Testowanie i Jakość Oprogramowania

Akademia Tarnowska

Katedra Informatyki

Tomasz Gądek

Wykład 2: Python

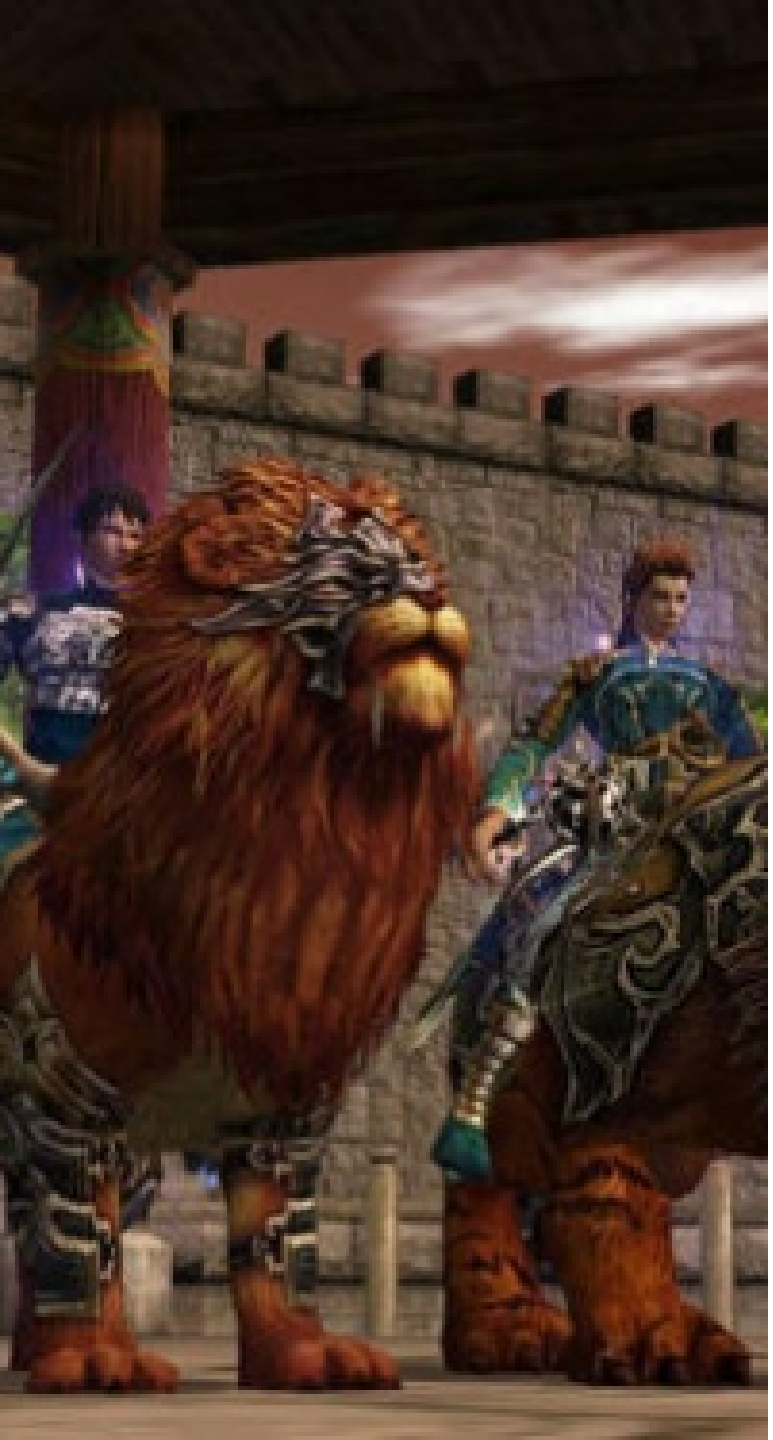
Szczypta historii

- Nazwa języka nie pochodzi od zwierzęcia, lecz od serialu komediowego emitowanego w latach 70 przez **BBC**: *Latający cyrk Monthy Pythona*,
- Twórca języka: *Guido van Rossum*,
- Pierwsze wydanie: **1991**.

