Python

Siódmy wykład

dr inż. Marcin Ciura, Uniwersytet Komisji Edukacji Narodowej

Proszę robić notatki :-)

Przepraszam was za to, że tu miał być dowcip,
a jest o tym, po co nam operator
reszty z dzielenia %

Operator reszty z dzielenia %

Operator % oblicza resztę z dzielenia lewej strony przez prawą:

```
print(10 % 3)
1
print(12 % 4)
0
```

```
if x % 2 == 0:
    print(f'{x} to liczba parzysta')
```

Do czego może się przydać operator reszty z dzielenia %

Reszty z dzielenia liczby a
przez liczbę b tworzą cykl
o długości b

Wyobraź sobie idealny
świat

Do czego może się przydać operator reszty z dzielenia %

W idealnym świecie mieszkają idealni ludzie
Idealni ludzie miesiączkują dokładnie co cztery tygodnie
Aby nie zapomnieć, że zbliża się miesiączka, możemy napisać program. Oto jego fragment:

```
if liczba_dni_od_początku_roku % 28 == x:
    print('Przygotuj się do miesiączki :-)')
```

Plan na dziś

+ Klasy
+ Instrukcja **pass**, instrukcja **break**, instrukcja **continue**
+ Pętla **while**
+ Wyjątki
+ Funkcje jako argumenty funkcji, funkcje anonimowe **lambda**

Klasy

Klasy

Wyobraźmy sobie studenta

Student ma:
+ imię
+ nazwisko
+ zaliczenia z różnych przedmiotów

Student umie:
+ podpisać się
+ dostać zaliczenie

Klasy

Student myśli o sobie: „ja"
Inni myślą o studencie: „Anna"

Inni proszą studenta: „Anna, podpisz się"
Student mógłby siebie prosić: „ja, podpisz się" (gdyby myślał po chińsku :-)

Metody

Na zewnątrz klasy piszemy: anna.podpisz_się()
Wewnątrz klasy moglibyśmy pisać: I.podpisz_się(), ale:

+ I trzeba pisać dużą literą, a to trochę niewygodne
+ Duża litera I jest podobne do małej litery l i cyfry 1

Dlatego wewnątrz klasy piszemy self.podpisz_się() :-)

Metody

Gdy wywołujemy metodę obiektu, piszemy:

+ przed kropką: nazwę tego obiektu, na którym jest wywoływana metoda
+ kropkę .
+ po kropce: nazwę tej metody
+ nie zapominamy o nawiasach okrągłych ( ) — w nich piszemy argumenty tej metody

Atrybuty

Gdy odczytujemy lub zapisujemy atrybut obiektu:

+ przed kropką: nazwę tego obiektu, którego atrybut odczytujemy lub zapisujemy
+ kropkę .
+ po kropce: nazwę tego atrybutu

Atrybuty

```
print(anna.zaliczenia)
{}
anna.zaliczenia['Python'] = 5.0
print(anna.zaliczenia)
{'Python': 5.0}
anna.zaliczenia['biologia'] = 4.5
print(anna.zaliczenia)
{'Python': 5.0, 'biologia': 4.5}
```

Atrybuty

```
anna.zaliczenia['Python'] = 5.0
anna.zaliczenia['biologia'] = 4.5
```

Tak można dostawać zaliczenia, ale jest to niezgodne z zasadą DRY (Don't Repeat Yourself)

Aby nie powtarzać niepotrzebnych fragmentów programu, wymyślono metody

Metody

```
anna.zalicz('Python', 5.0)
anna.zalicz('biologia', 4.5)
```

Oto definicja metody zalicz:

```
def zalicz(self, przedmiot: str, ocena: float):
    self.zaliczenia[przedmiot] = ocena
```

Atrybuty

```
print(anna.imię)
Anna
print(anna.nazwisko)
Nowak
print(f'{anna.imię} {anna.nazwisko}')
Anna Nowak
print(f'{beata.imię} {beata.nazwisko}')
Beata Kowalska
```

Atrybuty

```
print(f'{anna.imię} {anna.nazwisko}')
print(f'{beata.imię} {beata.nazwisko}')
```

Tak można się podpisywać, ale jest to niewygodne
i niezgodne z zasadą DRY

Metody

```
anna.podpisz_się()
beata.podpisz_się()
```

Oto definicja metody podpisz_się:

```
def podpisz_się(self):
    print(f'{self.imię} {self.nazwisko}')
```

Metody

Wiemy już wszystko o metodach, oprócz tego:

Jaką metodą tworzymy obiekty?

