



UNIWERSYTET
EKONOMICZNY
W POZNANIU



Programowanie obiektowe

dr hab. inż. Adam Wójtowicz, prof. UEP



UNIWERSYTET
EKONOMICZNY
W POZNANIU



dr hab. inż. Adam Wójtowicz, prof. UEP
Katedra Technologii Informacyjnych

https://www.kti.ue.poznan.pl/adam_wojtowicz.html
awojtow@kti.ue.poznan.pl

1.8 CEUE

Godziny dyżurów:
<https://www.kti.ue.poznan.pl/dyzury>



UNIWERSYTET
EKONOMICZNY
W POZNANIU



Wprowadzenie

Programowanie obiektowe

Programowanie

Co to
takiego?



Wytwarzanie oprogramowania

- Analiza i specyfikacja wymagań
- Specyfikacja oprogramowania – model systemu
- Weryfikacja modelu
- Architektura systemu – podział na moduły
- Algorytmizacja modułów
- Implementacja – programowanie, pisanie kodu programu
- Dokumentacja kodu źródłowego
- Testowanie modułów
- Integracja i testowanie systemu
- Szkolenie użytkowników
- Pielęgnacja systemu
- Ewolucja systemu
- Likwidacja systemu (migracja)



Programowanie

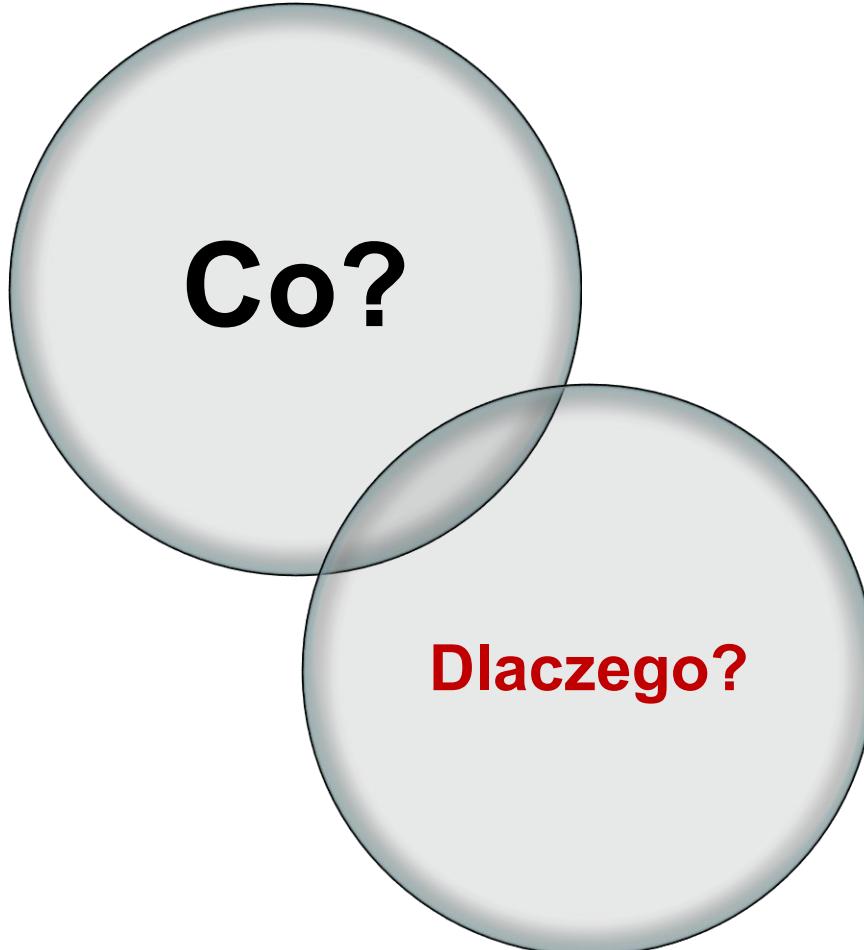
- Analiza i specyfikacja wymagań
- Specyfikacja oprogramowania – model systemu
- Weryfikacja modelu
- **Architektura systemu – podział na moduły**
- **Algorytmizacja modułów**
- **Implementacja – programowanie, pisanie kodu źródłowego**
- **Dokumentacja kodu źródłowego**
- **Testowanie modułów**
- Integracja i testowanie systemu
- Szkolenie użytkowników
- Pielęgnacja systemu
- Ewolucja systemu
- Likwidacja systemu (migracja)



Programowanie

- Kluczowa rola w:
 - wytwarzaniu oprogramowania (ang. software development),
 - inżynierii oprogramowania (ang. software engineering)
- Ważne także w:
 - analityce biznesowej, naukach o danych (ang. data science), uczeniu maszynowym (ML)
 - wielu naukach ścisłych, przyrodniczych, inżynierijnych i technicznych

Programowanie



Co?

Dlaczego?



Programowanie

- Zapotrzebowanie na programistów
 - Polska, UE, świat
- Zapotrzebowanie na menedżerów projektów IT
 - znających specyfikę pracy programistycznych
 - Project Manager, Technical Project Manager,
 - kodujący menedżer,
 - ale też tester, wdrożeniowiec, szkoleniowiec, administrator, etc.
- Zapotrzebowanie na analityków danych
- „Klasa kreatywna”

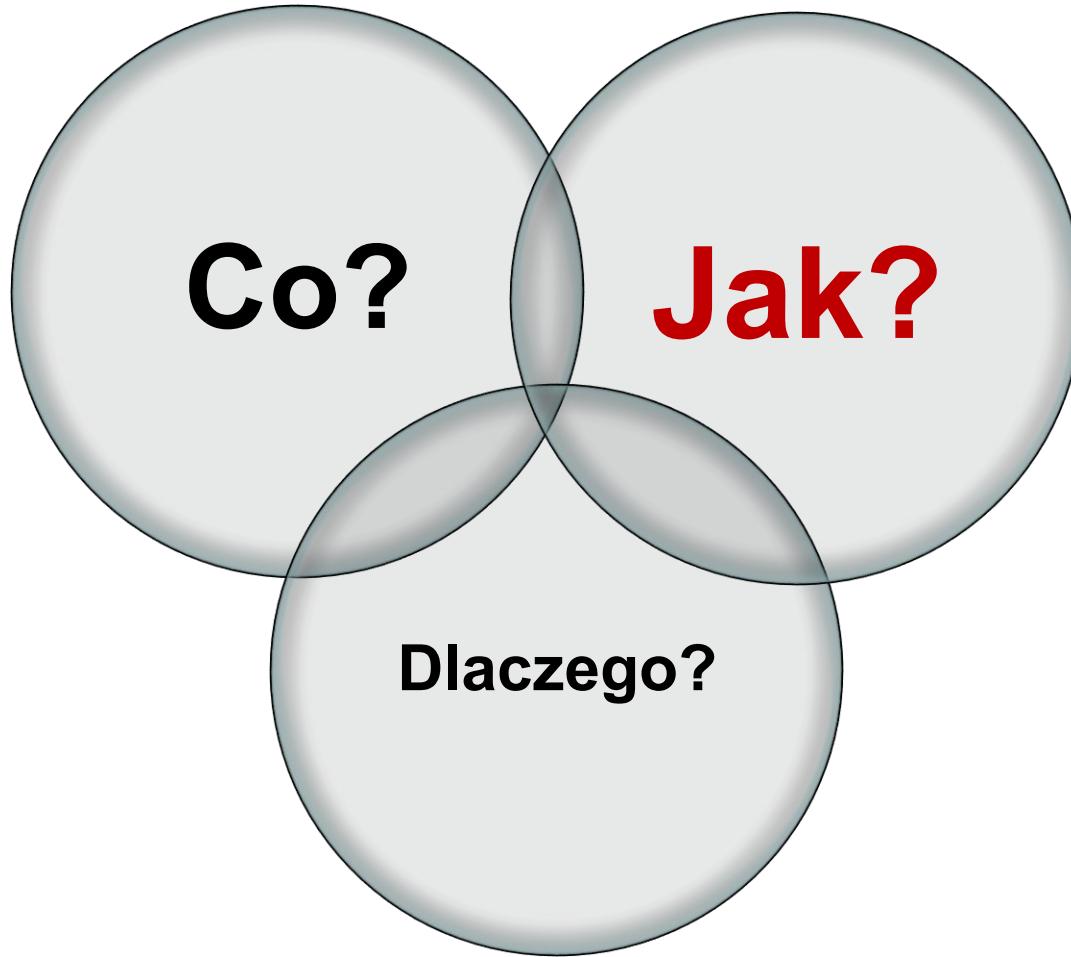
Programowanie

Rozwiązywanie problemów

- Uniwersalne kompetencje niezbędne w zarządzaniu, biznesie, planowaniu, etc.
- Masz problem?
 - rozłoż go **na czynniki pierwsze**
 - **przeanalizuj możliwe rozwiązania i wybierz najlepsze**
 - **wymień kroki, w których go rozwiążesz**
- Myślenie algorytmiczne
 - konfrontacja z problemem, **poszukiwanie rozwiązań**



Programowanie



Cele przedmiotu

- Wyjaśnienie ogólnych pojęć dot. programowania
 - Przedstawienie podstaw programowania w języku Python
 - instrukcje, konstrukcje języka, składnia, struktury danych...
 - Wprowadzenie podstaw programowania obiektowego
 - Nauczenie myślenia algorytmicznego
 - Wyrobienie dobrych nawyków
-
- **Nauczenie podstaw programowania**



Formuła przedmiotu

- Wykłady, ale...
- Wiedza i umiejętności
- Równolegle przedmiot *Programowanie i przetwarzanie danych*
 - Python w praktyce
- Python podstawą innych technologicznych przedmiotów w kolejnych semestrach
- Zaliczenie na ostatnich zajęciach



Plan wykładów

- Wprowadzenie i podstawy języka
- Składnia, operatory, instrukcje, typy danych
- Funkcje, struktury danych
- Klasy, obiekty, dziedziczenie
- Obsługa sytuacji wyjątkowych
- Programowanie wejścia / wyjścia
- Wyrażenia regularne





UNIWERSYTET
EKONOMICZNY
W POZNANIU



Wprowadzenie

Programowanie obiektowe

Programowanie obiektowe

JĘZYKI PROGRAMOWANIA



UNIWERSYTET
EKONOMICZNY
W POZNANIU

Składnia i semantyka

- Składnia
 - Reguły dot. struktury kodu w danym języku
 - Przykład: w Pythonie po nazwie zmiennej może wystąpić znak = a następnie pewna wartość
- Semantyka
 - Znaczenie konstrukcji językowych w danym języku
 - Przykład: w Pythonie licznik=10 znaczy „jeżeli licznik jest nowym identyfikatorem zmiennej to zarezerwuj na nią pamięć; następnie przypisz wartość 10 do zmiennej licznik”
- Typy danych



Klasyfikacja języków



Kompilacja vs. interpretacja

- **Kompilacja**
 - transformacja kodu źródłowego do kodu maszynowego
 - program specyficzny dla danego sprzętu i systemu operacyjnego
 - języki komplowane, np. C, C++
- **Interpretacja**
 - wykonanie kodu źródłowego
 - niezależne od sprzętu i systemu operacyjnego
 - języki interpretowane, np. JavaScript, **Python**
- **Podejście hybrydowe**
 - transformacja kodu źródłowego do *byte code*
 - interpretacja *byte code* na maszynie wirtualnej
 - maszyna wirtualna może być instalowana na różnych platformach sprzętowo-systemowych
 - np. Java, C#



Wybór języka programowania

- Pytanie: Jaki jest najlepszy język programowania? 😊
- Odpowiedź: To zależy.
- **Jaki problem?** np. JavaScript – Web; Java – enterprise systems; C/C++ - IoT, sprzęt; Python, R – obliczenia, analityka, machine learning, Web; Java, Scala, Go – cloud; SQL – bazy danych
- **Jaka wydajność?** np. C++ – wymagana szybkość grafiki; Go, Erlang – wymagana skalowalność systemów czasu rzeczywistego
- **Jaka platforma?** np. C# – Windows; Java – Android; Swift – iOS
- Czy **łatwość utrzymania** jest ważna? – Java
- Czy **łatwość uczenia się** jest ważna? – Python
- Jaki „**time to market**”? – Python, Ruby
- Jakie **biblioteki**? – Python, ...
- Czy **przenośność, wieloplatformowość**? – Python, Java



Python – zalety

- Dobra nadaje się do analizy danych, uczenia maszynowego
- Bardzo wiele użytecznych bibliotek
- Szybkie prototypowanie
- Prosty w nauce
- Ma również zastosowania w obszarze aplikacjach webowych (po stronie serwera)
- Open source, duża żywa społeczność
- Przenośność kodu między platformami



Python – ograniczenia

- Problemy z wydajnością i zużyciem pamięci
- Duże projekty trudne w utrzymaniu, język interpretowany, dynamicznie typowany
 - runtime errors
- Problemy z wielowątkowością
- Brak natywnego wsparcia w środowiskach mobilnych



Interpreter Python

- Do pobrania z:
<https://www.python.org/downloads/>
- Interpreter, ale nie tylko
- Domyślne IDE o nazwie IDLE
- Dokumentacja
- pip – package installer for Python
- tkinter – narzędzia do GUI
- narzędzia do testowania



Alternatywne IDE

- IDLE vs. PyCharm
- Do prostych programów i nauki podstaw IDLE wystarczy
- Do bardziej zaawansowanego wytwarzania oprogramowania - PyCharm
 - lepszy debugger i testy kodu
 - analiza kodu wspomagająca programistę
 - refaktoryzacja kodu



Python – przykład programu

```
import statistics  
data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]  
print(statistics.mean(data))  
print(statistics.median(data))  
print(statistics.variance(data))
```



Programowanie obiektowe

ZMIENNE



UNIwersytet
EKONOMICZNY
W POZNANIU

Definicja zmiennej

Zmienna jest przestrzenią w pamięci oznaczoną symbolem o ustalonej **nazwie**, której może być przypisana **wartość**



