

# Podstawy Informatyki

**Katedra Telekomunikacji, TI**

dr inż. Jarosław Bułat (c)

[kwant@agh.edu.pl](mailto:kwant@agh.edu.pl)

# Plan prezentacji

- » Warunki
- » Pętle
- » Klasy
- » Wyjątki
- » Biblioteki
- » Pliki

# Warunek

if ...:

# Python: warunek

```
x = 0
```

```
if x == 0:  
    print('x == 0')  
elif x == 1:  
    print('x == 1')  
else:  
    print('else')
```

- » Warunek rozbudowanej konfiguracji
  - dowolna liczba **elif**
  - opcjonalne **elif** oraz **else**
- » Wyrażenie warunkowe jest pomiędzy słowem kluczowym a znakiem :

# Python: warunek

```
x = 0
```

```
if x == 0:  
    print('x == 0')  
elif x == 1:  
    print('x == 1')  
else:  
    print('else')
```

- » Warunek rozbudowanej konfiguracji
  - dowolna liczba **elif**
  - opcjonalne **elif** oraz **else**
- » Wyrażenie warunkowe jest pomiędzy słowem kluczowym a znakiem :
- » Instrukcje wewnątrz warunku są wyróżnione indentacją

# Python: warunek zagnieżdżony

```
x = 0  
y = 1
```

```
if x == 0:  
    print('x == 0')  
    if y == 1:  
        print('x == 0')  
        print('y == 1')  
elif x == 1:  
    print('x == 1')  
else:  
    print('else')
```

» Instrukcje wewnątrz warunku są wyróżnione indentacją

# Python: warunek zagnieżdżony

```
x = 0  
y = 1
```

```
if x == 0:  
    print('x == 0')  
    if y == 1:  
        print('x == 0')  
        print('y == 1')  
elif x == 1:  
    print('x == 1')  
else:  
    print('else')
```

- » Instrukcje wewnątrz warunku są wyróżnione indentacją
  - pierwszy poziom

# Python: warunek zagnieżdżony

```
x = 0
y = 1

if x == 0:
    print('x == 0')
    if y == 1:
        print('x == 0')
        print('y == 1')
elif x == 1:
    print('x == 1')
else:
    print('else')
```

- » Instrukcje wewnątrz warunku są wyróżnione indentacją
- pierwszy poziom
  - drugi poziom



# Python: warunek złożony

```
x = 0  
y = 1  
z = 2
```

```
if x == 0 and y == 0:  
    print('x=y=0')  
if (x == 0 or y == 0) and z <= 0:  
    print('...')
```

- » Instrukcje warunkowe mogą mieć dowolnie skomplikowane wyrażenie logiczne
- » Nawiasy wymuszają kolejność operacji

# Python: warunek złożony

```
x = 0  
y = 1  
z = 2
```

```
if x == 0 and y == 0:  
    print('x=y=0')  
if (x == 0 or y == 0) and z <= 0:  
    print('...')
```

```
if x:  
    print('x')
```

```
if bool(x):  
    print('x')
```

- » Instrukcje warunkowe mogą mieć dowolnie skomplikowane wyrażenie logiczne
- » Nawiasy wymuszają kolejność operacji
- » Równoznaczne wyrażenia:
  - `f x`:
  - `if bool(x)`:

# Python: warunek

## » Nie ma instrukcji switch...

- można ją zastąpić if ... elif ... elif ...
- albo wykorzystując funkcję i słownik

<https://stackoverflow.com/questions/60208/replacements-for-switch-statement-in-python>

Pęta  
for ...:

# Python: pętle

- » Python implementuje dwie pętle:
  - **for**
  - **while**
- » Możliwe są zagnieżdżenia
- » Są pętle **for ... else**
- » Nie ma pętli znanej z C/C++:  
**for (i=0; i<10; ++i)**
- » Jest rekurencja (można ją użyć jako pętli)

# Python: pętle

```
colors = ['red','green','blue']
```

```
for color in colors:  
    print(color)
```

```
# red  
# green  
# blue
```