Metody

Obiekty tworzymy metodą _ _init_ _

Często mówimy o metodzie _ _init_ _ konstruktor

Metoda _ _init_ _ tworzy obiekt i przypisuje jego atrybutom wartości początkowe

Metody

Oto definicja konstruktora:

```
def __init__(self, imię: str, nazwisko: str):
    self.imię = imię
    self.nazwisko = nazwisko
    self.zaliczenia = {}
```

Klasy

Tak definiujemy klasę:

+ słowo kluczowe **class**
+ nazwa klasy, tradycyjnie pisana z dużej litery
+ dwukropek :
+ po wcięciu: definicje metod tej klasy

Klasy

```
class Student:
    """Model studenta lub studentki :-)"""
    def __init__(self, imię: str, nazwisko: str):
        self.imię = imię
        self.nazwisko = nazwisko
        self.zaliczenia = {}
    def podpisz_się(self):
        print(f'{self.imię} {self.nazwisko}')
    def zalicz(self, przedmiot: str, ocena: float):
        self.zaliczenia[przedmiot] = ocena
```

Klasy

Tak tworzymy obiekt klasy Student:

```
st = Student('Anna', 'Nowak')
```

Tak korzystamy z metod tego obiektu:

```
st.podpisz_się()
st.zalicz('Python', 5.0)
```

Metoda _ _str_ _

Oprócz metody _ _init_ _ można definiować inne specjalne metody. Ich nazwy zaczynają się i kończą dwoma podkreśleniami _ _

Metoda _ _str_ _ jest wywoływana wtedy, kiedy obiekt zamienia się na łańcuch, czyli w praktyce: kiedy obiekt jest wypisywany

Metoda __str__

```
class Student:
    def __init__(self, imię: str, nazwisko: str):
        ...
    def podpisz_się(self):
        print(f'{self.imię} {self.nazwisko}')
    def zalicz(self, przedmiot: str, ocena: float):
        self.zaliczenia[przedmiot] = ocena
    def __str__(self) -> str:
        return f'{self.imię} {self.nazwisko} {self.zaliczenia}'
```

Metoda __str__

```
st = Student('Jan', 'Kowalski')
st.zalicz('WF', 5.0)
st.zalicz('matematyka', 4.0)
print(st)
```

```
Jan Kowalski {'WF': 5.0, 'matematyka': 4.0}
```

Klasy — podsumowanie

```
class Klasa:
    """Co robi Klasa?"""
    def __init__(self, arg1: typ1, arg2: typ2): ...
    def metoda1(self): ...
    def metoda2(self, arg1: typ1): -> typ ...
    def __str__(self) -> str: ...
    ...
```

Proszę mi zadać pytanie :-)

Postęp wykładu

+ Klasy
+ Instrukcja **pass**, instrukcja **break**, instrukcja **continue**
+ Pętla **while**
+ Wyjątki
+ Funkcje jako argumenty funkcji, funkcje anonimowe **lambda**

Instrukcja pass

Instrukcja pass

Instrukcja pass nic nie robi :-)

Po co nam instukcja pass? Na przykład do pętli opóźniającej:

```
for i in range(1_000_000):
    pass
```

Instrukcja **break**

Instrukcja **break**

Instrukcja **break** przerywa działanie pętli

Przykład — szukanie pierwszego wystąpienia danego elementu w liście:

```
lista = ['jabłko', 'banan', 'pomarańcza', 'gruszka']
szukany = 'pomarańcza'
for i, element in enumerate(lista):
    if element == szukany:
        print(f'Znaleziono element {szukany} pod indeksem {i}')
        break
```

Instrukcja continue

Instrukcja **continue**

Instrukcja **continue** pomija resztę kodu w bieżącym obiegu pętli
i przechodzi do kolejnego obiegu pętli

