotka

tuples

```
colors = ('red', 'green', 'red')  
print(colors[0])
```

```
print(len(colors))           # 3  
print(colors.count('red'))   # 2  
print(colors.index('red'))   # 0
```

- » Uporządkowana kolekcja stałych wartości
- » Podobna do listy ale nie można zmieniać po zainicjalizowaniu (wartości, wielkości, typu)
- » Inicjalizowana w nawiasach okrągłych
- » Indeksowana nawiasami kwadratowymi

Kolekcje - krotka

tuples

```
colors = ('red', 'green', 'red')  
print(colors[0])
```

```
print(len(colors))           # 3  
print(colors.count('red'))   # 2  
print(colors.index('red'))   # 0
```

```
colors.append('blue')  
colors[1] = 'black'
```

- » Uporządkowana kolekcja stałych wartości
- » Podobna do listy ale nie można zmieniać po zainicjalizowaniu (wartości, wielkości, typu)
- » Inicjalizowana w nawiasach okrągłych
- » Indeksowana nawiasami kwadratowymi
- » Jest obiektem, **wiele różnych udogodnień** (jak wszystkie kolekcje)

Kolekcje - krotka

tuples

```
colors = ('red', 'green', 'red')  
print(colors[0])
```

```
print(len(colors))           # 3  
print(colors.count('red'))   # 2  
print(colors.index('red'))   # 0
```

```
colors.append('blue')  
colors[1] = 'black'
```

- » Uporządkowana kolekcja stałych wartości
- » Podobna do listy ale nie można zmieniać po zainicjalizowaniu (wartości, wielkości, typu)
- » Inicjalizowana w nawiasach okrągłych
- » Indeksowana nawiasami kwadratowymi
- » Jest obiektem, wiele różnych udogodnień (jak wszystkie kolekcje)
- » **immutable == niezmienny**

Kolekcje - słownik

dict

```
bw = {'white': 255, 'gray': 128 }  
print(bw['white']) # 255
```

```
record = {  
    'Name': 'James Bond',  
    'Age': 34  
}
```

```
record['Name'] = 'Smith'  
record['Age'] += 10  
print(record)  
# {'Age': 44, 'Name': 'Smith'}
```

- » Kolekcja do przechowywania danych w postaci par:
 'klucz': wartość
- » Klucz od wartości rozdziela dwukropek :
- » Kluczem może być dowolny niezmienny (immutable) typ
- » W jednym słowniku można:
 - mieszać typy kluczy
 - mieszać typy wartości
- » Klucze muszą być unikalne
- » Wartości można modyfikować stosownie do ich typów

Kolekcje - tablica

```
from array import *  
# array
```

```
x = array('I', [1, 2, 3])  
print(type(x))  
# <type 'array.array'>
```

```
Import numpy as np
```

```
x = np.array([1, 2, 3])  
print(type(x))  
# <class 'numpy.ndarray'>
```

- » Efektywny sposób przechowywania liczb
- » W przeciwieństwie do list, tablice mają podobny narzut pamięci jak te z C/C++
- » Podczas deklaracji należy podać typ:
 - 'b' signed char (1 byte)
 - 'B' unsigned char (1 byte)
 - 'l' signed long (4 bytes)
 - 'f' float (4 bytes)

Zmienne w Pytongu ;-)

inaczej niż w C/C++

Zmienne

//

// language: C/C++

//

int x = **10**;

- » W języku C/C++ (Java, C#, ...)
zmienne mają:
- nazwę
 - typ
 - wartość
 - miejsce w pamięci

Zmienne

```
//
```

```
// language: C/C++
```

```
//
```

```
int x = 10;
```

```
#
```

```
# language: Python
```

```
#
```

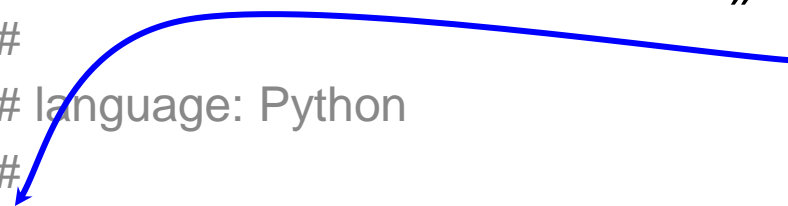
```
x = 10
```

- » W języku C/C++ (Java, C#, ...) zmienne mają:
 - nazwę
 - typ
 - wartość
 - miejsce w pamięci
- » W języku Python zmienne mają:
 - nazwę (wyłącznie nazwę...)