- » Python implementuje dwie pętle:
  - **for**
  - **while**
- » Możliwe są zagnieżdżenia
- » Są pętle **for ... else**
- » **Nie ma pętli znanej z C/C++:**  
    **for (i=0; i<10; ++i)**
- » Jest rekurencja (można ją użyć jako pętli)

# Python: pętla for

```
colors = ['red', 'green', 'blue']
```

```
for color in colors:  
    print(color)
```

```
# red  
# green  
# blue
```

- » W języku Python, najlepiej jest wykonywać pętle na obiekcie iterowalnym
- » Zmienna `color`, po kolei przyjmuje wszystkie wartości obiektu iterowalnego
  - obiekt iterowalny to właściwość np. kolekcji, można samemu tworzyć obiekty o takiej właściwości
  - zmienna w pętli może być obiektem dowolnego typu: liczba, napis, lista, obiekt, ..

# Python: pętla for

```
colors = ['red','green','blue']
```

```
for color in colors:
    if color == 'green':
        print('found green')
        break;
else:
    print('green not found')

# found green
```

- » Pętla **for ... else**:
- » Instrukcje warunku **else**: zostaną wykonane po wyczerpaniu możliwości iteracji (po zakończeniu pętli)
- » Można efektywnie tworzyć konstrukcje: **przeszukaj zbiór X jeżeli znalazłeś to “A” jeżeli nie to “B”**



# Python: pętla for

```
colors = ['red','green','blue']
```

```
for color in colors:  
    if color == 'green':  
        print('found green')  
        break;
```

```
else:  
    print('green not found')
```

```
# found green
```

```
if 'green' in colors:  
    print('found green')  
else:  
    print('green not found')
```

- » Pętla **for ... else**:
- » Instrukcje warunku **else**: zostaną wykonane po wyczerpaniu możliwości iteracji (po zakończeniu pętli)
- » Można efektywnie tworzyć konstrukcje: **przeszukaj zbiór X jeżeli znalazłeś to “A” jeżeli nie to “B”**
- » Można jeszcze prościej ale tylko w prostym przypadku

# Python: pętla for

```
for i in range(4):  
    print(i)  
# 0 1 2 3
```

```
for i in range(2,4):  
    print(i)  
# 2 3
```

```
for i in range(0,-5, -2):  
    print(i)  
# 0 -2 -4
```

- » Ekwiwalent **for(i=0; i<10; ++i);**
- » Funkcja **range(..)** wygeneruje listę wszystkich wartości iteratorów

# Python: pętla for

```
for i in range(4):  
    print(i)  
# 0 1 2 3
```

```
for i in range(2,4):  
    print(i)  
# 2 3
```

```
for i in range(0,-5, -2):  
    print(i)  
# 0 -2 -4
```

- » Ekwiwalent **for(i=0; i<10; ++i);**
- » Funkcja range(..) wygeneruje listę wszystkich wartości iteratorów
  - **początek** (0 jeżeli nie podano)

# Python: pętla for

```
for i in range(4):  
    print(i)  
# 0 1 2 3
```

```
for i in range(2,4):  
    print(i)  
# 2 3
```

```
for i in range(0,-5, -2):  
    print(i)  
# 0 -2 -4
```

- » Ekwiwalent **for(i=0; i<10; ++i);**
- » Funkcja range(..) wygeneruje listę wszystkich wartości iteratorów
  - **początek** (0 jeżeli nie podano)
  - **koniec** (mniejsze niż)

# Python: pętla for

```
for i in range(4):  
    print(i)  
# 0 1 2 3
```

```
for i in range(2,4):  
    print(i)  
# 2 3
```

```
for i in range(0,-5, -2):  
    print(i)  
# 0 -2 -4
```

- » Ekwiwalent **for(i=0; i<10; ++i);**
- » Funkcja range(..) wygeneruje listę wszystkich wartości iteratorów
  - **początek** (0 jeżeli nie podano)
  - **koniec** (mniejsze niż)
  - **krok**