Zastosowanie Pythona

Zastosowanie Pythona w firmach technologicznych:

- Google
- Facebook
- Instagram
- NASA



Zastosowanie Pythona

Zastosowanie Pythona w grach:

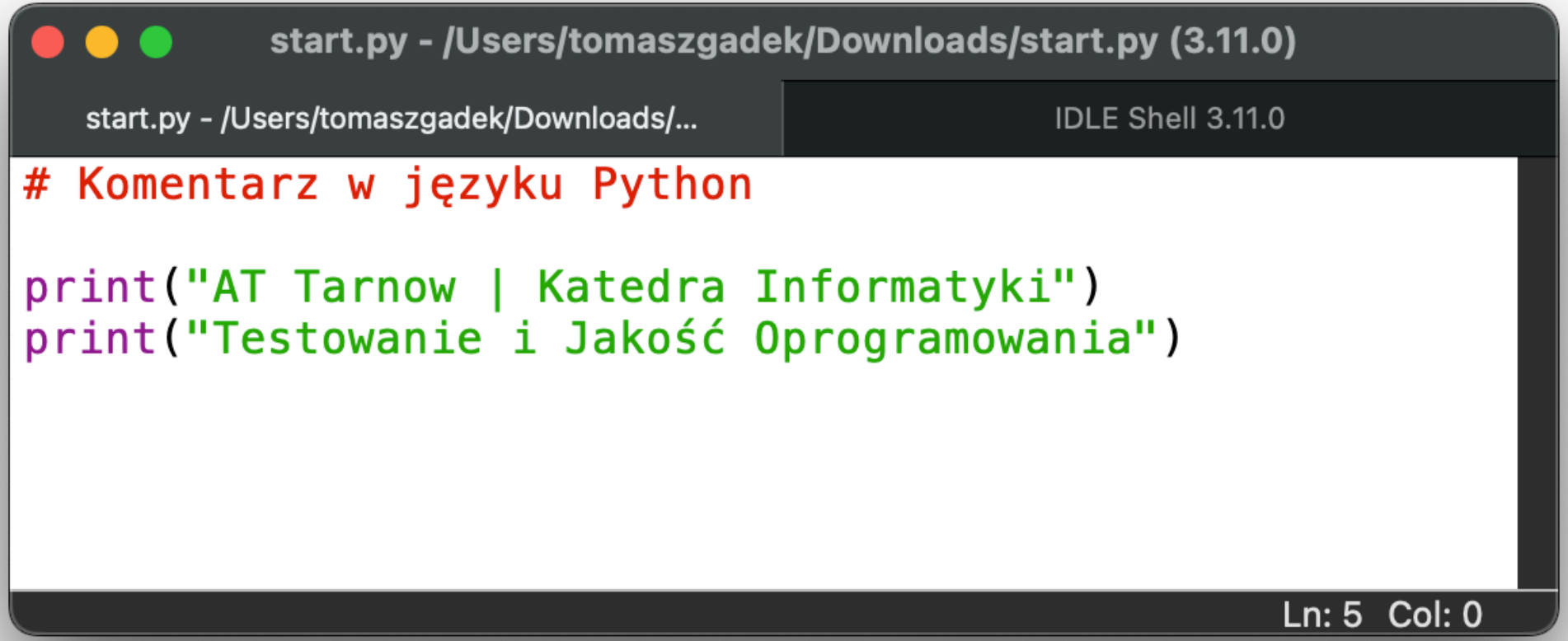
- Battlefield 2
- Metin 2
- Civilisation IV
- World of Tanks