Nazwa zmiennej w języku Python

- Inaczej: **identyfikator zmiennej**
- Ciąg znaków
 - Litery A-Z, a-z, _
 - Cyfry 0-9
- Pierwszy element ciągu: litera A-Z, a-z, _
- Wielkość liter ma znaczenie
- Uwaga – niektóre nazwy są zabronione



Python – słowa zarezerwowane

and
assert
break
class
continue
def
del
elif
else
except

exec
finally
for
from
global
if
import
in
is
lambda

not
or
pass
print
raise
return
try
while
with
yield



Zmienne w języku Python

- **Nie wymagają jawnej deklaracji z podaniem typu**
- Deklaracja odbywa się automatycznie w momencie przypisania wartości do zmiennej za pomocą operatora przypisania =
 - operator przypisania = zawsze przypisuje to, co jest po jego prawej stronie do tego co po lewej stronie
- **Dynamiczne typowanie zmiennych** – przypisywana wartość przesąduje o typie zmiennej

```
licznik = 10
```

```
kwota = 9.99
```

```
nazwisko = "Iksinski"
```



Przykłady nazw zmiennych

- Poprawne

mojaZmienna

Moja2gaZmienna

_prywatnaZmienna

___scislePrywatnaZmienna

- Błędne

1Zmienna

vice-versa



Programowanie obiektowe

WYRAŻENIA, INSTRUKCJE I BLOKI



UNIwersytet
EKONOMICZNY
W POZNANIU

Definicja wyrażenia

Wyrażenie jest ciągiem wartości, zmiennych, operatorów i wywołań funkcji, którego ewaluacja (wynik obliczeń) jest **wartością pewnego typu**



Wyrażenia w języku Python

5

licznik

licznik + wartoscMinimalna

3 * (pi + 5)

cenaNetto + obliczPodatek(cenaNetto)

czyKoniec == True

(kategoria > 3) and (not czyKoniec)



Definicja instrukcji

Instrukcja jest
częścią programu,
która można wykonać



Instrukcje w języku Python

- Najczęściej odpowiada jednej linii kodu
- Przykłady

```
licznik = 0
print(licznik)
break
wartosc = 10 + \
           10 + \
               5
```



Definicja bloku

Blok jest ciągiem jednej lub więcej instrukcji wyznaczonym przez takie samo wcięcie linii



Bloki w języku Python

- Brak nawiasów wyznaczających bloki kodu
- Wszystkie instrukcje w bloku muszą być wcięte tą samą liczbą wcięć (dowolną)
- Wcięcia tworzy się tabulatorami lub spacjami
- Interpreter analizuje wcięcia i przerywa wykonanie programu w przypadku błędnych wcięć
- Oczywiście nie każdy błąd zostanie wykryty

```
if False:  
    print(1)  
    print(2)
```

```
if False:  
    print(1)  
    print(2)
```



Definicja komentarza

Komentarz jest ciągiem znaków
który nie jest instrukcją i
nie jest wykonywany



Przykład komentarzy w języku Python

- Komentarz może obejmować
 - pojedynczy wiersz poprzedzony znakiem **#**

```
# to jest wiersz komentarza  
zmienna = 5 #to też ale tylko od krzyżyka  
print(5)
```

- wiele wierszy kodu ograniczonych trzema znakami **'''**

```
''' to jest blok komentarza  
licznik = 10  
nowaWartosc = nowaWartosc + 1  
'''
```



Przykładowy program

```
zmienna = 1 #przypisanie wartości
print(zmienna)
zmienna = zmienna + 1 #przypisanie nowej wartości
print(zmienna)
zmienna = 1.0 #przypisanie wartości innego typu
                #do tej samej zmiennej
print(zmienna)
zmienna = "tekst" #jak wyżej
print(zmienna)
```



Programowanie obiektowe - stacj. II st. - (wyk.) - prof. Adam Wójtowicz

GU3FB2EZ

KLUCZ DO KURSU NA MOODLE



UNIwersytet
EKONOMICZNY
W POZNANIU



UNIWERSYTET
EKONOMICZNY
W POZNANIU



Dziękuję!

Do zobaczenia za tydzień



UNIWERSYTET
EKONOMICZNY
W POZNANIU



Typy danych i operatory

Programowanie obiektowe

dr hab. inż. Adam Wójtowicz, prof. UEP

Typy danych

- **Typ logiczny**
 - bool
- **Typy liczbowe**
 - int, float, complex
- **Typ tekstowy**
 - str
- **Typy binarne**
 - bytes, bytearray
- Typy sekwencji
 - list, tuple, range
- Typy zbiorów
 - set, frozenset
- Typ słownikowy
 - dict



Operatory

- **Operatory logiczne**
 - iloczyn, suma, negacja
- **Operatory porównania**
 - równe, różne, większe, większe lub równe, ...
- **Operatory arytmetyczne**
 - dodawanie, odejmowanie, dzielenie, ...
- **Operatory bitowe**
 - iloczyn, suma, przesunięcie, ...
- **Operatory przypisania**
 - przypisanie, przypisanie z dodawaniem, ...
- Operatory identyczności
 - tożsamość, brak tożsamości
- Operatory zawierania
 - zawieranie, brak zawierania



Programowanie obiektowe

TYP LOGICZNY OPERATORY LOGICZNE I PORÓWNANIA



UNIwersytet
EKONOMICZNY
W POZNANIU

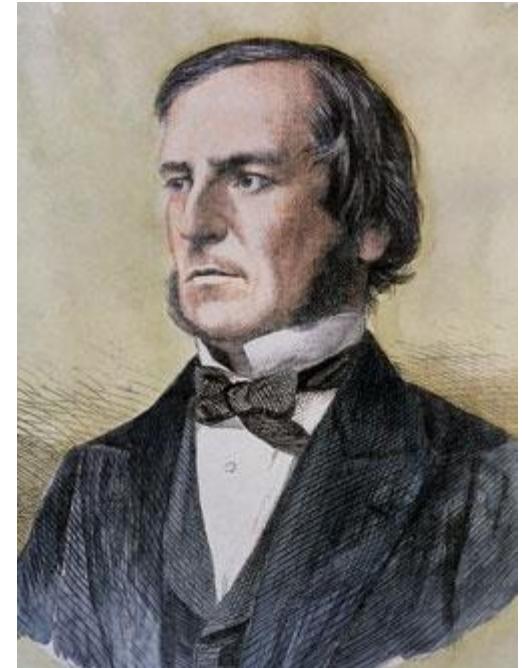
Algebra Boole'a

- George Boole
 - 1815 – 1864

- Dwie wartości

0	fałsz	False
1	prawda	True

- Pojęcie *bitu*
- W języku Python, typ **bool**



Operatory logiczne

p	q	$p \text{ and } q$	$p \text{ or } q$	$\text{not } p$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Operator logiczny XOR

	Fałsz (0)	Prawda (1)
Fałsz (0)	0	1
Prawda (1)	1	0



Operatory logiczne

- Operatory logiczne w języku Python
 - **and** iloczyn
drugi operand jest ewaluowany tylko gdy pierwszy jest True
 - **or** suma
drugi operand jest ewaluowany tylko gdy pierwszy jest False
 - **not** negacja
niższy priorytet niż operatory nie-logiczne, więc:
 $\text{not } a == b \Leftrightarrow \text{not } (a == b)$
 $a == \text{not } b$ BŁĄD
- Pochodne operatory logiczne implementujemy
 - XOR (alternatywa rozłączna) jako np.:
 $(\text{not } A \text{ and } B) \text{ or } (A \text{ and } \text{not } B)$
`bool(A) != bool(B)`
`bool(A) ^ bool(B)`



Inne wartości ewaluowane jako logiczne

- W języku Python wartość `False` posiada:
 - każde liczbowe zero (`0`, `0.0`, `0j`)
 - pusty/a: łańcuch znaków, lista, lub inny wbudowany typ złożony
 - stała `None` i oczywiście stała `False`
- Każda inna wartość będzie ewaluowana jako `True`
- Do jawnej ewaluacji służy funkcja `bool()`

`bool(-1)`



Operatory porównania

- Wynikiem działania wartość typu bool

równość

$==$

nierówność

$!=$

większy / mniejszy

$>$ $<$ \geq \leq



Przykłady wyrażeń logicznych

4 > 5

True and False

(4 > 5) or not(45 + 1 != 45)

(saldo > 45) and (wiek >= 3)



Programowanie obiektowe

TYPY LICZBOWE OPERATORY ARYTMETYCZNE



UNIwersytet
EKONOMICZNY
W POZNANIU

Systemy liczbowe

Ludzie

- System dziesiętny
 - Dziesięć palców
 - Historycznie jeden z wielu
 - Symbole 0 1 2 3 4 5 6 7 8 9
- Przykłady
 - 3
 - 35
 - 1215

Komputery

- System binarny
 - „Prąd płynie lub nie”
 - System dwójkowy
 - Symbole 0 i 1
- Przykłady
 - 11
 - 100011
 - 10010111111



System binarny

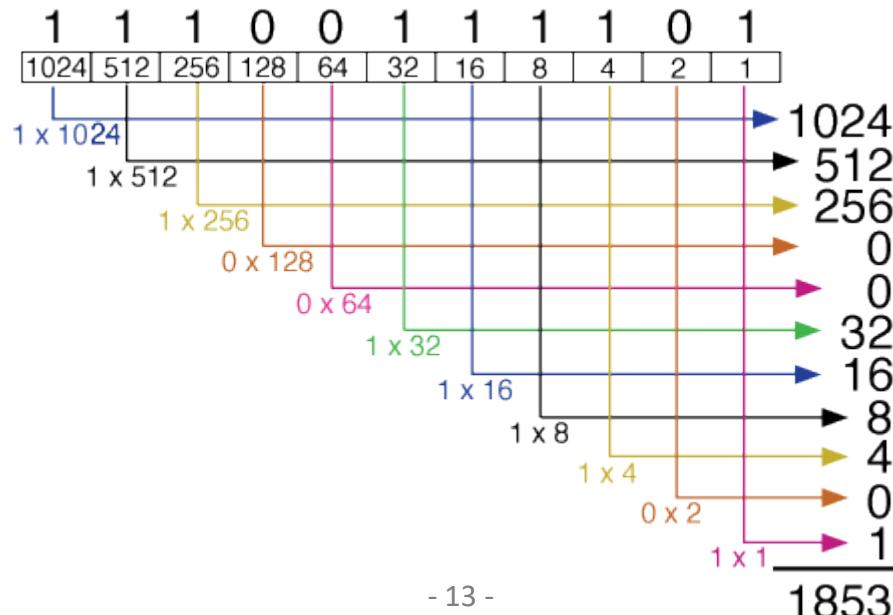
- Kolejne bity jako mnożniki potęg dwójki

$$1853 = 1024 + 512 + 256 + 32 + 16 + 8 + 4 + 1$$

$$1853 = 2^{10} + 2^9 + 2^8 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0$$

$$1*2^{10} + 1*2^9 + 1*2^8 + 0*2^7 + 0*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$$

1 1 1 0 0 1 1 1 1 0 1



Dodatnie liczby całkowite

- Zbiór bitów

8 bitów (bit) = bajt (byte)

2 bajty (byte) = słowo (word)

4 bajty (byte) = podwójne sł. (doubleword, longword)

8 bajtów (byte) = poczwórne sł. (quadword, longword)

- n bitów

$$0 \leq x \leq 2^n - 1$$

- 1 bajt = 8 bitów

$$0_{10} = 0000000_2 \leq x \leq 2^8 - 1 = 255_{10} = 11111111_2$$



Liczby całkowite ze znakiem

- Trzy metody
 - Znak-moduł (Sign-and-magnitude)
 - Kod **uzupełnień do jedności U1**
(One's complement)
 - Kod **uzupełnień do dwóch U2**
(Two's complement)
- Najbardziej popularna
 - Kod uzupełnień do dwóch



Znak-moduł

- 1 bit dla znaku
 - 0 = dodatnie
 - 1 = ujemne
- Inne bity dla modułu
 - +5 = **0101**
 - 5 = **1101**



Znak-moduł – konsekwencje

- Dwa zera
 - +0: 0000
 - 0: 1000
- Dla n bitów
 - Liczby od $-2^{n-1}+1$ do $2^{n-1}-1$
- 1 bajt = 8 bitów
 - $-127 \leq n \leq 127$
- Skomplikowane działania arytmetyczne



Uzupełnienie do jedności

- Inwersja

$1 \rightarrow 0$

$0 \rightarrow 1$

- Liczba ujemna = inwersja liczby dodatniej

5 = 0101

!5 = 1010

-5 = 1010



Uzupełnienie do jedności – konsekwencje

- Pierwsza cyfra = znak
0 dla dodatnich, 1 dla ujemnych
- Dwa zera
 $+0: 0000, -0: 1111$
 $\text{np. } 5 + (-5) = 0101 + 1010 = 1111 = -0$
- Dla n bitów
 - Liczby od $-2^{n-1}+1$ do $2^{n-1}-1$
- 1 bajt = 8 bitów
 $-127 \leq n \leq 127$



Uzupełnienie do dwóch

- Ujemna liczba = inwersja + 1

$$5 = 0101$$

$$!5 = 1010$$

$$!5 + 1 = 1011$$

$$-5 = 1011$$



Uzupełnienie do dwóch – konsekwencje

- Pierwsza cyfra = znak
 - 0 dla dodatnich
 - 1 dla ujemnych
- Jedno zero
 - 0: 0000
- Dla n bitów
 - Liczby od -2^{n-1} do $2^{n-1}-1$
- 1 bajt = 8 bitów
 - $-128 \leq n \leq 127$



Liczby całkowite w języku Python

- Typ `int` (od: *integer*, liczba całkowita)
- Ze znakiem: wartości dodatnie i ujemne
- Uzupełnienie do dwóch
- W języku Python 3 tożsame z `long`
- ...a więc praktycznie bez limitów



Liczby rzeczywiste

- Reprezentacja stałoprzecinkowa
 - Osobne zestawy bitów na część całkowitą i część ułamkową
 - Przykłady
 - $0.5 = 1/2 = 00000000.10000000$
 - $1.25 = 1 \frac{1}{4} = 00000001.01000000$
 - $7.375 = 7 \frac{3}{8} = 00000111.01100000$
- Reprezentacja zmiennoprzecinkowa
 - Zapis wykładniczy
 - Standaryzowany w IEEE 754