Zmienne

```
//  
// language: C/C++  
//  
int x = 10;
```

```
#  
# language: Python  
#  
x = 10
```

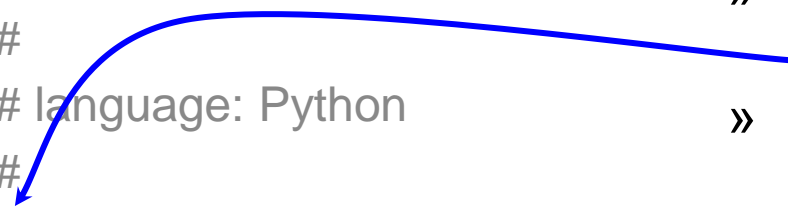


- » W języku C/C++ (Java, C#, ...)
zmienne mają:
 - nazwę
 - typ
 - wartość
 - miejsce w pamięci
- » W języku Python zmienne mają:
 - nazwę (wyłącznie nazwę...)

Zmienne

```
//  
// language: C/C++  
//  
int x = 10;
```

```
#  
# language: Python  
#  
x = 10
```



- » W języku C/C++ (Java, C#, ...)
zmienne mają:
 - nazwę
 - typ
 - wartość
 - miejsce w pamięci
- » W języku Python zmienne mają:
 - nazwę (wyłącznie nazwę...)
- » W języku Python obiekty mają:
 - typ
 - wartość
 - miejsce w pamięci

Zmienne

```
//  
// language: C/C++  
//  
int x = 10;
```

```
#  
# language: Python  
#  
x = 10
```

- » W języku C/C++ (Java, C#, ...)
zmienne mają:
 - nazwę
 - typ
 - wartość
 - miejsce w pamięci
- » W języku Python zmienne mają:
 - nazwę (wyłącznie nazwę...)
- » W języku Python obiekty mają:
 - typ
 - wartość
 - miejsce w pamięci

Zmienne

#

language: Python

#

x = 10

- » W języku Python **zmienne** mają:
 - nazwę (wyłącznie nazwę...)
- » W języku Python **obiekty** mają:
 - typ
 - wartość
 - miejsce w pamięci

Zmienne

#

language: Python

#

x = 10

- » W języku Python **zmienne** mają:
 - nazwę (wyłącznie nazwę...)
- » W języku Python **obiekty** mają:
 - typ
 - wartość
 - miejsce w pamięci
- » Zmienna to jest referencja do istniejącego obiektu