# Python: pętla for

```
for i in range(4):  
    print(i)  
# 0 1 2 3
```

```
for i in range(2,4):  
    print(i)  
# 2 3
```

```
for i in range(0,-5,-2):  
    print(i)  
# 0 -2 -4
```

- » Ekwiwalent **for(i=0; i<10; ++i);**
- » Funkcja range(..) wygeneruje listę wszystkich wartości iteratorów
  - **początek** (0 jeżeli nie podano)
  - **koniec** (mniejsze niż)
  - **krok**
- » Pełna funkcjonalność pętli standardowej pętli z C/C++

# Python: pętla for

```
x = [2, 5, 'afa', [3, 4]]
```

```
for i in range(len(x)):  
    print(x[i])
```

» Można użyć range, żeby symulować pętlę  
for(i=0; i<size; ++i)

# Python: pętla for

```
x = [2, 5, 'afa', [3, 4]]
```

```
for i in range(len(x)):  
    print(x[i])
```

```
for var in x:  
    print(var)
```

```
# 2  
# 5  
# afa  
# [3, 4]
```

- » Można użyć range, żeby symulować pętlę  
for(i=0; i<size; ++i)
- » Tylko po co? skoro można znacznie prościej to jest:

Python-way

tak należy projektować i implementować pętlę w Pythonie



# Python: range

```
i = range(5,10)
print(i)
print(i[0])
print(i[4])
```

```
# Python 2.x
# [5, 6, 7, 8, 9]
# 5
# 9
```

```
# Python 3.x
# range(5, 10)
# 5
# 9
```

- » Generator listy z liczbami
- » Funkcja range tworzy listę indeksów
- » Python 2.x tworzy zwyczajną listę o odpowiedniej długości
- » Python 3.x zwraca obiekt iterowalny, który jest tzw. “lazy-evaluation”
  - cała lista nie jest generowana
  - elementy listy są generowane w miarę potrzeb
  - można go użyć tak jak listy
  - oszczędność zasobów

# Python: pętla **while**

```
i = 0
```

```
while i < 5:
```

```
    i += 1
```

```
    print('i=' + str(i))
```

```
# i=1
```

```
# i=2
```

```
# i=3
```

```
# i=4
```

```
# i=5
```

- » Konstrukcja podobna jak C/C++
- » Nie ma wariantu **do...while()**
- » Nie ma inkrementacji ale można użyć **+=**, **-=**
- » Sklejanie napisów operatorem **+**
- » wymagaj konwersji do jednego typu dlatego **'i=' + str(i)**

# Python: pętla **while**

```
i = 0
```

```
while i < 5:  
    i += 1  
    print('i=' + str(i))
```

```
# i=1  
# i=2  
# i=3  
# i=4  
# i=5
```

```
while True:  
    i += 1  
    print(i)  
    if i > 100:  
        break
```

- » Konstrukcja podobna jak C/C++
- » Nie ma wariantu **do...while()**
- » Nie ma inkrementacji ale można użyć **+=**, **-=**
- » Sklejanie napisów operatorem **+**
- » wymagaj konwersji do jednego typu dlatego **'i=' + str(i)**
- » Pętla nieskończona
- » Istnieje również konstrukcja **while...else:**

# Programowanie obiektowe w Pythonie

# Python: klasy

```
class MyClass:
    x = 'abc'

    def printX(self):
        x = 10
        print(x)
        print(self.x)

obj = MyClass()
obj.printX()
print(obj.x)
```

» Konstrukcja podobna jak w języku C/C++ (Java, any OOL):

# Python: klasy

```
class MyClass:  
    x = 'abc'  
  
    def printX(self):  
        x = 10  
        print(x)  
        print(self.x)
```

```
obj = MyClass()  
obj.printX()  
print(obj.x)
```