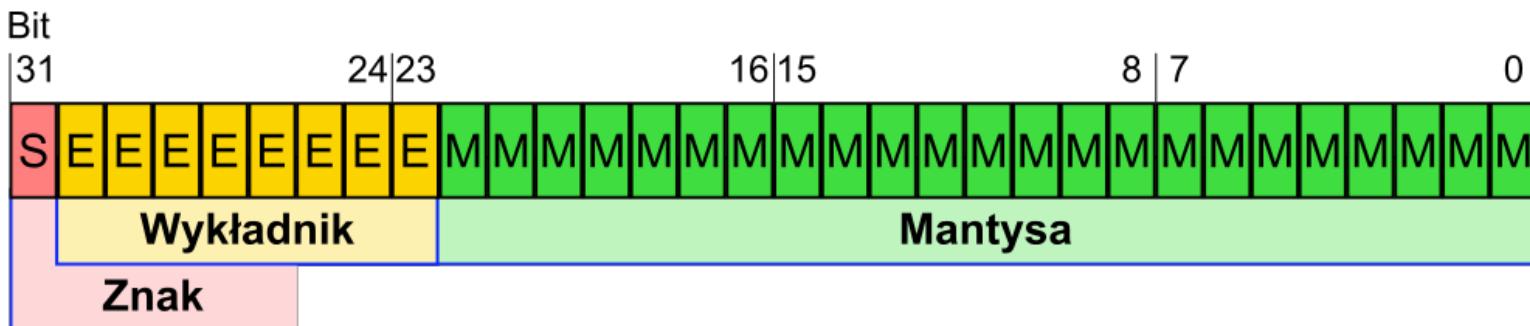


Reprezentacja zmiennoprzecinkowa

- Zapis wykładniczy

znak \times (1 + ułamek mantisy) \times $2^{\text{wykładnik}}$

- Reprezentacja binarna



Reprezentacja zmiennoprzecinkowa

- Przykłady

$$0.05 = 5 \times 10^{-2} = 1.6 \times 2^{-5}$$

$$-6.84 = -6.84 \times 10^0 = -1.71 \times 2^2$$

$$0.085 = 8.5 \times 10^{-2} = 1.36 \times 2^{-4}$$

$$0.085 = +1.36 \times 2^{-4}$$

0 01111011 01011100001010001111011



Liczby rzeczywiste w języku Python

- Typ float (od: *floating-point real number*, liczba rzeczywista zmiennoprzecinkowa)
- Ze znakiem: wartości dodatnie i ujemne
- Na ogół reprezentowane na 64 bitach
 - jak *double* w innych językach
- ...a więc z przedziału
od -1.797693134862E+308 do 1.797693134862E+308



Liczby zespolone w języku Python

- Typ `complex`
- Część rzeczywista i urojona ($i^2=-1$)
- Część urojona oznaczona symbolem `j`
- Przykład

```
z1 = 4 + 6j  
z2 = -0.5 + 2j  
z3 = z1 * z2  
print(z3)
```



Operatory arytmetyczne w języku Python

- 2 argumenty
 - Dodawanie **+** op1 + op2
 - Odejmowanie **-** op1 - op2
 - Mnożenie ***** op1 * op2
 - Dzielenie (zwr. float) **/** op1 / op2
 - Dzielenie całkowite **//** op1 // op2
(wynik bez cz. ułamkowej, „podłoga”)
 - Modulo **%** op1 % op2
 - Potęgowanie ****** op1 ** op2



Programowanie obiektowe

OPERATORY BITOWE



UNIwersytet
EKONOMICZNY
W POZNANIU

Operatory bitowe w języku Python

- 1 argument
 - Inwersja \sim $\sim \text{op}$
- 2 argumenty
 - AND $\&$ $\text{op1} \& \text{op2}$
 - OR $|$ $\text{op1} | \text{op2}$
 - XOR \wedge $\text{op1} \wedge \text{op2}$



Operatory bitowe

przykłady działania w języku Python

- 1 argument

- Inwersja \sim

$$\sim 5_{10} = \sim 0101_2 = 1010_2 = -6_{10} \quad (\text{dopełnienie } 2)$$

$$\sim X_{10} = -(X+1)_{10}$$

- 2 argumenty

- AND $\&$

$$5_{10} \& 3_{10} = 101_2 \& 011_2 = 001_2 = 1_{10}$$

- OR $|$

$$5_{10} | 3_{10} = 101_2 | 011_2 = 111_2 = 7_{10}$$

- XOR \wedge

$$5_{10} \wedge 3_{10} = 101_2 \wedge 011_2 = 110_2 = 6_{10}$$



Operatory przesunięcia

- 2 argumenty

- w prawo

$$6_{10} >> 2_{10}$$

$$0110_2 >> 2_{10} = 0001_2 = 1_{10}$$

- w lewo

$$6_{10} << 2_{10}$$

$$0110_2 << 2_{10} = 11000_2 = 24_{10}$$



Operatory przesunięcia

liczby ujemne

- 2 argumenty

- w prawo

$$-5_{10} >> 2_{10}$$

>>

op1 >> op2

$$1011_2 >> 2_{10} = \textcolor{red}{11}10_2 = -2_{10}$$

- w lewo

$$-5_{10} << 2_{10}$$

<<

op1 << op2

$$1011_2 << 2_{10} = 101100_2 = -20_{10}$$



Programowanie obiektowe

OPERATORY PRZYPISANIA



UNIwersytet
EKonomiczny
W POZNANIU

Operatory przypisania

- Wykonanie operacji i przypisanie wyniku

$$x = x \text{ operator } y \Leftrightarrow x \text{ operator=} y$$
$$x = x + y \Leftrightarrow x += y$$

- 2 argumenty

$$x += y$$
$$x -= y$$
$$x *= y$$
$$x **= y$$
$$x /= y$$
$$x \% y$$
$$x \&= y$$
$$x |= y$$
$$x ^= y$$
$$x <<= y$$
$$x >>= y$$
$$x // y$$


Wielokrotne przypisania

- Wielokrotne przypisanie jednej wartości do wielu zmiennych

```
a = b = c = 4
```

```
print(b)
```

- Wielokrotne przypisanie kilku wartości do kilku zmiennych

```
a, b, c = 4, 8.2, 'tekst'
```

```
print(a,b,c)
```



Programowanie obiektowe

TYP TEKSTOWY



UNIWERSYTET
EKONOMICZNY
W POZNANIU

Kodowanie znaków

- Kodowanie znaków
 - *Character encoding*
 - Znak ↔ liczba
 - Przykłady →
- Zbiór znaków
 - *Character set*
 - Kolekcja wszystkich kodów i odpowiadających im symboli

Dec	Char	Dec	Char	Dec	Char
32	SPACE	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	\$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(72	H	104	h
41)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[123	{
60	<	92	\	124	
61	=	93]	125	}
62	>	94	^	126	~
63	?	95	_	127	DEL



Popularne kodowanie znaków

- Standard ASCII
 - American Standard Code for Information Interchange
 - Alfabet rzymski na 7 bitów
 - Przykłady: A = 65, B = 66, a = 97
- Standardy ISO-8859-x
 - 8 bitów: 1 bit dla dodatkowych znaków
- Windows Cp125x
 - Niestandardyzowane
 - 8 bitów



Popularne kodowanie znaków

- Rodzina Unicode
 - Kodowania UTF-32, UTF-16, UTF-8, UTF-7
 - Różna liczba bitów na znak
 - Wszystkie pisma używane na świecie
- Domyślne kodowanie w języku Python
 - UTF-8

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
01C0	I	ł	‡	!	DŽ	Dž	dž	LJ	Lj	lj	NJ	Nj	nj	Ă	ă
01D0	ĩ	ő	ö	ශ	්	ු	ූ	ු	්	ශ	්	ු	ූ	ු	්
01E0	Ā	ā	Ā	ā	G	g	Ğ	ğ	Ķ	ķ	Q	q	Ӯ	Ӯ	Ӯ



Typ tekstowy w języku Python

- Typ tekstowy str od „string”, czyli łańcuch znaków
- Domyślnie kodowany UTF-8
- Zapis w apostrofach lub w cudzysłowie (gdy tekst zawiera apostrofy)
- Znaki specjalne poprzedzamy odwrotnym ukośnikiem \
- 3 apostrofy/cudzysłowy -> łańcuchy wieloliniowe



Typ tekstowy w języku Python

- Przykłady

```
zmienna = "cześć"
```

```
print(zmienna)
```

```
zmienna = 'cześć'
```

```
print(zmienna)
```

```
zmienna = """Dzień dobry,
```

```
cześć
```

```
i czołem"""
```

```
print(zmienna)
```



Znaki specjalne

- Znaki specjalne poprzedzamy \

tab = '\t'

newline = '\n'

quote = '\''

doublequote = '\"'

backslash = '\\'

- Ale w łańcuchu poprzedzonym **r** (od raw = surowy) znak \ nie wprowadza znaku specjalnego



Typ tekstowy w języku Python

- Przykłady

```
zmienna = "c:\\nowyfolder"  
print(zmienna)
```

```
zmienna = "c:\\\\nowyfolder"  
print(zmienna)
```

```
zmienna = r"c:\\nowyfolder"  
print(zmienna)
```



Typ tekstowy w języku Python

- Wiele linii w zapisie jednoliniowym

```
zmienna = "Dzień dobry\nczęść\ni czołem"  
print(zmienna)
```

- Jedna linia w zapisie wieloliniowym

```
zmienna = """Dzień dobry, \  
część \  
i czołem"""  
print(zmienna)
```



Wsparcie znaków różnych języków

- Kody Unicode poprzedzamy \u
- Przykład

```
tekst="\u0423\u043A\u0440\u0430\u0457\u0301\u043D\u0430"  
print(tekst)
```



Łączeniełańcuchów znaków

w języku Python

- W przypadku bezpośrednich wartości (nie: zmiennych) łańcuchy można ze sobą po prostu zestawiać
- Przykład

```
zdanie = "Ala"" ma psa"  
print(zdanie)  
zdanie = ("Ala"  
          " ma psa")  
print(zdanie)
```



Łączeniełańcuchów znaków w języku Python

- Dla zmiennych i wartości typu str
- Operator łączenia +
- Przykład

```
imie = "Ala"  
zdanie = imie + " ma psa"  
print(zdanie)
```



Powielanie łańcuchów znaków w języku Python

- Dla zmiennych i wartości typu str
- Operator powielenia *
- Przykład

```
imie = "Ala"  
zdanie = imie + " ma psa" + 3 * " i psa"  
print(zdanie)
```



Łączenie tekstów z... w języku Python

- Aby wykorzystać operator **+** do łączenia tekstów z innymi wartościami / zmiennymi
 - logicznymi, liczbowymi...
- ...wartość nie-tekstowa musi zostać zamieniona na tekst za pomocą funkcji `str()`
- Przykład

```
ile = 3  
zdanie = "Ala ma "+ str(ile) +" psy"  
print(zdanie)
```



Indeksowanie łańcuchów znaków w języku Python

- Za pomocą nawiasów kwadratowych []
- Łańcuchy są indeksowane **od zera**
- Ujemne indeksy – odwrotny kierunek indeksowania
- Przykład

```
zdanie = "Ala ma 3 psy"  
print("Liczba psów: " + zdanie[7])  
print("Liczba psów: " + zdanie[-5])  
liczba = int(zdanie[7]) + 1;  
print("Liczba psów: " + str(liczba))  
print("Liczba psów: " + zdanie[12]) #Błąd  
print("Liczba psów: " + zdanie[12:23]) #Brak błędu
```

Wycinanie łańcuchów znaków w języku Python

- Za pomocą przedziałów **[X:Y]**
- X i Y są opcjonalne
- X – domknięta granica przedziału
- Y – otwarta granica przedziału
- Przykład

```
zdanie = "Ala ma 3 psy"  
print(zdanie[:])  
print("Rodzaj zwierząt: " + zdanie[9:12])  
print("Rodzaj zwierząt: " + zdanie[9:])  
print(zdanie[9:] + " posiada " + zdanie[:3])  
print(zdanie[:2]+zdanie[2:])  
print(zdanie[:5]+zdanie[5:]) #s[:Z]+s[Z:] to s
```



Długość łańcuchów znaków w języku Python

- Za pomocą funkcji `len()`
- Przykład

```
zdanie = "Ala ma 3 psy"  
print(len(zdanie))
```



Niezmienność łańcuchów znaków w języku Python

- Uwaga: łańcuchy znaków są niezmienne (ang. immutable)
- Przykład

```
zdanie = "Ala ma 3 psy"
```

```
zdanie[7] = "4" #Błąd
```



Programowanie obiektowe

TYPY BINARNE



UNIwersytet
EKONOMICZNY
W POZNANIU

Typ binarny w języku Python

- Typ binarny bytes – przechowuje łańcuch bajtów
- Przykład

```
tekst = 'cześć'  
print(tekst, len(tekst))  
bajty = bytes(tekst, 'utf-8')  
print(bajty, len(bajty))
```



Tablica bajtów w języku Python

- Typ binarny `bytearray` – przechowuje tablicę bajtów
- Przykład

```
tekst = 'cześć'  
tablicaBajtow = bytearray(tekst, 'utf-8')  
print(tablicaBajtow)  
tablicaBajtow[0]=ord("C")  
print(tablicaBajtow)  
print(tablicaBajtow.decode())
```





UNIWERSYTET
EKONOMICZNY
W POZNANIU



Dziękuję!