Zmienne

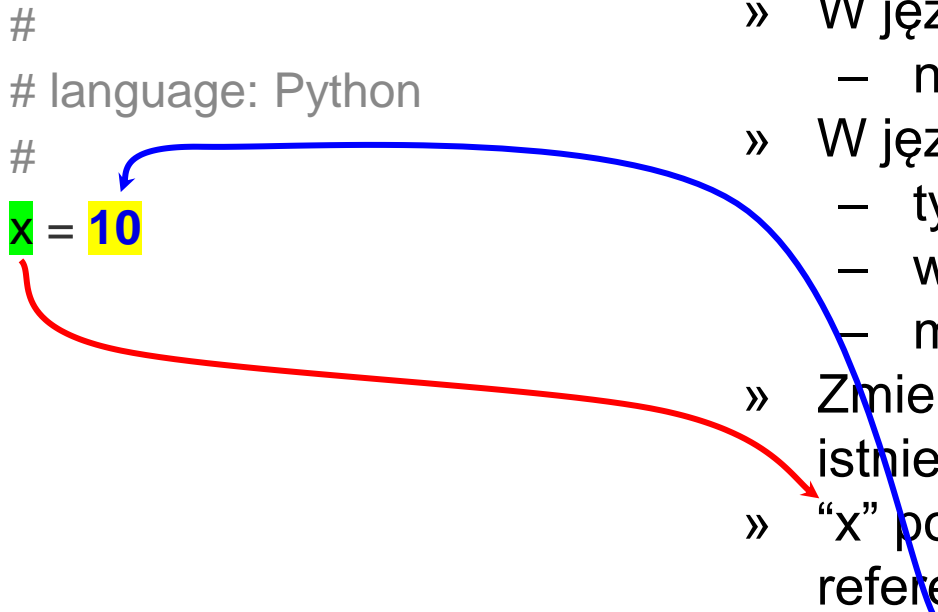
```
#  
# language: Python  
#  
x = 10
```



- » W języku Python **zmienne** mają:
 - nazwę (wyłącznie nazwę...)
- » W języku Python **obiekty** mają:
 - typ
 - wartość
 - miejsce w pamięci
- » Zmienna to jest referencja do istniejącego obiektu
- » “x” powinien być traktowany jako referencja albo wskaźnik do obiektu (w sensie języka C/C++)

Zmienne

```
#  
# language: Python  
#  
x = 10
```



- » W języku Python **zmienne** mają:
 - nazwę (wyłącznie nazwę...)
- » W języku Python **obiekty** mają:
 - typ
 - wartość
 - miejsce w pamięci
- » Zmienna to jest referencja do istniejącego obiektu
- » “x” powinien być traktowany jako referencja albo wskaźnik do obiektu (w sensie języka C/C++)

Zmienne

» Co to oznacza?

#

language: Python

#

x = 10

Zmienne

```
#
```

```
# language: Python
```

```
#
```

```
x = 10
```

» Co to oznacza?

- utworzyłem obiekt “10”
- nazwałem go “x”
- od teraz mogę używać “x” jako nazwy (referencji, wskaźnika) która odnosi się do obiektu “10”

Zmienne

```
#  
# language: Python  
#  
x = 10  
x = 20
```

- » Co to oznacza?
 - utworzyłem obiekt “10”
 - nazwałem go “x”
 - od teraz mogę używać “x” jako nazwy (referencji, wskaźnika) która odnosi się do obiektu “10”
- » Utworzyłem obiekt “20” i teraz “x” będzie wskazywał na obiekt “20”

Zmienne

```
#  
# language: Python  
#  
x = 10  
x = 20
```

- » Co to oznacza?
 - utworzyłem obiekt “10”
 - nazwałem go “x”
 - od teraz mogę używać “x” jako nazwy (referencji, wskaźnika) która odnosi się do obiektu “10”
- » Utworzyłem obiekt “20” i teraz “x” będzie wskazywał na obiekt “20”
- » Jeżeli obiekt “10” nie będzie dalej używany to zostanie automatycznie usunięty z pamięci (garbage collector)

Zmienne

#

language: Python

#

a = 1

- » Sprawdźmy to
- » `id(x)` zwraca unikalny identyfikator obiektu (adres pamięci - wskaźnik!!!)

Zmienne

a=1

id(1) 94915617105304

id(a) 94915617105304

- » Sprawdźmy to
- » id(x) zwraca unikalny identyfikator obiektu (adres pamięci - wskaźnik!!!)
- » id(a) == id(1) dlatego "a" jest referencją na obiekt "1"

Zmienne

a=1

id(1) 94915617105304

id(a) 94915617105304

a=2

id(a) 94915617105280

id(2) 94915617105280

» Sprawdźmy to

» id(x) zwraca unikalny identyfikator obiektu (adres pamięci - wskaźnik!!!)