Przykład — pomijanie elementów w liście:

```
adresy_email = ['test@gmail.com', 'niepoprawny-email', 'drugi.email@wp.pl']
for email in adresy_email:
    if '@' not in email:
        print(f"Adres '{email}' jest nieprawidłowy, pomijam.")
        continue
    print(f"Przetwarzam adres: '{email}'")
    ...
```

Instrukcja **continue**

Gdyby nie instrukcja **continue**, trzeba by było użyć instrukcji **else**.
Zbyt duże wcięcia są niewygodne

```
adresy_email = ['test@gmail.com', 'niepoprawny-email', 'drugi@wp.pl']
for email in adresy_email:
    if '@' not in email:
        print(f"Adres '{email}' jest nieprawidłowy, pomijam.")
    else:
        print(f"Przetwarzam adres: '{email}'")
        ...
```

Pętle **while**

Pętla **while**

Pętla **while** warunek: powtarza blok instrukcji tak długo, jak długo warunek jest prawdziwy

Przykład 1: pętla nieskończona

```
while True:
    pass
```

Pętla **while**

Przykład 2 — wczytywanie danych:

```
while True:
    dane = input('Podaj dane> ')
    if not dane:  # To samo, co if dane == '':
        break
    przetwórz(dane)
```

Pętla **while**

Przykład 3 — wczytywanie danych:

```
while True:
    dane = input('Podaj dane> ')
    if not są_poprawne(dane):
        print('Dane są niepoprawne. Popraw je.')
        continue
    przetwórz(dane)
```

Proszę mi zadać pytanie :-)

Postęp wykładu

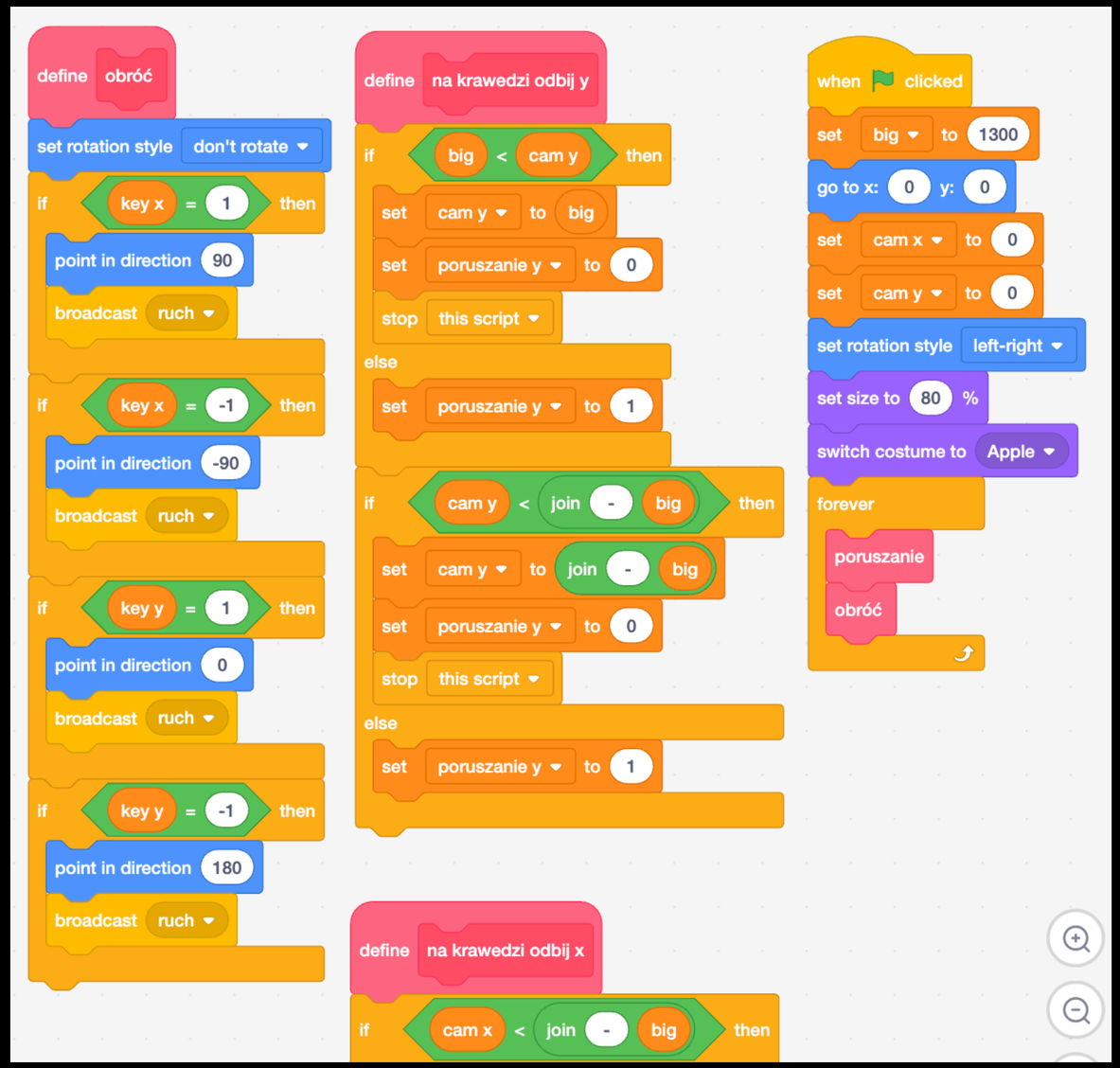
+ Klasy
+ Instrukcja **pass**, instrukcja **break**, instrukcja **continue**
+ Pętla **while**
+ Wyjątki
+ Funkcje jako argumenty funkcji, funkcje anonimowe **lambda**