- » Konstrukcja podobna jak w języku C/C++ (Java, any OOL):
  - definicja klasy

# Python: klasy

```
class MyClass:
```

```
    x = 'abc'
```

```
    def printX(self):
```

```
        x = 10
```

```
        print(x)
```

```
        print(self.x)
```

```
obj = MyClass()
```

```
obj.printX()
```

```
print(obj.x)
```

- » Konstrukcja podobna jak w języku C/C++ (Java, any OOL):
  - definicja klasy
  - nazwa klasy

# Python: klasy

```
class MyClass:  
    x = 'abc'  
  
    def printX(self):  
        x = 10  
        print(x)  
        print(self.x)  
  
obj = MyClass()  
obj.printX()  
print(obj.x)
```

- » Konstrukcja podobna jak w języku C/C++ (Java, any OOL):
- definicja klasy
  - nazwa klasy
  - definicja metody



# Python: klasy

```
class MyClass:  
    x = 'abc'  
  
    def printX(self):  
        x = 10  
        print(x)  
        print(self.x)  
  
obj = MyClass()  
obj.printX()  
print(obj.x)
```

- » Konstrukcja podobna jak w języku C/C++ (Java, any OOL):
- definicja klasy
  - nazwa klasy
  - definicja metody
  - zmienna/składowa klasy

# Python: klasy

```
class MyClass:
    x = 'abc'

    def printX(self):
        x = 10
        print(x)
        print(self.x)

obj = MyClass()
obj.printX()
print(obj.x)
```

- » Konstrukcja podobna jak w języku C/C++ (Java, any OOL):
- definicja klasy
  - nazwa klasy
  - definicja metody
  - zmienna/składowa klasy
  - obiekt (instancja) klasy

# Python: klasy

```
class MyClass:  
    x = 'abc'  
  
    def printX(self):  
        x = 10  
        print(x)  
        print(self.x)  
  
obj = MyClass()  
obj.printX()  
print(obj.x)
```

- » Konstrukcja podobna jak w języku C/C++ (Java, any OOL):
- definicja klasy
  - nazwa klasy
  - definicja metody
  - zmienna/składowa klasy
  - obiekt (instancja) klasy
  - tworzenie obiektu

# Python: klasy

```
class MyClass:  
    x = 'abc'  
  
    def printX(self):  
        x = 10  
        print(x)  
        print(self.x)  
  
obj = MyClass()  
obj.printX()  
print(obj.x)
```

- » Konstrukcja podobna jak w języku C/C++ (Java, any OOL):
- definicja klasy
  - nazwa klasy
  - definicja metody
  - zmienna/składowa klasy
  - obiekt (instancja) klasy
  - tworzenie obiektu
  - wywołanie metody na obiekcie

# Python: klasy

```
class MyClass:  
    x = 'abc'  
  
    def printX(self):  
        x = 10  
        print(x)  
        print(self.x)  
  
obj = MyClass()  
obj.printX()  
print(obj.x)
```

- » Konstrukcja podobna jak w języku C/C++ (Java, any OOL):
- definicja klasy
  - nazwa klasy
  - definicja metody
  - zmienna/składowa klasy
  - obiekt (instancja) klasy
  - tworzenie obiektu
  - wywołanie metody na obiekcie
  - dostęp do składowej (zmiennej obiektu)

# Python: klasy

```
class MyClass:
    x = 'abc'

    def printX(self):
        x = 10
        print(x)
        print(self.x)

obj = MyClass()
obj.printX()
print(obj.x)
```

- » Pierwszy argument metody to referencja do obiektu na którym metoda została wykonana
- » Zmienną self można traktować jak **this** w języku C++

# Python: klasy

```
class MyClass:  
    x = 'abc'  
  
    def printX(self):  
        x = 10  
        print(x)  
        print(self.x)  
  
obj = MyClass()  
obj.printX()  
print(obj.x)
```

- » Pierwszy argument metody to referencja do obiektu na którym metoda została wykonana
- » Zmienną self można traktować jak **this** w języku C++
- » Metoda może zawierać **zmienne lokalne** o takiej samej nazwie jak **komponenty klasy**