» id(a) == id(1) dlatego ponieważ “a” jest referencją na obiekt “1”

» id(a) == id(2) != id(1)

Zmienne

a=1

id(1) 94915617105304

id(a) 94915617105304

a=2

id(a) 94915617105280

id(2) 94915617105280

a=1

id(a) 94915617105304

» Sprawdźmy to

» id(x) zwraca unikalny identyfikator obiektu (adres pamięci - wskaźnik!!!)

» id(a) == id(1) dlatego ponieważ "a" jest referencją na obiekt "1"

» id(a) == id(2) != id(1)

» ponownie, id(a) == id(1)

Zmienne

a=1

id(1) 94915617105304

id(a) 94915617105304

a=2

id(a) 94915617105280

id(2) 94915617105280

a=1

id(a) 94915617105304

a=1+2

id(a) 94915617105256

id(1+2) 94915617105256

id(3) 94915617105256

- » Sprawdźmy to
- » id(x) zwraca unikalny identyfikator obiektu (adres pamięci - wskaźnik!!!)
- » id(a) == id(1) dlatego ponieważ “a” jest referencją na obiekt “1”
- » id(a) == id(2) != id(1)
- » ponownie, id(a) == id(1)

Zmienne

a=1

id(1) 94915617105304

id(a) 94915617105304

a=2

id(a) 94915617105280

id(2) 94915617105280

a=1

id(a) 94915617105304

a=1+2

id(a) 94915617105256

id(1+2) 94915617105256

id(3) 94915617105256

- » Sprawdźmy to
- » id(x) zwraca unikalny identyfikator obiektu (adres pamięci - wskaźnik!!!)
- » id(a) == id(1) dlatego ponieważ “a” jest referencją na obiekt “1”
- » id(a) == id(2) != id(1)
- » ponownie, id(a) == id(1)
- » 3 = 1+2
 - “3” jest obiektem o identyfikatorze 94915617105256

Zmienne

a=1

id(1) 94915617105304

id(a) 94915617105304

a=2

id(a) 94915617105280

id(2) 94915617105280

a=1

id(a) 94915617105304

a=1+2

id(a) 94915617105256

id(1+2) 94915617105256

id(3) 94915617105256

- » Sprawdźmy to
- » id(x) zwraca unikalny identyfikator obiektu (adres pamięci - wskaźnik!!!)
- » id(a) == id(1) dlatego ponieważ “a” jest referencją na obiekt “1”
- » id(a) == id(2) != id(1)
- » ponownie, id(a) == id(1)
- » 3 = 1+2
 - “3” jest obiektem o identyfikatorze 94915617105256
 - id obiektu 3 jest różne 2 i 1

Zmienne

» A co z typami?

Zmienne

```
a = 1  
type(a)  
<type 'int'>
```

- » A co z typami?
- » `type(...)` jest wbudowaną funkcją która zwraca typ obiektu (lub zmiennej)

Zmienne

```
a = 1  
type(a)  
<type 'int'>
```

- » A co z typami?
- » `type(...)` jest wbudowaną funkcją która zwraca typ obiektu (lub zmiennej)
- » “1” jest typu integer (int)

Zmienne

```
a = 1
```

```
type(a)
```

```
<type 'int'>
```

```
a = 'xxx'
```

```
type(a)
```

```
<type 'str'>
```

- » A co z typami?
- » `type(...)` jest wbudowaną funkcją która zwraca typ obiektu (lub zmiennej)
- » `"1"` jest typu integer (int)
- » `'xxx'` jest typu string (str)

Zmienne

```
a = 1
```

```
type(a)
```

```
<type 'int'>
```

```
a = 'xxx'
```

```
type(a)
```

```
<type 'str'>
```

- » A co z typami?
- » `type(...)` jest wbudowaną funkcją która zwraca typ obiektu (lub zmiennej)
- » `"1"` jest typu integer (int)
- » `'xxx'` jest typu string (str)

W języku Python “tak zwane” zmienne to referencje do obiektów

Zmienna nie ma typu, mają obiekt do którego wskazuje ta zmienna

Dziękuję