Domyślne IDE dla Pythona

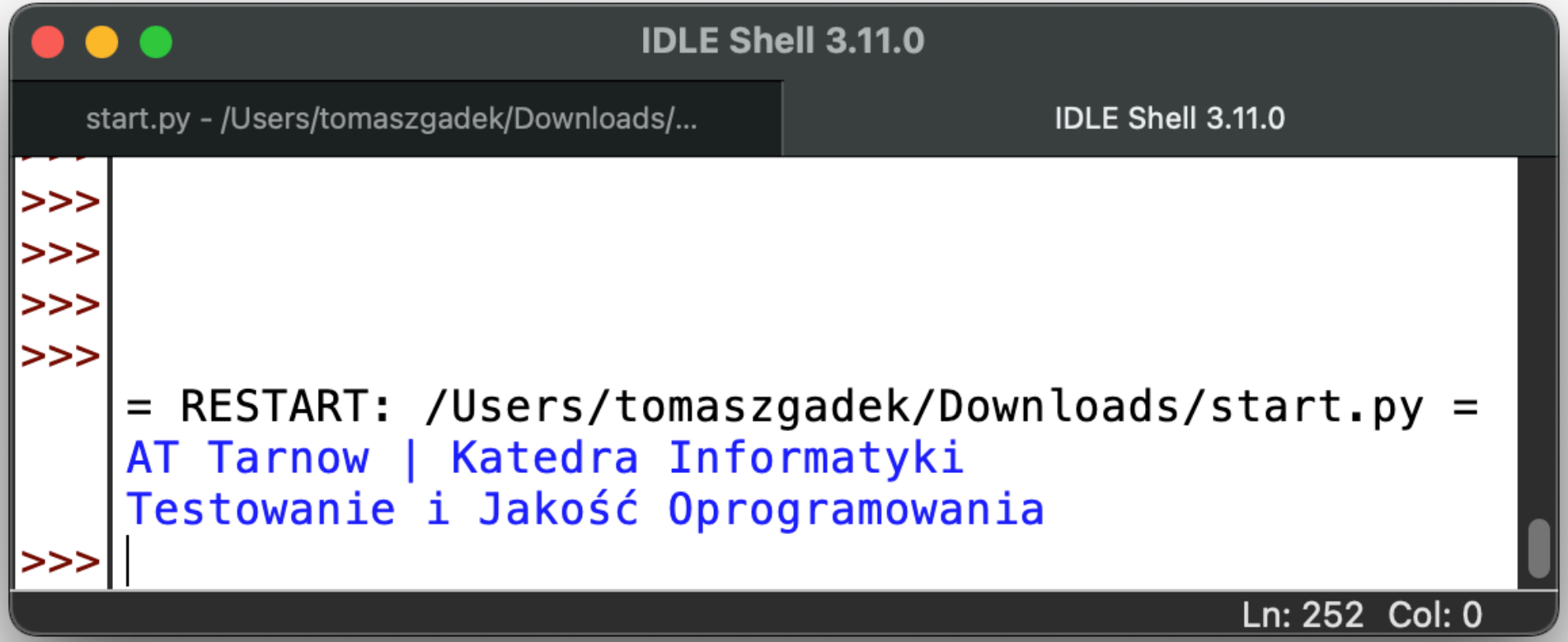
IDLE (Integrated Development and Learning Environment) to domyślne środowisko programistyczne dostarczane razem z Pythonem. Jest to prosty edytor i interpreter Pythona, który umożliwia pisanie, uruchamianie i testowanie kodu w wygodnym interfejsie graficznym.

Uruchomienie skryptu: `Run -> Run Module`

IDLE, edytor


start.py - /Users/tomaszgadek/Downloads/start.py (3.11.0)
start.py - /Users/tomaszgadek/Downloads/...
IDLE Shell 3.11.0
# Komentarz w języku Python
print("AT Tarnow | Katedra Informatyki")
print("Testowanie i Jakość Oprogramowania")
Ln: 5 Col: 0

IDLE, shell


IDLE Shell 3.11.0
start.py - /Users/tomaszgadek/Downloads/...
IDLE Shell 3.11.0
>>>
>>>
>>>
>>>
= RESTART: /Users/tomaszgadek/Downloads/start.py =
AT Tarnow | Katedra Informatyki
Testowanie i Jakość Oprogramowania
>>>
Ln: 252 Col: 0

Funkcja print()

- **\n** - nowa linia
- **\t** - tabulacja

```
print("Fundamenty Python")
print("***")
print("Fundamenty\nPython")
print("***")
print("Fundamenty\tPython")
print("***")
```

Zmienna oraz operator przypisania

Zmienna to **pudełko** na dane. W programowaniu **=** to nie jest znak równości!

```
phone = "iPhone X"

print(phone)
print("Telefon:", phone)
```

Rezultat:

```
iPhone X
Telefon: iPhone X
```

Typy danych oraz type()

```
university_name = "AT Tarnow"
age = 25
height = 1.75
is_bachelor = True # False

print(type(age))
```

Rezultat

```
<class 'int'>
```

Formatowanie tekstu

```
name = "Tomasz"

sentence1 = "Moje imie to {}!".format(name)
sentence2 = f"Moje imie to {name}!"

print(sentence1)
print(sentence2)
```

Rezultat:

```
Moje imie to Tomasz!
Moje imie to Tomasz!
```

Operacje na ciągach tekstowych

```
university_name = "AT Tarnow"

print(university_name[0])    # Pierwszy znak
print(university_name[-1])   # Ostatni znak
print(university_name[:5])   # 5 znaków (od początku)
print(university_name[5:])   # Od znaku o idx = 5 do końca
print(university_name[2:7])  # Znaki od idx = 2 do idx = 6
```