Do zobaczenia za tydzień



UNIWERSYTET
EKONOMICZNY
W POZNANIU



Listy, krotki, zbiory, słowniki

Programowanie obiektowe

dr hab. inż. Adam Wójtowicz, prof. UEP

Typy danych

- Typ logiczny
 - bool
- Typy liczbowe
 - int, float, complex
- Typ tekstowy
 - str
- Typy binarne
 - bytes, bytearray
- Typy sekwencyjne
 - list, tuple, range
- Typy zbiorów
 - set, frozenset
- Typ słownikowy
 - dict



Programowanie obiektowe

TYPY SEKWENCYJNE: LISTA, KROTKA, PRZEDZIAŁ



UNIwersytet
EKonomiczny
W POZNANIU

Typy sekwencyjne

- **Typy sekwencyjne** to typy danych przechowujące wiele wartości w jednej zmiennej w sposób **uporządkowany** (ang. *ordered*)
- Do typów sekwencyjnych należą:
 - **lista** – list
 - **krotka** – tuple
 - **przedział** – range
- ...ale też typami sekwencyjnymi są typy poznane wcześniej:
 - typ tekstowy – str
 - typ binarny – bytes
 - tablica bajtów – bytearray



Lista

- Typ list
- Indeksowanie od 0 w [], wycinanie [X:Y], indeksy ujemne
 - jak w typie tekstowym – patrz poprzedni wykład
- Na liście można umieszczać duplikaty wartości
- Tworzymy korzystając z []

```
lista = [3, 6, -15, 6]
```

```
print(lista)
```

```
print(lista[2])
```



Lista – modyfikacje

- Lista jest uporządkowana (ang. *ordered*), nadany porządek „sam” się nie zmienia
- Zawartość listy może być zmieniana (ang. *mutable*)
 - można modyfikować, dodawać i usuwać elementy listy

```
lista = [3, 6, -15, 6]
```

```
lista[2]=100
```

```
lista.append(0)
```

```
lista.remove(6)
```

```
lista.insert(1,7)
```

```
lista.pop(0)
```

```
print(lista)
```



Lista – typy elementów, długość

- Elementy listy mogą być różnego typu (nawet na jednej liście)

```
lista = [3, -6.43, "no data", True]  
print(lista)
```

- Długość listy jest zwracana przez funkcję len()

```
print(len(lista))
```



Lista – sortowanie

- Elementy listy można sortować funkcją `sort()`, jeżeli operator porównania jest zdefiniowany dla par typów:

```
lista = [3, 6, -15, 6]
lista.sort()
print(lista)
lista = [3, 6, -15, 5.5]
lista.sort() #OK
print(lista)
lista = [3, 6, -15, "5.5"]
lista.sort() #błąd
```



Lista – kierunek sortowania

- Sortować funkcją sort () można rosnąco lub malejąco korzystając z parametru reverse:

```
lista = [3, 6, -15, 5.5]
```

```
lista.sort(reverse = True)
```

```
print(lista)
```



Lista – sortowanie typów tekstowych

- Sortując funkcją `sort()` można rozróżniając lub nie małe/wielkie litery:

```
lista = ["Wybór", "pobierzDane",
         "Użytkownik", "wyszukaj"]

lista.sort(reverse = False)

print(lista)

lista.sort(key = str.lower, reverse = False)

print(lista)
```



Lista – kopiowanie

- List nie można kopiować operatorem przypisania =

```
lista = [3, 6, -15, 5.5]
```

```
lista2 = lista
```

```
lista2[3] = 1000
```

```
print(lista) #te same wartości co w lista2
```

- Przypisywany jest adres w pamięci, a nie zawartość listy



Lista – kopiowanie

- Listy można kopiować metodą `copy()`

```
lista = [3, 6, -15, 5.5]
```

```
lista2 = lista.copy()
```

```
lista2[3] = 1000
```

```
print(lista) #inne wartości niż w lista2
```



Lista – kopiowanie

- Listy można kopiować również metodą

```
list()
```

```
lista = [3, 6, -15, 5.5]
```

```
lista2 = list(lista)
```

```
lista2[3] = 1000
```

```
print(lista) #inne wartości niż w lista2
```



Lista – łączenie

- Listy można łączyć operatorem +

```
lista = [3, 6, -15, 5.5]
```

```
lista2 = ["mały", "średni", "duży"]
```

```
lista3 = lista + lista2
```

```
print(lista3)
```



Lista – łączenie

- Listy można wydłużać metodą `extend()`

```
lista = [3, 6, -15, 5.5]
```

```
lista2 = ["mały", "średni", "duży"]
```

```
lista.extend(lista2)
```

```
print(lista)
```



Lista – powielanie

- Listy można powielać operatorem *

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

```
lista = 3 * lista
```

```
print(lista)
```



Lista – zliczanie

- Liczbę elementów o danej wartości można sprawdzić za pomocą metody `count()`

```
lista = ["mały", "średni", "średni", "duży"]
print(lista.count("średni"))
```



Lista – wyszukiwanie

- Indeks pierwszego elementu o danej wartości można pobrać za pomocą metody `index()`

```
lista = ["mały", "średni", "średni", "duży"]
print(lista.index("średni"))
```



Lista – przypisywanie elementów

- W jednym kroku można przypisać zawartość listy do wielu zmiennych

```
lista = [3.4, 6.9, -1.1]
```

```
x, y, z = lista
```

```
print(y)
```



Lista – zagnieżdżanie

- Zagnieżdżając listę w liście można tworzyć listy dwuwymiarowe lub wielowymiarowe

```
lista2D = [[3, 6, -15, 6], [1, 8, 4, -2],  
           [0, -3, -9, 10]]
```

```
print(lista2D[2])
```

```
print(lista2D[2][3])
```



Lista – kasowanie

- Metoda `clear()` usuwa wszystkie elementy z listy, **ale nie usuwa samej listy z pamięci**
- Funkcja `del()` usuwa listę (lub inny obiekt) **z pamięci**

```
lista2D[2].clear()  
print(lista2D)  
del(lista2D[1])  
print(lista2D)  
lista2D.clear()  
print(lista2D)  
del(lista2D)  
print(lista2D) #błąd
```



Lista – kasowanie

- Metoda `clear()` usuwa wszystkie elementy z listy, ale nie usuwa samej listy, ani jej elementów z pamięci
- Funkcja `del()` usuwa listę (lub inny obiekt) z pamięci, ale w przypadku listy **to nie oznacza, że usunięte z pamięci zostaną jej elementy**

```
lista2D = [[3, 6, -15, 6], [1, 8, 4, -2],  
           [0, -3, -9, 10]]
```

```
lista1D = lista2D[0]
```

```
lista2D[0][0]=300
```

```
print(lista1D) #lista1D i lista2D[0] to ta sama pamięć
```

```
lista2D.clear()
```

```
print(lista1D)
```

```
del(lista2D)
```

```
print(lista1D) #ciążę w pamięci
```



Krotka

- Typ tuple
- Krotki **tworzymy** korzystając z ()

```
krotka = (3, 6, -15, 6)
```

```
print(krotka)
```

```
print(krotka[2])
```



Krotka – brak modyfikacji

- Krotki są **niezmienne** (ang. *immutable*): zawartość krotki nie może być zmieniana
 - nie można modyfikować, dodawać i usuwać elementów, brak stosownych metod
 - można obejść to ograniczenie konwertując krotkę do listy za pomocą `list()`, modyfikując listę, a następnie konwertując z powrotem za pomocą `tuple()`



Krotka – podobieństwa do listy

- Indeksowanie od 0 w [], wycinanie [X:Y] , indeksy ujemne
- W krotce można umieszczać duplikaty wartości
- Krotka jest uporządkowana, nadany porządek „sam” się nie zmienia
- Elementy krotki mogą być różnego typu
- Długość krotki jest zwracana przez funkcję len ()
- Krotki można łączyć i powielać za pomocą operatorów + i *
- Elementy krotek można zliczać i wyszukiwać za pomocą metod count () i index ()
- Zawartość krotki można przypisać do wielu zmiennych za pomocą operatora =



Krotka jednoelementowa

- Tworząc krotkę jednoelementową trzeba po elemencie dodać przecinek

```
krotka = ("samotnik")  
print(type(krotka))  
  
print(krotka)  
  
krotka = ("samotnik",)  
print(type(krotka))  
  
print(krotka)
```

- Do takich testów – użyteczna funkcja `type()`



Przedział

- Typ range
- Ciąg arytmetyczny
- Przedziały tworzymy korzystając z **range()**

```
przedzial = range (3, 5)
```

```
print(przedzial[1]) #jak lista
```

```
print(przedzial) #ale to nie lista
```

```
print(type(przedzial))
```

- Zachowuje się podobnie jak lista, ale **nie jest listą**
- Nie materializuje danych lecz je **wylicza**
 - oszczędność pamięci dla dużych zbiorów danych



Przedział

- Trzeci parametr określa **krok**, może być ujemny

```
przedzial = range (-50, -100, -2)
```

```
print(przedzial[0], przedzial[1], przedzial[2])
```



Programowanie obiektowe

TYPY ZBIORÓW: ZBIÓR, ZBIÓR NIEZMIENNY



UNIwersytet
EKONOMICZNY
W POZNANIU

Typy zbiorów

- **Typy zbiorów** to typy danych przechowujące wiele wartości w jednej zmiennej w sposób **nieuporządkowany** (ang. *unordered*)
- Do typów zbiorów należą:
 - **zbiór** – set
 - **zbiór niezmienny** – frozenset



Zbiór

- Typ set
- Zbiór jest **nieuporządkowany** (ang. *unordered*)
 - nie jest indeksowany, nie działają []
- W zbiorze **nie można przechowywać duplikatów** wartości
 - dodanie duplikatu jest ignorowane
- Tworzymy korzystając z { }

```
zbior = {"user", "network", "device",
"server", "network"}  
  
print(zbior) #inna kolejność, brak duplikatu  
print(zbior[1]) #błąd
```



Zbiór – modyfikacje

- Elementy zbioru nie mogą być zmieniane
- Ale można dodawać i usuwać elementy do/ze zbioru

```
zbior = {"user", "network", "device"}  
zbior.add("server")  
zbior.remove("user")  
zbior.remove("user") #błąd, bo nie istnieje  
zbior.discard("user") #brak błędu  
zbior.pop() #usuwa losowy element  
print(zbior)
```

- `pop()` usuwa ostatni (więc losowy) element i go zwraca



Działania na zbiorach (1/2)

- `update()` – dodaje do zbioru elementy innego zbioru przekazanego jako parametr
 - i usuwa ewentualne duplikaty
- `union()` – zwraca nowy zbiór zawierający elementy zbioru, na którym ta metoda jest wywołana i zbioru przekazanego jako parametr
 - i usuwa ewentualne duplikaty
- `intersection_update()` – pozostawia w zbiorze tylko duplikaty ze zbioru, na którym ta metoda jest wywołana i zbioru przekazanego jako parametr
- `intersection()` – zwraca nowy zbiór zawierający duplikaty ze zbioru, na którym ta metoda jest wywołana i zbioru przekazanego jako parametr



Działania na zbiorach (1/2)

- Przykład

```
zbior = {"user", "network", "device"}  
zbior2 = {"user", "owner", "admin"}  
zbior3 = zbior.intersection(zbior2)  
print(zbior3)
```



Działania na zbiorach (2/2)

- `symmetric_difference_update()` – pozostawia w zbiorze wszystkie elementy, które nie są duplikatami dla zbioru, na którym ta metoda jest wywołana i zbioru przekazanego jako parametr
- `symmetric_difference()` – zwraca nowy zbiór zawierający wszystkie elementy, które nie są duplikatami dla zbioru, na którym ta metoda jest wywołana i zbioru przekazanego jako parametr
- `difference_update()` – pozostawia w zbiorze wszystkie elementy, których nie ma w zbiorze przekazanym jako parametr
- `difference()` – zwraca nowy zbiór zawierający wszystkie te elementy ze zbioru, na którym ta metoda jest wywołana, których nie ma w zbiorze przekazanym jako parametr



Działania na zbiorach (2/2)

- Przykład

```
zbior = {"user", "network", "device"}  
zbior2 = {"user", "owner", "admin"}  
zbior.symmetric_difference_update(zbior2)  
print(zbior)
```



Zbiór niezmienny

- Typ frozenset
- Brak metod z członem "update"
- Tworzymy przekazując zbiór, listę, etc. jako parametr
frozenset ()

```
z = {"user", "network", "device", "server",
```

```
"network"}
```

```
zn = frozenset(z)
```

```
print(zn)
```



Programowanie obiektowe

TYP SŁOWNIKOWY



UNIwersytet
EKONOMICZNY
W POZNANIU

Słownik

- Typ dict
- Słownik umożliwia przechowywanie **par** danych
klucz : wartość
- Odwołania do danych następują **przez klucz**
- W słowniku **nie można przechowywać duplikatów** wartości klucza
 - dodanie duplikatu **nadpisuje** poprzednią wartość



Słownik – przykład (1/3)

- Tworzymy korzystając z {} oraz par **klucz : wartość**

```
słownik = {  
    "user": "end user of the system",  
    "device": "any client-side hardware",  
    "server": "any server-side hardware"  
}  
  
print(słownik["device"] )
```



Słownik – przykład (2/3)

- Kluczami i wartościami **nie muszą być dane tekstowe**

```
uzytkownicy = {  
    1: "John Smith",  
    2: "Alice Jones",  
    3: "Harry Newman"  
}  
  
print(uzytkownicy[1])
```



Słownik – przykład (3/3)

- Typy kluczy i wartości **mogą się zmieniać** w jednym słowniku

```
webpage = {  
    "title": "My homepage",  
    "length": 31451,  
    "read-only": True  
}  
  
print(webpage["read-only"] )
```



Słownik – modyfikacje

- Słownik jest **uporządkowany** (ang. *ordered*)
 - nadany porządek „sam” się nie zmienia
 - ale słownik nie jest indeksowany liczbowo, tylko za pomocą kluczy



Słownik – modyfikacje

- Zawartość słownika **może być zmieniana** (ang. *mutable*)
 - można modyfikować, dodawać i usuwać elementy słownika

```
webpage["title"] = "My great hompage" #zmiana  
print(webpage)  
webpage.update({"length": 54000}) #zmiana  
print(webpage)  
webpage["format"] = "HTML5" #dodanie pary  
print(webpage)  
webpage.pop("read-only") #usunięcie pary  
print(webpage)  
webpage.popitem() #usunięcie ostatnio dodanej  
print(webpage)
```