Wyjątki

Wyjątki

Tak wygląda obrazkowy język Scratch, w którym programuje mój syn

Scratch ma kilka wad i jedną wielką zaletę: nigdy nie zgłasza błędów, żeby dzieci się nie stresowały



Wyjątki

Python też jest przyjazny, ale inaczej. Czasem zgłasza wyjątki.
Przyczyną wyjątku może być błąd w programie lub błędne dane

```
if :
SyntaxError: invalid syntax
x = 7/0
ZeroDivisionError: division by zero
print([3, 4, 7][999])
IndexError: list index out of range
```

Wyjątki

Wyjątków nie trzeba się bać
Na wyjątki nie trzeba się irytować…

…chociaż może to być trudne, jeśli nasz program działa od piętnastu minut, i już ma wypisać wyniki, ale wystąpił w nim nieprzewidziany wyjątek :-(

Po co nam wyjątki

Scratch służy do zabawy

W Scratchu też mogą się zdarzyć sytuacje wyjątkowe, na przykład próba pobrania takiego elementu listy, który nie istnieje
Twórcy Scratcha zaprogramowali obsługę każdego typu wyjątku. Każdy typ wyjątku ma swój, zawsze ten sam, przyjazny dla użytkownika sposób obsługi

Po co nam wyjątki

W Pythonie można pisać całkiem spore programy
Przyczynami wyjątków mogą być:
  + błędy w programie
  + lub niepoprawne dane wejściowe
I temu, i temu trudno zapobiec w każdym dużym programie

Dzięki wyjątkom programiści mają wybór. Mogą:
  + nie przechwytywać wyjątków, niech kończą działanie programu
  + albo przechwycić i obsłużyć poszczególne typy wyjątków

Wyjątki

Programiści mówią: „program rzucił wyjątek”

Jeśli się nie złapie/przechwyci wyjątku, to wyjątek wyskoczy
(na ekran :-)

Wyjątki

Możemy przechwytywać wyjątki dzięki blokom **try**...**except**
Po **except** podajemy typ wyjątku. Jeśli w bloku **try** zostanie rzucony wyjątek, Python pomija pozostałe instrukcje bloku **try** i przechodzi do bloku **except**

```
try:
    with open('plik.txt') as plik:
        # Działania na pliku
except FileNotFoundError:
    print("Nie ma pliku 'plik.txt'")
```

Wyjątki

Możemy pisać wiele bloków **except**, żeby łapać różne typy wyjątków:

```
try:
    with open('plik.txt') as plik:
        # Działania na pliku
except FileNotFoundError:
    print("Nie ma pliku 'plik.txt'")
except IndexError as e:
    print(f"Błąd przetwarzania pliku 'plik.txt': {e}")
```