# Python: klasy

```
class MyClass:
    x = 'abc'

    def printX(self):
        x = 10
        print(x)
        print(self.x)

obj = MyClass()
obj.printX()
print(obj.x)
```

- » Pierwszy argument metody to referencja do obiektu na którym metoda została wykonana
- » Zmienną self można traktować jak **this** w języku C++
- » Metoda może zawierać zmienne lokalne o takiej samej nazwie jak komponenty klasy
- » Python nie obsługuje paradygmatu hermetyzacji, wszystkie komponenty klasy są **publiczne**



# Python: klasy

```
class MyClass:
```

```
    x = 'abc'
```

```
    def printX(self):
```

```
        x = 10
```

```
        print(x)
```

```
        print(self.x)
```

```
obj = MyClass()
```

```
obj.printX()
```

```
print(obj.x)
```

» Rezultat działania programu:

10

abc

abc

# Python: klasy

```
class MyClass:  
    x = 'abc'  
  
    def __init__(self, name):  
        self.x = name  
  
    def printX(self):  
        print(self.x)  
  
obj = MyClass('XXX')  
obj.printX()  
print(obj.x)
```

» **Konstruktor** (inicjalizacja obiektu),  
działa podobnie jak w C++

# Python: klasy

```
class MyClass:
    x = 'abc'

    def __init__(self, name):
        self.x = name

    def printX(self):
        print(self.x)

obj = MyClass('XXX')
obj.printX()
print(obj.x)
```

- » Konstruktor (inicjalizacja obiektu), działa podobnie jak w C++
- » Przekazywanym argumentem jest **name**
  - parametr nie ma typu (w Pythonie wartość ma typ nie zmienna)
  - parametr **self** jest automatycznie dodawany
- » Istnieje również destruktork **\_\_del\_\_(self)**: jest wołany przez garbage collector

# Python: klasy

```
class MyClass:  
    x = 'abc'  
  
    def calc(self, arg):  
        return self.x, arg
```

```
obj = MyClass()  
a, b = obj.calc(3.14)
```

```
print(a)  
print(b)
```

```
q = obj.calc(3.14)  
print(type(q))  
print(q[0])  
print(q[1])
```

- » Funkcja/metoda może zwrócić dowolną liczbę wartości
- » Nie trzeba deklarować co będzie zwracać (zmienna nie ma typu, wartość ma typ)

# Python: klasy

```
class MyClass:  
    x = 'abc'  
  
    def calc(self, arg):  
        return self.x, arg
```

```
obj = MyClass()  
a, b = obj.calc(3.14)
```

```
print(a)  
print(b)
```

```
q = obj.calc(3.14)  
print(type(q))  
print(q[0])  
print(q[1])
```

- » Funkcja/metoda może zwrócić dowolną liczbę wartości
- » Nie trzeba deklarować co będzie zwracać (**zmienna nie ma typu, wartość ma typ**)
- » W przykładzie, technicznie zwracana jest krotka
- » Poszczególne wartości można odebrać **pojedynczo** lub **grupowo** (jako krotkę)
- » Można to zrobić lepiej - np. zwrócić słownik

# Sytuacje wyjątkowe

# Python: wyjątki

```
x = [0, 1, 2, 3]
x[4] = 0
```

- » Przykład **przekroczenia zakresu**
- » C++: problemy
- » Python:
  - Traceback (most recent call last):
  - File "ex30.py", line 7, in
  - <module>
  - x[4]
  - IndexError: list index out of range**
- » Wyjątek (ang. *exception*):
  - gdzie to się stało
  - jaka instrukcja zawiniła
  - co się stało

# Python: wyjątki

```
x = [0, 1, 2, 3]  
x[4] = 0
```

```
Traceback (most recent call last):  
  File "ex30.py", line 7, in <module>  
    x[4]  
IndexError: list index out of range
```