Słownik – dostęp

- Dostęp do wszystkich **kluczy** słownika w postaci listy: metoda `keys()`
 - zwracana jest lista zawierająca te same obiekty-klucze
 - zatem zmiana słownika (np. dodanie nowej pary) zmienia również listę

```
klucze = webpage.keys()
```

```
print(klucze)
```

```
webpage["encoding"] = "UTF-8"
```

```
print(klucze)
```



Słownik – dostęp

- Dostęp do wszystkich **wartości** słownika w postaci listy: metoda `values()`
- Dostęp do wszystkich **elementów** słownika (klucze i wartości) w postaci listy: metoda `items()`
- W/w listy również zawierają **te same obiekty** co słownik – jak w przypadku listy kluczy



Słowniki – długość

- Długość słownika jest zwracana przez funkcję `len()`



Słowniki – kopiowanie

- Słownika nie można kopiować operatorem przypisania, ponieważ przypisywany jest jedynie adres w pamięci, a nie zawartość słownika
 - podobnie jak w przypadku list
- Słownik można kopiować metodą `copy()` lub `dict()`



Słowniki – zagnieżdżanie (1/2)

```
users = {
    "user1" : {
        "name" : "John Smith",
        "authentication" : "password"
    },
    "user2" : {
        "name" : "Alice Jones",
        "authentication" : "biometrics"
    },
    "user3" : {
        "name" : "Harry Newman",
        "authentication" : "smart card"
    }
}
print(users)
print(users["user1"]["name"])
```



Słowniki – zagnieżdżanie (2/2)

```
user1 = {  
    "name" : "John Smith",  
    "authentication" : "password"  
}  
  
user2 = {  
    "name" : "Alice Jones",  
    "authentication" : "biometrics"  
}  
  
user3 = {  
    "name" : "Harry Newman",  
    "authentication" : "smart card"  
}  
  
users = {  
    "user1" : user1,  
    "user2" : user2,  
    "user3" : user3  
}
```



Słowniki, listy – kopiowanie płytkie vs. kopiowanie głębokie

- Słownik/listę można kopiować metodą `copy()` lub `dict() / list()`
- Generowana jest tzw. płytką kopią
 - kopiowane są dane słownika/listy, ale jeżeli słownik/lista zawiera zagnieżdżone słowniki/listy, to one nie podlegają kopiowaniu (są referowane)
- Dlatego do kopiowania głębokiego zaleca się korzystanie z metody `deepcopy()`



Słowniki, listy – kopiowanie płytkie vs. kopiowanie głębokie

```
users2 = users.copy()  
  
users["user1"]["name"] = "Rick Smith"  
  
print(users2["user1"]["name"]) #to samo - płytna kopia  
  
import copy  
  
users3 = copy.deepcopy(users)  
  
users["user2"]["name"] = "Rick Smith"  
  
print(users3["user2"]["name"]) #inna wart. głęboka kopia
```





UNIWERSYTET
EKONOMICZNY
W POZNANIU



Dziękuję!

Do zobaczenia za tydzień



UNIWERSYTET
EKONOMICZNY
W POZNANIU



Instrukcje i funkcje

Programowanie obiektowe

dr hab. inż. Adam Wójtowicz, prof. UEP

Plan wykładu

- Instrukcje
- Operatory identyczności i zawierania
- Funkcje
- Wyrażenia lambda



Programowanie obiektowe

PODSTAWOWE INSTRUKCJE



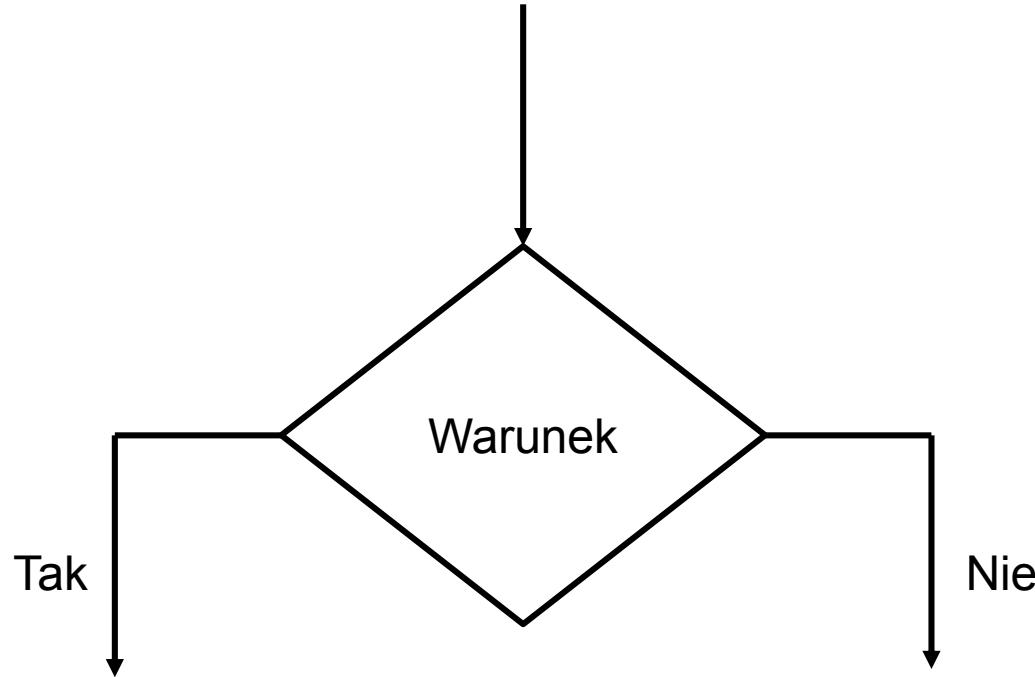
UNIwersytet
EKONOMICZNY
W POZNANIU

Definicja instrukcji

Instrukcja jest
częścią programu,
która można wykonać



Instrukcja warunkowa



Definicja instrukcji warunkowej

Instrukcja warunkowa wykonuje
różne instrukcje w zależności od
wartości danego **wyrażenia**



Definicja wyrażenia

Wyrażenie jest ciągiem wartości, zmiennych, operatorów i wywołań funkcji, którego ewaluacja (wynik obliczeń) jest **wartością pewnego typu**



Wyrażenia w języku Python

5

licznik

licznik + wartoscMinimalna

3 * (pi + 5)

cenaNetto + obliczPodatek(cenaNetto)

czyKoniec == True

(kategoria > 3) and (not czyKoniec)



Instrukcje warunkowe w języku Python

- Jeden albo nic

if ...

- Jeden z dwóch

if ...

else

- Jeden z wielu

if ...

match ...

elif ...

case ...

elif ...

case ...

else

case ...



Instrukcja if

- Składnia

if warunek :

wcięty BlokTak

- Składnia skrócona

if warunek : instrukcjaTak

- warunek

- wyrażenie o wartości typu bool
 - bezpośrednio lub po konwersji



Inne wartości ewaluowane jako logiczne

- W języku Python wartość `False` posiada:
 - każde liczbowe zero (`0`, `0.0`, `0j`)
 - pusty/a: łańcuch znaków, lista, lub inny wbudowany typ złożony
 - stała `None` i oczywiście stała `False`
- Każda inna wartość będzie ewaluowana jako `True`
- Do jawnej ewaluacji służy funkcja `bool()`

`bool(-1)`



Przykłady instrukcji if

```
if (a % 2) == 1 :  
    print(str(a) + " jest nieparzyste")  
  
if a % 2 : print(str(a) + " jest nieparzyste")
```



Instrukcja if else

- Składnia

if warunek :

wciętyBlokTak

else :

wciętyBlokNie

- wciętyBlokTak i wciętyBlokNie - mogą zawierać kolejne „zagnieżdżone” instrukcje if-else

- Składnia skrócona („warunkowy operator ternarny”)

instrukcjaTak **if** warunek **else** instrukcjaNie

- instrukcjaTak i instrukcjaNie - zwykłe pojedyncze instrukcje lub kolejne „zagnieżdżone” instrukcje if-else o skróconej składni



Przykłady instrukcji if else

```
if (a % 2) == 1 :  
    print(str(a) + " jest nieparzyste")  
else :  
    print(str(a) + " jest parzyste")  
  
print("nieparzyste") if a % 2 else print("parzyste")
```



Przykłady instrukcji if else

```
if jestWynik and jestMiejsce :  
    print("Zapis...")  
else :  
    pass
```



Zagnieżdżone instrukcje if else

```
if (a % 2) == 1 :  
    if (czyOtwarty) :  
        print(str(a) + " nieparzyste przy strumieniu otwartym")  
    else :  
        print(str(a) + " nieparzyste przy strumieniu zamkniętym")  
else :  
    if (czyOtwarty) :  
        print(str(a) + " parzyste przy strumieniu otwartym")  
    else :  
        print(str(a) + " parzyste przy strumieniu zamkniętym")
```



Instrukcja if elif else

- Składnia

if warunek :

wciętyBlok

elif warunek1 :

wciętyBlok

...

elif warunekN :

#dowolne N >= 0

wciętyBlok

else :

#opcjonalne else

wciętyBlok



Przykład instrukcji if elif else

```
if player1>player2 :  
    print("Pierwszy wygrał")  
elif player1==player2 :  
    print("Remis")  
else :  
    print("Drugi wygrał")
```



Instrukcja match case

match wyrażenie :

case wartość1:

 blok1

...

case wartośćN:

 blokN

- wyrażenie dowolnego typu
- wykonywany jest tylko blok pierwszego dopasowania
- znak _ oznacza dowolną wartość
- bloki mogą otrzymywać wartości w zmiennych



Przykład instrukcji match case

match kodBledu:

case 400:

 odpowiedz ("Nieprawidłowe zapytanie")

case 401:

 alertBezpieczenstwa=True

 odpowiedz ("Nieautoryzowany dostęp")

case 404:

 odpowiedz ("Nie znaleziono")

case _:

 odpowiedz ("Coś poszło nie tak")



Przykład instrukcji match case

match kodBledu:

case 500 | 501 | 502:

 odpowiedz("Błąd serwera")

case 503 | 504:

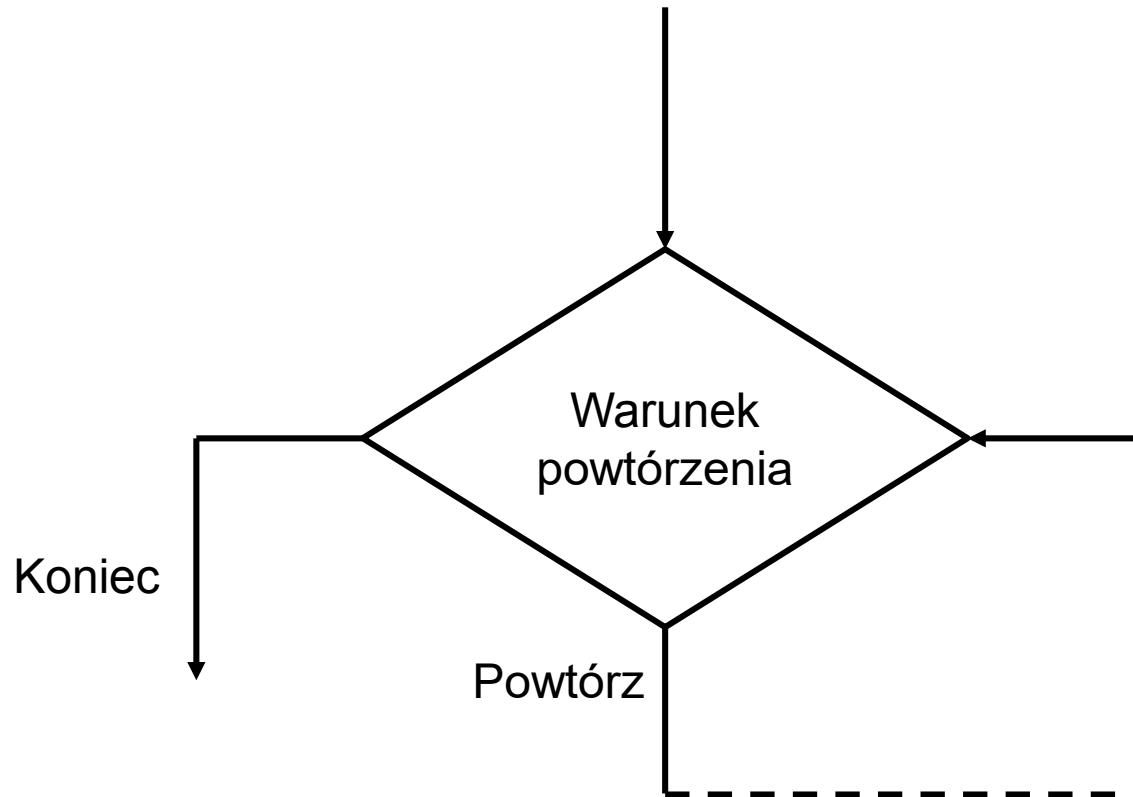
 odpowiedz("Przekroczono czas")

case _:

 odpowiedz("Coś poszło nie tak")



Instrukcja pętli



Definicja instrukcji pętli

Instrukcja pętli powtarza
wykonanie wskazanych instrukcji



Instrukcje pętli w języku Python

- Warunkowa

while

- Iteracyjna

for



Pętla warunkowa while

- Składnia

while warunek :

wciętyBlokPetli

else :

wciętyBlokKonca

#opcjonalnie

#opcjonalnie

- Składnia skrócona

while warunek : instrukcjaPetli

- warunek

- wyrażenie o wartości typu `bool` (bezpośrednio lub po konwersji)
- wciętyBlokPetli jest wykonywany tak długo, jak `warunek` jest spełniony (`True`)
- wciętyBlokKonca jest wykonywany jednokrotnie, gdy warunek nie jest spełniony



Przykład instrukcji while

```
tekst = 'programowanie'
```

```
i = 0
```

```
znalezione = 0
```

```
while i < len(tekst):  
    if tekst[i] == 'o':  
        znalezione+=1  
    i += 1  
print(znalezione)
```