Rezultat:

```
A
w
AT Ta
rnow
 Tarn
```

Operatory arytmetyczne

Obowiązują zasady, jak w klasycznej matematyce.

```
a = 9
b = 4

addition = a + b
subtraction = a - b
multiplication = a * b
division = a / b
integer_division = a // b
remainder_division = a % b
exponentiation = 2 ** 3

print(addition)
print(division)
print(integer_division)
print(b + b * b) # ???
```

Operatory arytmetyczne

Rezultat:

```
13
2.25
2
20
```

Operatory relacji

```
a = 9
b = 4

is_a_greater_than_b = a > b
is_a_greater_or_equal_to_b = a >= b
is_a_less_than_b = a < b
is_a_less_or_equal_to_b = a <= b
is_a_equal_to_b = a == b
is_a_not_equal_to_b = a != b

print(is_a_greater_than_b)
```

Rezultat:

```
True
```

Operatory logiczne

```
name = "Alicja"
age = 20

true_or_false = not(True)
alice_or_not_alice = not(name == "Alicja")
is_adult = (name == "Alicja" and age >= 18)
is_teenager = (name == "Arek" or age < 18)

print(true_or_false)
print(alice_or_not_alice)
print(is_adult)
print(is_teenager)
```

Operatory logiczne

Rezultat:

```
False
False
True
False
```

Instrukcje warunkowe

Zastosowanie **if**:

```
digit = 13

if digit % 2 == 0:
    print("Liczba parzysta")

if digit % 2 != 0:
    print("Liczba nieparzysta")

print("Kolejne instrukcje...")
```

Rezultat:

```
Liczba nieparzysta
Kolejne instrukcje...
```

Instrukcje warunkowe

Zastosowanie **if**, **else**:

```
digit = 13

if digit % 2 == 0:
    print("Liczba parzysta")
else:
    print("Liczba nieparzysta")

print("Kolejne instrukcje...")
```

Rezultat:

```
Liczba nieparzysta
Kolejne instrukcje...
```

Instrukcje warunkowe

Zastosowanie **if**, **elif**, **else**:

```
digit = 13

if digit > 0:
    print("Liczba dodatnia")
elif digit < 0:
    print("Liczba ujemna")
else:
    print("Liczba jest '0'")

print("Kolejne instrukcje...")
```

Rezultat:

```
Liczba dodatnia
Kolejne instrukcje...
```

Dane wejściowe

Odczyt danych wejściowych, funkcja **input()**:

```
name = input("Podaj imie: ")
age = input("Podaj wiek: ")

print(name)
print(age)
```

Dane wejściowe

Chcemy dodać dwie liczby całkowite. Co takiego się wydarzy?

```
a = input("Podaj a: ")
b = input("Podaj b: ")

print(type(a))

add = a + b

print(add)
```

Operacja rzutowania

Rzutowanie poprawi nasze obliczenia.

```
a = int(input("Podaj a: ")) # int()
b = int(input("Podaj b: ")) # int()

print(type(a))

add = a + b

print(add)
```

Pętla programowa for

Jeżeli warunek będzie **prawdziwy** to wykonuj operacje w bloku pętli **for**.

Pętla **for**:

```
university = "AT Tarnow"

for c in university:
    print("Znak:", c)
```

Pętla programowa for

Rezultat:

```
Znak: A
Znak: T
Znak:
Znak: T
Znak: a
Znak: r
Znak: n
Znak: o
Znak: w
```

Pętla programowa while

Pętla **while**:

```
university = "AT Tarnow"

counter = 0
while counter < len(university): # len(university) = 9
    print("Znak: ", university[counter])

    counter = counter + 1
```

Pętla programowa while

```
Znak:  A
Znak:  T
Znak:
Znak:  T
Znak:  a
Znak:  r
Znak:  n
Znak:  o
Znak:  w
```

Funkcja

Funkcja to blok kodu wielokrotnego użytku. Unikamy powtórzeń w kodzie.