- » Wyjątki to sposób obsługi błędów w czasie wykonywania programu
- » Mechanizm jest stosowany w wielu językach (w tym C++, Java)
- » Jest implementowany na poziomie języka (nie biblioteki)
- » Wyjątek nie jest nieuchronnym końcem programu, można go “obsłużyć”
- » Wyjątek nieobsłużony przechodzi na wyższy poziom wywołania
- » Nigdzie nieobsłużony kończy program



# Python: wyjątki

```
x = [0, 1, 2, 3]  
x[4] = 0
```

```
Traceback (most recent call last):  
  File "ex30.py", line 7, in <module>  
    x[4]  
IndexError: list index out of range
```

- » Przykładowe wyjątki:
  - brak pliku
  - przekroczony zakres listy
  - brak klucza w słowniku
  - niepoprawny format danych
  - problemy w konstruktorze
  - ...
- » Powyższe sytuacje nie są “śmiertelne” dla programu
- » Na wyjątki często przerzuca się walidację danych
- » Wyjątki nie są kosztowne

# Python: wyjątki

```
x = [0, 1, 2, 3]
```

```
try:  
    x[4] = 0  
except IndexError:  
    print('Index error')
```

- » Wyjątek będzie przechwytywany wewnątrz bloku **try**
- » Jeżeli wystąpi wyjątek **IndexError** zostanie on przechwycony i wykonają się instrukcje w bloku **except**
- » Nieprawidłowa operacja **x[4]** nie zostanie wykonana
- » Próba jej wykonania spowodowała zgłoszenie wyjątku

# Python: wyjątki

```
x = [0, 1, 2, 3]
```

```
try:  
    x[4] = 0  
except IndexError:  
    print('Index error')  
except:  
    print('Unknown error')
```

- » Wyjątek będzie przechwytywany wewnątrz bloku **try**
- » Jeżeli wystąpi wyjątek **IndexError** zostanie on przechwycony i wykonają się instrukcje w bloku **except**
- » Nieprawidłowa operacja **x[4]** nie zostanie wykonana
- » Próba jej wykonania spowodowała zgłoszenie wyjątku
- » Można obsłużyć wiele wyjątków z jednego **try**

# Python: wyjątki

```
x = {'length':10, 'height':20}
```

```
try:  
    print(x['length'])  
    print(x['height'])  
    print(x['height'])  
except KeyError:  
    print('invalid key')
```

- » Przykład walidacji danych przez wyjątki
- » Nie sprawdzam czy klucz jest poprawny
- » Indeksuję kluczem i sprawdzam czy się udało
- » **Rezultat:**  
**10**  
**invalid key**
- » Pierwszy wyjątek w bloku kończy jego wykonywanie

# Biblioteki, pakiety, moduły

import ...

# Python: pakiety, moduły

- » Ogromna popularność języka Python wynika między innymi z bardzo bogatych bibliotek
- » Moduł to jest plik.py zawierający definicje klas, metod, etc...
- » Moduły można importować do innych modułów tworząc hierarchię programu
- » Moduły to również sposób na tworzenie przestrzeni nazw i radzenie sobie z problemem kolizji nazw

# Python: moduły

```
# file: mymod.py  
def inc(var):  
    return var+1
```

```
def dec(var):  
    return var-1
```

---

```
# file ex.py  
import mymod  
print(mymod.inc(3))
```

»

W pliku **mymod.py** zdefiniowałem dwie funkcje

»