Przykład instrukcji while

```
while i < len(tekst) :  
    if tekst[i] == 'o':  
        znalezione+=1  
        i += 1  
  
else :  
    print("zliczanie zakończone")  
print(znalezione)
```



Pętla iteracyjna for

- Pętla `for` służy do iterowania **po danych typu iterowanego (ang. iterable)**
- Składnia

```
for zmienna in wyrażenie :  
    wciętyBlokPetli  
  
else : #opcjonalnie  
    wciętyBlokKonca #opcjonalnie
```

- Składnia skrócona
- *zmienna*
 - zmienna sterująca przybierająca w iteracjach kolejne wartości
- *wyrażenie*
 - *wyrażenie* typu iterowanego, np. tekst, lista, krotka, zbiór, słownik
 - struktura określona przez *wyrażenie* źródłem wartości dla *zmienna*
 - rozmiar struktury z *wyrażenie* to l. wykonania wciętyBlokPetli

Przykład instrukcji for

```
tekst = 'programowanie'  
znalezienie = 0  
  
for litera in tekst:  
    if litera == 'o':  
        znalezienie+=1  
  
else :  
    print("zliczanie zakończone")  
print(znalezienie)
```



Przykład instrukcji for

```
str1 = "Monty"  
str2 = " Python"  
for x in str1+str2 :  
    print(x)
```



Przykład instrukcji for

```
webpage = {  
    "title": "My homepage",  
    "length": 31451,  
    "read-only": True  
}  
  
for x in webpage : print(x, webpage[x])
```



Pętle zagnieżdżone

```
for y in range(1,10) :  
    for x in range(1,10) :  
        iloczyn = x*y  
        if iloczyn<10 : print(' ', end=' ')  
        print(iloczyn, end=' ')  
    print("\n")
```



Definicja instrukcji zakończenia

**Instrukcja zakończenia
przerywa wykonywanie bloku**



Instrukcje zakończenia

- Przerwanie pętli for lub while
break
 - nie jest również wykonywany blok else przerwanej pętli
 - w przypadku wielu zagnieżdżonych pętli, przerywana jest tylko jedna - „wewnętrzna”
- Przerwanie iteracji pętli for lub while
 - nie wychodzi z pętli
 - wraca na początek pętli do kolejnej iteracji
continue
- Przerwanie bloku funkcji
return



Przykład break

```
liczba = 100;  
while liczba > 0 :  
    if liczba % 7 == 0 :  
        print(str(liczba)+" jest podzielna przez 7!")  
        break  
    print(str(liczba)+" nie jest podzielna przez 7")  
    liczba-=1
```



Przykład continue

```
for i in range(100, 1, -1) :  
    if i % 7 != 0 :  
        continue  
    print(str(i)+" podzielna przez 7")
```



Programowanie obiektowe

OPERATORY IDENTYCZNOŚCI I ZAWIERANIA



UNIWERSYTET
EKONOMICZNY
W POZNANIU

Operatory

- Operatory logiczne
 - iloczyn, suma, negacja
- Operatory porównania
 - równe, różne, większe, większe lub równe, ...
- Operatory arytmetyczne
 - dodawanie, odejmowanie, dzielenie, ...
- Operatory bitowe
 - iloczyn, suma, przesunięcie, ...
- Operatory przypisania
 - przypisanie, przypisanie z dodawaniem, ...
- **Operatory identyczności**
 - tożsamość, brak tożsamości
- **Operatory zawierania**
 - zawieranie, brak zawierania



Operatory identyczności

- Tożsamość
 - `is`
- Brak tożsamości
 - `not is`

Przykład

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

```
y = [1, 2, 3]
```

```
if (x is y) : print("ten sam obiekt")
```

```
else : print("innny obiekt")
```

```
y = x
```

```
if (x is y) : print("ten sam obiekt")
```

```
else : print("innny obiekt")
```



Operatory zawierania

- Zawiera
 - in
- Nie zawiera
 - not in
- Przykład

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

```
if (2 in x) : print("jest")
else : print("nie ma")
if (1,2 in x) : print("jest")
else : print("nie ma")
if ([1,2] in x) : print("jest")
else : print("nie ma")
```



Programowanie obiektowe

FUNKCJE



UNIwersytet
EKONOMICZNY
W POZNANIU

Funkcja

- Specyficzny blok kodu – nazwany, posiada **nazwę**
- Kod uruchamiany tylko gdy funkcja jawnie **wywołana**
- Funkcje porządkują kod
- Możliwa do wielokrotnego wykorzystania
 - code reuse
- Może posiadać **parametry wywołania** (=argumenty)
- Może przekazywać (zwracać) **wartość (wynik)**
- Funkcje wbudowane vs. funkcje użytkownika



Definiowanie funkcji – parametry

- Parametry (argumenty), przekazywane z zewnątrz do funkcji
 - Dowolnego typu, dowolnie wiele, oddzielone przecinkami
 - Parametr poprzedzony * oznacza dowolną liczbę parametrów (krotkę)
 - Parametr poprzedzony ** oznacza dowolną liczbę parametrów nazwa-wartość (słownik)
 - Parametrom można nadawać domyślne wartości:
`parametr1="wartośćDomyślna1"`



Definiowanie funkcji – return

- Wartość wyrażenia po instrukcji `return` jest zwracana na zewnątrz funkcji
 - Instrukcja `return` przerywa działanie funkcji
 - Instrukcja `return` bez wyrażenia zwraca wartość `None`
 - Funkcja bez instrukcji `return` zwraca wartość `None`



Definiowanie funkcji – ciało funkcji

- Zmienne definiowane wewnątrz funkcji nie są dostępne poza nią
 - zmienne lokalne
 - Wewnątrz funkcji można zdefiniować funkcje zagnieżdżone
 - ich widoczność jest również ograniczona do danej funkcji



Wywołanie funkcji

nazwa (parameter₁, ... , parameter_N)

- Z użyciem nazwy...
- ...i podaniem wartości argumentów zgodnie z definicją funkcji
 - jeżeli są wymagane
- Funkcja może wywoływać sama siebie (rekurencja)



Przykłady – parametry

```
def krzycz(x):
    return x+"!"

def pytaj(x):
    return "czy "+x+"?"

zdanie = "lubię pierogi"

wynik = krzycz(zdanie)
print(wynik)
wynik = pytaj(zdanie)
print(wynik)

wynik = pytaj() #błąd
wynik = pytaj(zdanie, zdanie) #błąd
```



Przykłady – nazwa-wartość

```
def funkcjaLiniowa1(x) :  
    return 3*x-5  
  
def funkcjaLiniowa2(x1,x2) :  
    return 3*x1-2*x2+6  
  
print(funkcjaLiniowa1(7))  
print(funkcjaLiniowa2(7,4))  
print(funkcjaLiniowa2(x2=4,x1=7))
```



Przykłady – wartości domyślne

```
def funkcjaLiniowa2(x1, x2=0):  
    return 3*x1-2*x2+6  
  
print(funkcjaLiniowa2(7, 4))  
print(funkcjaLiniowa2(7))  
print(funkcjaLiniowa2()) #błąd
```



Przykłady – dowolna liczba parametrów

```
def sredniaZeSkrajnych(*krotka):  
    suma = max(krotka) + min(krotka)  
    return suma/2  
  
print(sredniaZeSkrajnych(3, 7, 4, 6, 9, 2, 3, 7))  
print(suma) #błąd
```



Przykłady – dowolna liczba parametrów

```
def wypiszUzytkownika(**uzytkownik):  
    print(uzytkownik["login"] +  
          ": " + uzytkownik["imie"] +  
          " " + uzytkownik["nazwisko"])  
  
wypiszUzytkownika(imie="John", nazwisko="Smith",  
                      login="jsmith@example.com")
```



Przykłady – parametr obiektowy (1)

```
def dodajSrednia(li) :  
    srednia=sum(li)/len(li)  
    li.append(srednia) #ta sama lista  
    print(li)
```

```
lista = [1,1,10]  
dodajSrednia(lista)  
print(lista)
```



Przykłady – parametr obiektowy (2)

```
def dodajSrednia(li) :  
    srednia=[sum(li)/len(li)]  
    li = li + srednia #inna lista  
    print(li)
```

```
lista = [1,1,10]  
dodajSrednia(lista)  
print(lista)
```



Przykłady – zagnieżdżanie funkcji

```
def f1() :  
    print("f1")  
def f2() :  
    print("f2")
```

f2() #OK

f1()

f2() #błąd



Przykłady – funkcje wbudowane

```
imie = input("Podaj imię: ")  
dlugosc = len(imie)  
print(dlugosc)
```



Programowanie obiektowe

WYRAŻENIA LAMBDA



UNIwersytet
EKONOMICZNY
W POZNANIU

Wyrażenie lambda (funkcja anonimowa) to krótki kod pobierający parametry i zwracający wartość, ale w przeciwieństwie do funkcji nie posiadający nazwy i przechowywany jak zmienna

Wyrażenie lambda

lambda param₁, ... , param_N : wyrażenie

- Przykład:

```
f = lambda a, b : a * b
```

```
print(f(4, 5))
```



Wyrażenie lambda – przykład

```
users = [(1, 'Smith'), (2, 'Jones'), (3, 'Newman')]  
print(users)
```

```
lam = lambda krotka: krotka[1]  
users.sort(key=lam)  
print(users)
```





UNIWERSYTET
EKONOMICZNY
W POZNANIU



Dziękuję!

Do zobaczenia za tydzień



UNIWERSYTET
EKONOMICZNY
W POZNANIU



Programowanie obiektowe

Programowanie obiektowe

dr hab. inż. Adam Wójtowicz, prof. UEP

Plan wykładu

- Dlaczego programujemy obiektowo?
- Podstawowe pojęcia
 - klasa, obiekt, metoda, pole
- Obiekty w języku Python
- Konstruktor / destruktor
- Dziedziczenie
 - przykłady polimorfizmu
- Hermetyzacja



Programowanie obiektowe

DLACZEGO PROGRAMUJEMY OBIEKTOWO?



UNIwersytet
EKONOMICZNY
W POZNANIU

Programowanie Spaghetti

- Skomplikowany kod
 - z wieloma warunkami i zagnieżdżeniami
 - z użyciem przeskoków, np. goto
 - nieustrukturyzowane dane (zbior niezależnych zmiennych)
- Assembler, BASIC

Łatwość utrzymania: niska



Przykłady w BASIC

```
10 PRINT "Wprowadź liczbę, zero żeby wyjść:";  
20 INPUT A  
30 IF A = 0 THEN GOTO 70  
40 LET A = A + 10  
50 PRINT "Wprowadzona liczba + 10 = "; A  
60 GOTO 10  
70 STOP
```



Programowanie proceduralne

- Modularność kodu
 - procedury
 - funkcje
 - ustrukturyzowane dane
- PASCAL

Łatwość utrzymania: średnia



Przykład rekordu w PASCAL

```
program RECORD_INTRO (output);
type data = record
    miesiąc, dzień, rok : integer
end;
var dziś : data;
begin
    dziś.dzień      :=      25;
    dziś.miesiąc    :=      09;
    dziś.rok        := 1983;
    writeln('Dzisiaj: ',
            dziś.dzień, ':',
            dziś.miesiąc, ':',
            dziś.rok);
end.
```



Przykład procedury w PASCAL

```
program DODAJ_LICZBY (input, output);  
procedure DODAJ ( pierwsza, druga : integer );  
    var wynik : integer;  
begin  
    wynik := pierwsza + druga;  
    writeln('Wynik = ', wynik);  
end;
```

```
var liczba1, liczba2 : integer;  
begin  
    writeln('Wprowadź dwie liczby');  
    readln( liczba1, liczba2 );  
    DODAJ( liczba1, liczba2 );  
end.
```



Programowanie strukturalne

- Podział kodu na
 - deklaracje
 - definicje
- Nagłówki i biblioteki
- Ustrukturyzowane dane
- Język C

Łatwość utrzymania: wysoka



Przykład w C

W pliku myMath.h

```
int dodaj(int i, int j);
```

W pliku myMath.c

```
#include "myMath.h"
int dodaj(int i, int j) { return i+j };
```

W pliku myProg.c

```
#include "myMath.h"
#include <iostream.h>
#include <cstdlib>

int main( int argc, char* argv[] ) {
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    int suma = dodaj(a, b);
    cout << a << "+" << b << "=" << suma;
}
```



Ograniczenia języka C

- Brak bezpośredniego związku między
 - procedurami / funkcjami
 - strukturami danych
- Rozproszony kod
 - potencjalny problem spójności nagłówków i bibliotek



Programowanie obiektowe

- Powiązanie
 - struktur danych
 - procedur ich modyfikacji
 - wszystko razem w jednym pliku
- Analogia do świata fizycznego
 - cechy obiektu fizycznego = dane
 - zachowania obiektu fizycznego = metody

Łatwość utrzymania: wysoka



Obiektowe języki programowania

- Historia
 - Simula 67
 - Smalltalk
 - Object Pascal
- Współczesne obiektowe języki programowania
 - C++
 - Java
 - C#
 - **Python**
 - ...