Funkcja, która nie zwraca wartości:

```
def print_stars(limit):
    print("Rysuje {} x '*': ".format(limit))

    counter = 0
    while counter < limit:
        print("*", end="")
        counter = counter + 1
    print()

print_stars(3)
print_stars(4)
```

Funkcja

Rezultat:

```
Rysuje 3 x '*':
***
Rysuje 4 x '*':
****
```

Funkcja

Funkcja, która zwraca wartość:

```
def add(a, b):
    sum = a + b
    return sum

print("1 + 3 =", add(1, 3))
print("2 + 4 =", add(2, 4))
```

Rezultat:

```
1 + 3 = 4
2 + 4 = 6
```

Klasa

Obiekty otaczają nasz świat. Do projektowania obiektów służą klasy.

```
class Square:
    def __init__(self, side):
        self.side = side

    def calculate_area(self):
        return self.side * self.side


square = Square(3)  # konstruktor

area = square.calculate_area()

print("Pole kwadratu =", area)
```

Klasa

Rezultat:

```
Pole kwadratu = 9
```

Przechwytywanie wyjątków

```
try:
    1 / 0 # dzielenie przez 0 (wyjatek: ZeroDivisionError)
except ZeroDivisionError:
    print("Wyjatek! Dzielenie przez 0!")
else:
    print("Wyjatek nie wystapil!")
finally:
    print("Tutaj zawsze sie wykona.")
print("Dalsze instrukcje")
```

Rezultat:

```
Wyjatek! Dzielenie przez 0!
Tutaj zawsze sie wykona.
Dalsze instrukcje
```

Własny wyjątek

```
# klasa dziedziczy po Exception
class MyError(Exception):
    def __init__(self, message):
        self.message = message
    def __str__(self):
        return "Wyjatek: " + str(self.message)

# zglaszanie wyjatku
raise MyError("Wystapil wyjatek...")
```

Rezultat:

```
Traceback (most recent call last):
  File "/Users/tgadek/tijo/start.py", line 9, in <module>
    raise MyError("Wystapil wyjatek...")
MyError: Wyjatek: Wystapil wyjatek...
```

Kolekcje / listy (list)

Listy to uporządkowane, mutowalne sekwencje, które mogą przechowywać elementy różnych typów.

```
my_list = [1, 2, 3, 4]
my_list.append(5)      # dodawanie elementu
my_list.remove(2)      # usuwanie elementu
print(my_list[0])      # dostęp do elementu
```

Kolekcje / krotki (tuple)

Krotki to uporządkowane, niemutowalne sekwencje, które są szybsze i zajmują mniej pamięci niż listy.

```
my_tuple = (10, 20, 30)
print(my_tuple[1])   # dostęp do elementu
```

Nie można zmieniać zawartości krotki po jej utworzeniu.

Ciekawostka / zamiana wartości dwóch zmiennych

W Pythonie można zamienić wartości dwóch zmiennych bez użycia zmiennej tymczasowej dzięki zapisowi krotkowemu.

```
a = 5
b = 10

a, b = b, a # zamiana wartości

print(a)    # 10
print(b)    # 5
```

Python tworzy krotkę `(b, a)` *i rozpakowuje ją do* `a, b`.

Kolekcje / zbiory (set)

Zbiory przechowują unikalne, nieuporządkowane elementy i umożliwiają szybkie operacje matematyczne.

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

print(A | B)    # suma: {1, 2, 3, 4, 5, 6}
print(A & B)    # przecięcie: {3, 4}
print(A - B)    # różnica: {1, 2}
print(A ^ B)    # różnica symetryczna: {1, 2, 5, 6}

A.add(7)        # dodawanie elementu
A.remove(2)     # usuwanie elementu
print(3 in A)   # sprawdzenie, czy element należy do zbioru
```

Kolekcje / słowniki (dict)

Słowniki przechowują pary klucz-wartość i pozwalają na szybki dostęp do danych.

```
my_dict = {"imie": "Jan", "wiek": 25}   # tworzenie słownika
print(my_dict["imie"])                  # odczyt wartości
my_dict["miasto"] = "Warszawa"          # dodanie nowej pary
del my_dict["wiek"]                     # usunięcie klucza
```

Funkcja help()

Funkcja help() służy do wyświetlania dokumentacji wbudowanych funkcji, modułów i klas w Pythonie. Aby uzyskać pomoc, wystarczy przekazać nazwę obiektu, np. `help(len)`, aby zobaczyć opis metod dostępnych dla łańcuchów znaków.

```
print(help(len))
```

Funkcja help()

Rezultat:

```
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.

None
```

Dokumentacja

Oficjalna dokumentacja języka Python.

https://docs.python.org/3/

Dlaczego warto korzystać?

- Najnowsze informacje o języku Python.
- Opis wbudowanych funkcji i modułów.
- Przykłady kodu i dobre praktyki.

Koniec