W pliku **ex.py** zaimportowałem moduł **mymod** i użyłem funkcji

# Python: moduły

```
# file: mymod.py  
def inc(var):  
    return var+1
```

```
def dec(var):  
    return var-1
```

---

```
# file ex.py  
import mymod  
print(mymod.inc(3))
```

» W pliku **mymod.py** zdefiniowałem dwie funkcje

» W pliku **ex.py** zaimportowałem moduł **mymod** i użyłem funkcji

» Wymagany **prefiks**



# Python: moduły

```
# file: mymod.py  
def inc(var):  
    return var+1
```

```
def dec(var):  
    return var-1
```

---

```
# file ex.py  
import mymod  
print(mymod.inc(3))
```

---

```
from mymod import *  
print(inc(3))
```

» W pliku **mymod.py** zdefiniowałem dwie funkcje

» W pliku **ex.py** zaimportowałem moduł **mymod** i użyłem funkcji

» Wymagany prefiks

» Możliwy import do przestrzeni nazw aby **pozbyć się prefiksu**

# Python: moduły

```
# file: mymod.py  
def inc(var):  
    return var+1
```

```
def dec(var):  
    return var-1
```

---

```
# file ex.py  
import mymod  
print(mymod.inc(3))
```

---

```
from mymod import *  
print(inc(3))
```

---

```
from mymod import inc  
print(inc(3))
```

» W pliku **mymod.py** zdefiniowałem dwie funkcje

» W pliku **ex.py** zaimportowałem moduł **mymod** i użyłem funkcji

» Wymagany prefiks

» Możliwy import do przestrzeni nazw aby pozbyć się prefiksu

» Można importować **pojedyncze** definicje:

# Python: moduły

```
# file: mymod.py  
def inc(var):  
    return var+1
```

```
def dec(var):  
    return var-1
```

---

```
# file ex.py  
import mymod  
print(mymod.inc(3))
```

---

```
from mymod import *  
print(inc(3))
```

---

```
from mymod import inc  
print(inc(3))
```

---

```
import mymod as my  
print(my.inc(3))
```

» W pliku **mymod.py** zdefiniowałem dwie funkcje

» W pliku **ex.py** zaimportowałem moduł **mymod** i użyłem funkcji

» Wymagany prefiks

» Możliwy import do przestrzeni nazw aby pozbyć się prefiksu

» Można importować pojedyncze definicje:

» Można **zmienić** przestrzeń nazw (prefix)

# Python: pakiety, moduły

- » Pakiety to wiele modułów umieszczonych w hierarchii podobnej jak system plików
- » Przykłady: <https://docs.python.org/2/tutorial/modules.html>

```
import sound.effects.echo  
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

```
from sound.effects import echo  
echo.echofilter(input, output, delay=0.7, atten=4)
```

```
from sound.effects.echo import echofilter  
echofilter(input, output, delay=0.7, atten=4)
```

# Obsługa plików

# Python: pliki

```
f = open('text.txt', 'r')  
print(f)
```

- » Obsługa podobna jak w C
- » Plik należy otworzyć i podać w jakim trybie będziemy go używać
- » Rezultatem jest obiekt (odpowiednik “file descriptor”)

<\_io.TextIOWrapper  
name='text.txt' mode='r'  
encoding='UTF-8'>

# Python: pliki

```
f = open('text.txt', 'r')  
print(f)
```

- » Obsługa podobna jak w C
- » Plik należy otworzyć i podać w jakim trybie będziemy go używać
- » Rezultatem jest obiekt (odpowiednik “file descriptor”)

Traceback (most recent call last):  
File "ex43.py", line 6, in <module>  
f = open('text.txt', 'r')  
**FileNotFoundError**: [Errno 2] No such file or directory: 'text.txt'

<\_io.TextIOWrapper  
name='text.txt' mode='r'  
encoding='UTF-8'>

- » Jeżeli pliku nie uda się otworzyć, otrzymamy wyjątek

# Python: pliki

```
f = open('text.txt', 'r')  
print(f)
```

```
txt = f.read()  
print(txt)
```

- » Obsługa podobna jak w C
- » Plik należy otworzyć i podać w jakim trybie będziemy go używać
- » Rezultatem jest obiekt (odpowiednik “file descriptor”)
- » Plik można **cały przeczytać**



# Python: pliki

```
f = open('text.txt', 'r')  
print(f)
```

```
txt = f.read()  
print(txt)
```

```
print(f.readline())
```

- » Obsługa podobna jak w C
- » Plik należy otworzyć i podać w jakim trybie będziemy go używać
- » Rezultatem jest obiekt (odpowiednik “file descriptor”)
- » Plik można cały przeczytać
- » Można czytać linia po linii

# Python: pliki