Programowanie obiektowe

KLASA, OBIEKT, METODA, POLE



UNIwersytet
EKONOMICZNY
W POZNANIU

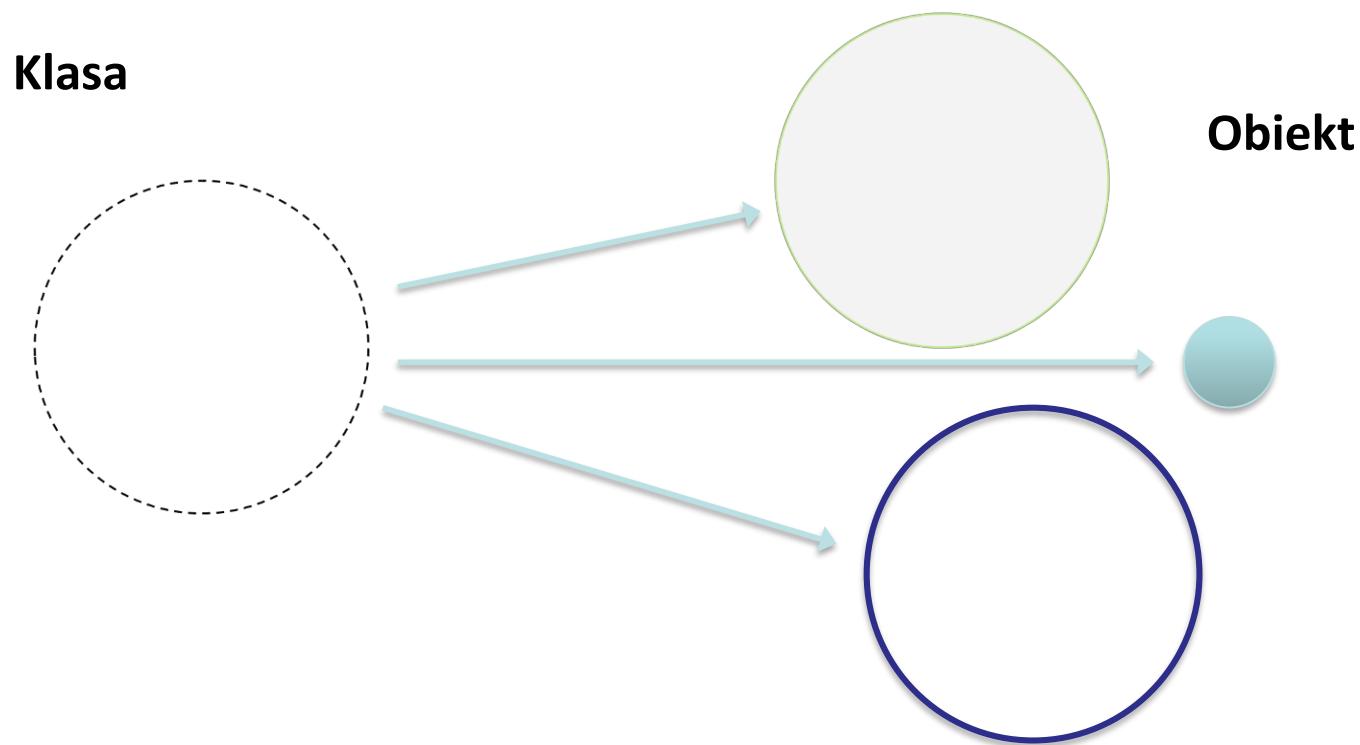
Klasa jest szablonem
cech obiektu
i jego zachowania



Obiekt jest konkretnym
wystąpieniem (instancją) klasy



Klasa a obiekt



Klasa a obiekt



Definiowanie klas w języku Python

- Składnia

```
class <nazwa> :  
    ...
```

- Przykład

```
class Kwadrat :  
    ...
```

- W jednym pliku można zdefiniować wiele klas
- Klasę można zdefiniować również w innej klasie (klasa wewnętrzna), a nawet w dowolnym bloku kodu
- Nazwa z reguły z dużej litery (konwencja, nie wymóg)



Klasa jest zbiorem
pól (zmiennych)
i *metod* operujących
na tych polach



Przykład klasy w języku Python

```
class Kwadrat :  
    opis = "Prostokąt, który ma wszystkie boki \  
    równej długości"  
  
    def liczPole(self):  
        return self.bok**2  
  
    def liczObwod(self):  
        return 4 * self.bok
```

pole klasy
(zmienna klasy)

metoda

pole obiektu
(zmienna obiektu)



Przykład klasy w języku Python

```
import math

class Kolo :
    opis = "Zbiór punktów na płaszczyźnie, których \
            odległość od środka jest <= promieniowi"

    def liczPole(self):
        return math.pi * self.promien**2

    def liczObwod(self):
        return 2 * math.pi * self.promien
```



Pola w języku Python

- **Pole klasy** (zmienna klasy, atrybut klasy)
 - przechowuje wartość wspólną dla wszystkich obiektów klasy („pole statyczne”) – przy klasie
 - definiowane w klasie, ale poza definicjami metod
 - wykorzystywane rzadziej niż pole obiektu
- **Pole obiektu** (zmienna obiektu, atrybut obiektu)
 - przechowuje niezależną wartość dla każdego obiektu danej klasy
 - definiowane wewnątrz definicji metody z użyciem pierwszego parametru metody (często nazywanego `self`)
 - częściej wykorzystywane



Metody w języku Python

- Metoda to blok kodu należący **do obiektu**
- Implementowana jako specjalna **funkcja** definiowana **wewnątrz klasy**
- Metoda operuje na polach klasy/obiektu



Programowanie obiektowe

OBIEKTY W JĘZYKU PYTHON



UNIwersytet
EKONOMICZNY
W POZNANIU

Obiekt jest zbiorem
wartości pól i metod
zdefiniowanych w danej klasie



Tworzenie obiektów

- Tworzenie obiektu

`<zmiennaPrzechowujacaObiekt> = <nazwaKlasy>()`

- Przykłady

`mojeKolo = Kolo()`

`mojeInneKolo = Kolo()`

`mojKwadrat = Kwadrat()`



Użycie obiektów

- Składnia użycia obiektu
`<zmiennaPrzechowujacaObiekt>.<poleLubMetoda>`
- Odwołanie do **pola obiektu** przez **zmienną przechowującą**
`mojeKolo.promien=4`
- Odwołanie do **pola klasy** przez **nazwę klasy**
`tekst = Kolo.opis
print(tekst)`
- Odwołanie do **metody** przez **zmienną przechowującą**
`print(mojeKolo.liczPole())`



Przykład: różne obiekty, niezależne wartości pól obiektów

```
kol = Kolo()  
kol.promien = 4  
ko2 = Kolo()  
ko2.promien = 2 * kol.promien  
wynik = kol.liczPole()  
print(wynik)  
innyWynik = ko2.liczPole()  
print(innyWynik)
```



Przykład: odwołania do pól klas przez nazwę klasy

```
ko1 = Kolo()  
ko2 = Kolo()  
Kolo.opis = "nowy opis" #odwołanie przez klasę  
print(Kolo.opis)  
print(ko1.opis)  
print(ko2.opis)
```



Przykład: odwołania przez zmienną przechowującą obiekt

- Uwaga:
- przypisanie nowej wartości do **pola klasy** za pomocą odwołania **przez zmienną przechowującą obiekt** tworzy **nowe pole dla tego obiektu**
- wartość pola klasy i wartości pól (o tej samej nazwie) innych obiektów tej klasy nie są wtedy modyfikowane

```
ko1 = Kolo()  
ko2 = Kolo()  
ko1.opis = "uwaga" #odwołanie przez zmienną  
print(Kolo.opis)  
print(ko1.opis)  
print(ko2.opis)
```



Programowanie obiektowe

KONSTRUKTOR I DESTRUKTOR



UNIwersytet
EKONOMICZNY
W POZNANIU

Konstruktor

- **Konstruktor** – specjalna metoda wywoływana automatycznie, tuż po utworzeniu obiektu w momencie jego inicjalizacji
 - metoda inicjalizacyjna
- W języku Python konstruktor nazywa się `__init__`

```
def __init__(lista_parametrów):  
    ...
```

- Pierwszym parametrem konstruktora jest konstruowany obiekt
 - zwykle nazwany `self`
 - można podać własną nazwę
 - w wywołaniu dodawany automatycznie



Przykład konstruktora

```
import math

class Kolo :

    def __init__(self):
        self.promien = 1.0
        self.pozycjaX = 10
        self.pozycjaY = 20

    def liczPole(self):
        return math.pi * self.promien**2

    def liczObwod(self):
        return 2 * math.pi * self.promien
```



Użycie konstruktora

```
k = Kolo()  
print(k.liczPole())
```



Konstruktor z parametrami

```
import math

class Kolo :

    def __init__(self, pr, x, y):
        self.promien = pr
        self.pozycjaX = x
        self.pozycjaY = y

    def liczPole(self):
        return math.pi * self.promien**2

    def liczObwod(self):
        return 2 * math.pi * self.promien
```



Użycie konstruktora z parametrami

```
k = Kolo(2.0, 100, 100)  
print(k.liczPole())
```



Destruktor

- Metoda klasy o specjalnym znaczeniu
- Wywoływana w momencie usunięcia obiektu z pamięci
 - ma za zadanie zwolnić pamięć zajmowaną przez pola obiektu i „posprzątać” po obiekcie
- Czasami możliwość „ręcznego” zarządzanie pamięcią jest przydatna
 - np. algorytmy o dużej złożoności pamięciowej



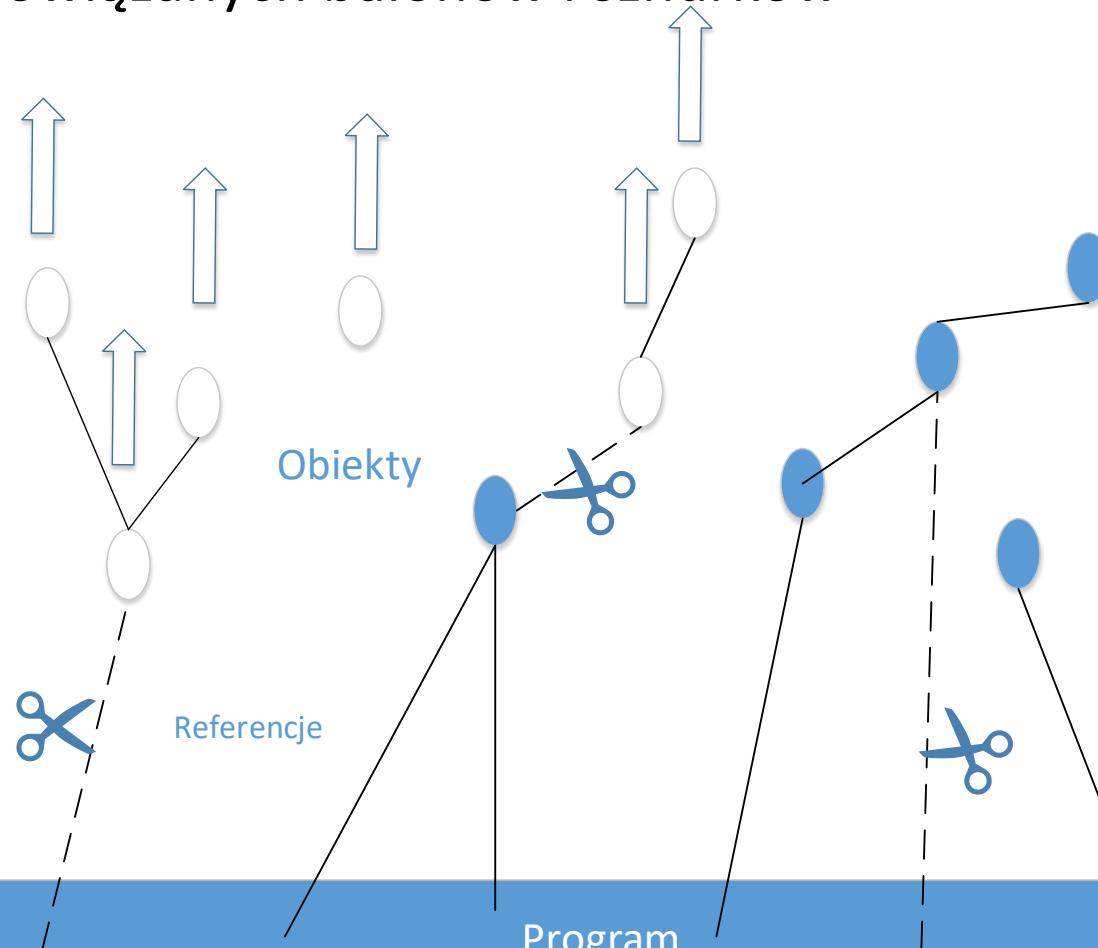
Destruktor – przykład

```
class Kolo :  
    def __init__(self):  
        self.promien = 1.0  
        self.pozycjaX = 10  
        self.pozycjaY = 20  
def __del__(self):  
    print("Sprzątanie...")  
def liczPole(self):  
    return math.pi * self.promien**2  
def liczObwod(self):  
    return 2 * math.pi * self.promien  
  
k = Kolo()  
del k
```



Garbage Collector

- Garbage Collector – automatycznie usuwa nieużywane obiekty
- Metafora powiązanych balonów i sznurków