Wyjątki

Dzięki ogólnemu typowi Exception można złapać dowolny wyjątek. Tak możemy zignorować wszystkie typy wyjątków:

```
try:
    # Różne działania
except Exception:
    pass
```

Wyjątki

Świadomie użyłem wyrażenia typ wyjątku

IndexError to nazwa klasy
Exception to też nazwa klasy

Klasa IndexError dziedziczy po klasie Exception, ale dziś nie będziemy mówić o dziedziczeniu. Dziedziczenie to temat na inny przedmiot: Programowanie obiektowe

Wyjątki

Możemy sami decydować o tym, kiedy zgłaszać wyjątki, dzięki słowu kluczowemu raise

```
if wiek < 0:
    raise ValueError('Wiek nie może być ujemny')
```

Wyjątki — podsumowanie

```
try:
    ...
except TypWyjątku as e:
    ...
except InnyTypWyjątku as e:
    ...
raise TypWyjątku('Dokładniejszy komunikat')
```

Proszę mi zadać pytanie :-)

Postęp wykładu

+ Klasy
+ Instrukcja **pass**, instrukcja **break**, instrukcja **continue**
+ Pętla **while**
+ Wyjątki
+ Funkcje jako argumenty funkcji, funkcje anonimowe **lambda**

Funkcje jako argumenty funkcji

Funkcje jako argumenty funkcji

```
class Student:
    """Przechowuje dane o studencie"""
    def __init__(self, imię: str, nazwisko: str, indeks: int):
        self.imię = imię
        self.nazwisko = nazwisko
        self.indeks = indeks
        self.zaliczenia = {}
    def __str__(self) -> str: ...
```

Funkcje jako argumenty funkcji

```
studenci = [
    Student('Jan', 'Kowalski', 123),
    Student('Anna', 'Nowak', 987),
    Student('Żaneta', 'Kowalska', 234)]

print(sorted(studenci))
```

```
TypeError: '<' not supported between instances of
'Student' and 'Student'
```

Funkcje jako argumenty funkcji

Funkcja **sorted** ma oprócz reverse jeszcze jeden opcjonalny argument key. Argument key to funkcja, która zwraca klucz sortowania. Proszę pamiętać, że podajemy ją bez nawiasów

```
def nazwisko_studenta(s: Student) -> str:
    return s.nazwisko

print(sorted(studenci, key=nazwisko_studenta))
```

Żaneta Kowalska 234 {} Jan Kowalski 123 {} Anna Nowak 987 {}

Funkcje jako argumenty funkcji

Dzięki argumentowi key możemy różnie sortować tę samą listę

```
def indeks_studenta(s: Student) -> int:
    return s.indeks

print(sorted(studenci, key=indeks_studenta))
```

Jan Kowalski 123 {} Żaneta Kowalska 234 {} Anna Nowak 987 {}

Funkcje anonimowe **lambda**

Funkcje anonimowe **lambda**

Niektóre funkcje, zwłaszcza funkcje – argumenty funkcji, są tak proste, że mieszczą się w jednej linii programu

Aby ułatwić pracę programistom, twórcy Pythona wprowadzili do niego funkcje anonimowe

Funkcje anonimowe **lambda**

Tak się definiuje funkcje anonimowe:

lambda lista argumentów: to, co zwraca funkcja

```
lambda x: x**2
lambda a, b: a + b
lambda x: x % 2
```

Po co nam funkcje anonimowe

Funkcje anonimowe przydają się jako funkcje – argumenty funkcji
Zamiast

```
def imię_studenta(s: Student) -> int:
    return s.imię

print(sorted(studenci, key=imię_studenta))
```

możemy napisać:

```
print(sorted(studenci, key=lambda s: s.imię))
```

Anna Nowak 987 {} Jan Kowalski 123 {} Żaneta Kowalska 234 {}

Postęp wykładu

+ Klasy
+ Instrukcja **pass**, instrukcja **break**, instrukcja **continue**
+ Pętla **while**
+ Wyjątki
+ Funkcje jako argumenty funkcji, funkcje anonimowe **lambda**

To wszystko na dziś
Proszę mi zadać pytanie :-)

Czy ktoś z was czegokolwiek
nie zrozumiał z tego wykładu?

Do zobaczenia na ostatnim wykładzie