```
f = open('text.txt', 'r')  
print(f)
```

```
txt = f.read()  
print(txt)
```

```
print(f.readline())
```

```
for line in f:  
    print(line)
```

- » Obsługa podobna jak w C
- » Plik należy otworzyć i podać w jakim trybie będziemy go używać
- » Rezultatem jest obiekt (odpowiednik “file descriptor”)
- » Plik można cały przeczytać
- » Można czytać linia po linii
- » Obiekt `f` jest iterowalny więc można go iterować w pętli linia po linii

# Python: pliki

```
f = open('text.txt', 'r')
```

```
for line in f:  
    tokens = line.split(',')  
    for token in tokens:  
        print(token, end="")
```

» Założmy zawartość pliku (csv):

a, b, c

1, 2, 3

4, 5, 6

# Python: pliki

```
f = open('text.txt', 'r')
```

```
for line in f:
```

```
    tokens = line.split(',')
```

```
    for token in tokens:
```

```
        print(token, end="")
```

» Założmy zawartość pliku (csv):

a, b, c

1, 2, 3

4, 5, 6

» Iterowanie po linii

# Python: pliki

```
f = open('text.txt', 'r')
```

```
for line in f:
```

```
    tokens = line.split(',')
```

```
    for token in tokens:
```

```
        print(token, end=" ")
```

» Założmy zawartość pliku (csv):

a, b, c

1, 2, 3

4, 5, 6

» Iterowanie po linii

» **Dzielenie linii** na fragmenty  
odseparowane znakami “,”

# Python: pliki

```
f = open('text.txt', 'r')
```

```
for line in f:
```

```
    tokens = line.split(',')
```

```
    for token in tokens:
```

```
        print(token, end="")
```

» Założmy zawartość pliku (csv):

a, b, c

1, 2, 3

4, 5, 6

» Iterowanie po linii

» Dzielenie linii na fragmenty  
odseparowane znakami “,”

» Rezultat to **lista**

# Python: pliki

```
f = open('text.txt', 'r')
```

```
for line in f:  
    tokens = line.split(',')  
    for token in tokens:  
        print(token, end="")
```

- » Założmy zawartość pliku (csv):  
a, b, c  
1, 2, 3  
4, 5, 6
- » Iterowanie po linii
- » Dzielenie linii na fragmenty odseparowane znakami “,”
- » Rezultat to lista
- » Lista jest iterowalna więc można ją przeglądać element po elemencie

# Python: pliki

```
f = open('text.txt', 'r')
```

```
for line in f:
```

```
    tokens = line.split(',')
```

```
    for token in tokens:
```

```
        print(token, end="")
```

» Założmy zawartość pliku (csv):

a, b, c

1, 2, 3

4, 5, 6

» Iterowanie po linii

» Dzielenie linii na fragmenty  
odseparowane znakami “,”

» Rezultat to lista

» Lista jest iterowalna więc można ją  
przeoglądać element po elemencie

» Zmienna **token** to pojedynczy  
element pliku csv



# Python: pliki

```
f = open('text.txt', 'r')
```

```
for line in f:  
    tokens = line.split(',')  
    for token in tokens:  
        print(token, end="")
```

rezultat:

```
a b c  
1 2 3  
4 5 6
```

- » Założmy zawartość pliku (csv):  
a, b, c  
1, 2, 3  
4, 5, 6
- » Iterowanie po linii
- » Dzielenie linii na fragmenty odseparowane znakami “,”
- » Rezultat to lista
- » Lista jest iterowalna więc można ją przeglądać element po elemencie
- » Zmienna token to pojedynczy element pliku csv

# MiTP - kontynuacja 2019

- » Ostatnie dwa zajęcia z dr inż. Stanisław Soch
  - email: [stoch@agh.edu.pl](mailto:stoch@agh.edu.pl)
  - mobile: 503 667 025
- » Zajęcia odbędą się po 9.04.2019



# Dziękuję