Programowanie obiektowe

DZIEDZICZENIE KLAS



UNIwersytet
EKonomiczny
W POZNANIU

Dziedziczenie opisuje sposób współdzielenia pól i metod między wybranymi klasami



Produkt

- nazwa
- cena

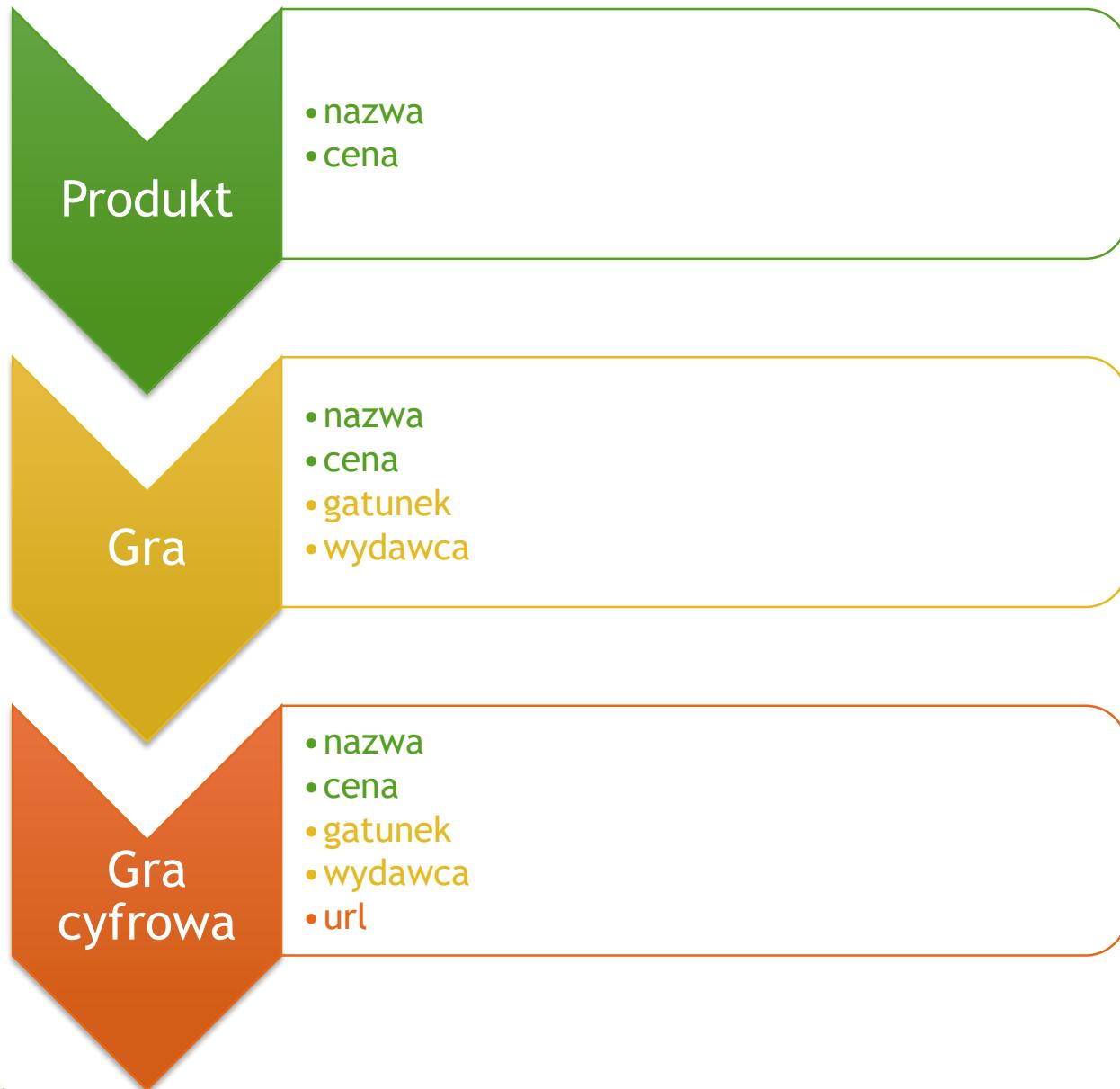
Gra

- gatunek
- wydawca

Gra cyfrowa

- url





Podklaśa to klasa,
która dziedziczy pola i metody
innej klasy

Nadklaśa to klasa, z której
dziedziczone są pola i metody
przez inną klasę



Nadklasy i podklasy w języku Python

- Składnia deklaracji podklasy

```
class NazwaPodkласy (Nadklaſa1, Nadklaſa2, ...):  
    ...
```

- Przykład

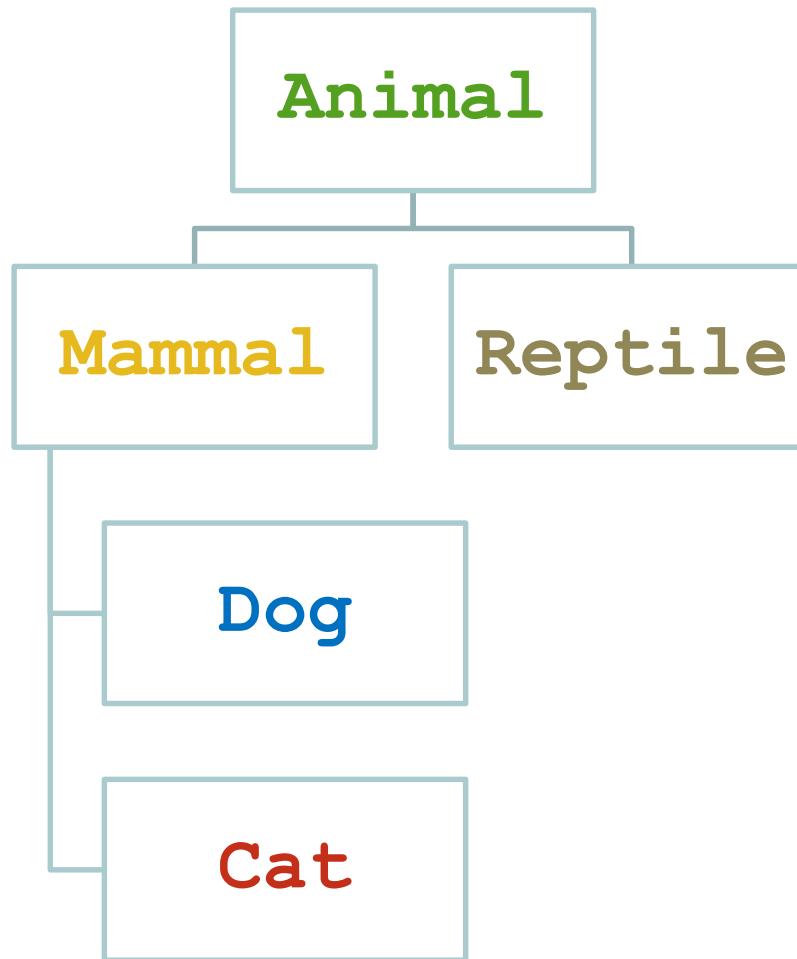
```
class Kolo (Figura):  
    ...
```

- Jako **Nadklaſa** można umieścić wyrażenie, np.

```
class Kolo (nazwaModulu.Figura):  
    ...
```



Hierarchia klas



Hierarchia klas

```
class Animal :  
    pass
```

```
class Mammal (Animal) :  
    pass
```

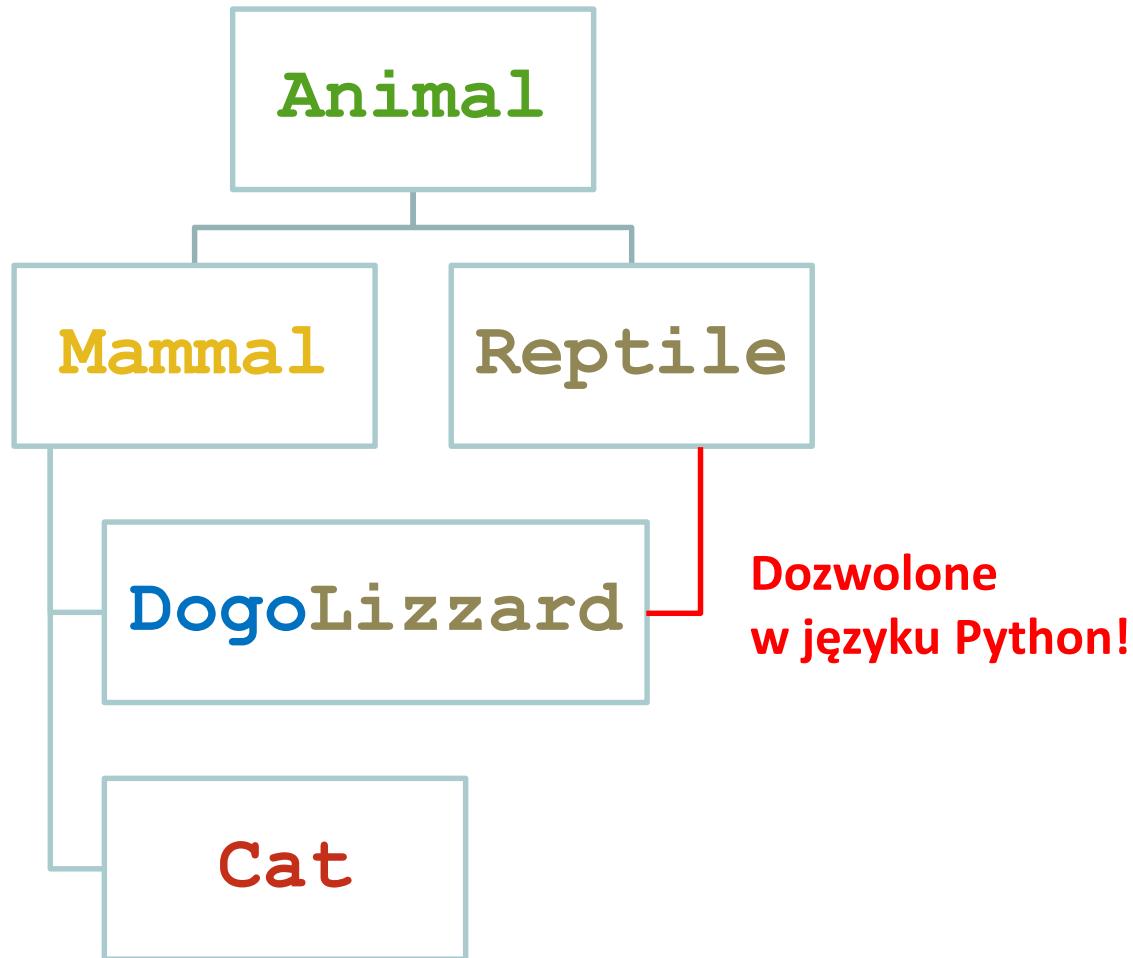
```
class Reptile (Animal) :  
    pass
```

```
class Dog (Mammal) :  
    pass
```

```
class Cat (Mammal) :  
    pass
```



Wielokrotne dziedziczenie



Wielokrotne dziedziczenie

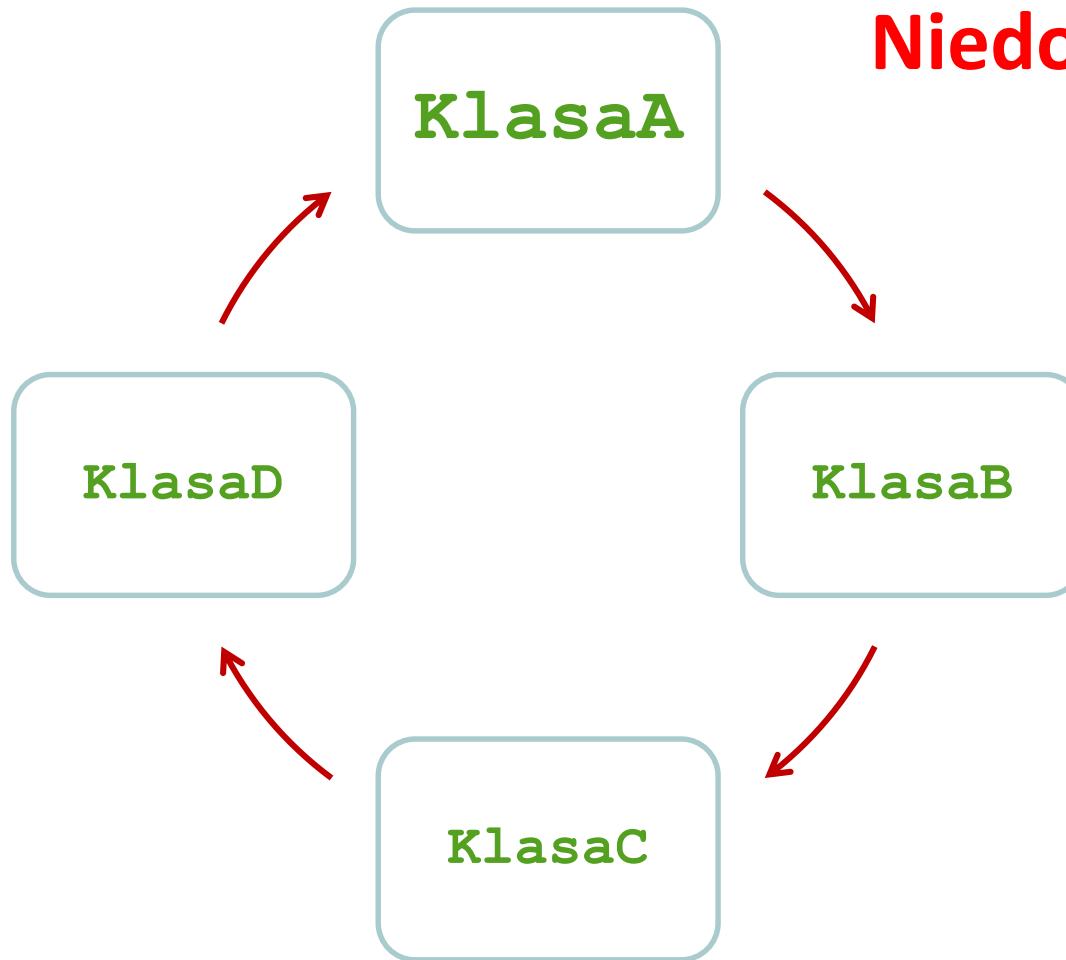
- W wielu różnych nadklasach mogą istnieć pola/metody o tych samych nazwach
 - które są wybierane do dziedziczenia?
- Gdy interpreter szuka pól i metod, które mają znaleźć się w podklasie kieruje się zasadami:
 - przeszukiwania „w głąb”
 - przeszukiwania „od lewej do prawej”

```
class NazwaPodkласy (Nadklaſa1, Nadklaſa2, ...):  
    ...
```



Dziedziczenie cykliczne

Niedozwolone!



Dziedziczenie w języku Python

- Dziedziczone są
 - metody
 - pola
- Obiekty podklasy są instancjami wszystkich nadklas



Hierarchia klas w języku Python

przykład polimorfizmu

```
class Animal :  
    pass  
  
class Mammal (Animal):  
    pass  
  
class Dog (Mammal):  
    pass  
  
a = Animal()  
m = Mammal()  
d = Dog()  
  
print(isinstance(a, Animal))  
print(isinstance(m, Mammal))  
print(isinstance(m, Animal))  
print(isinstance(d, Dog))  
print(isinstance(d, Mammal))  
print(isinstance(d, Animal))
```



Rozszerzenie nadklasy

```
class Animal :  
    def grow(self) :  
        print("Zwierz rośnie")  
class Dog(Animal) :  
    def bark(self) :  
        print("Pies szczeka")  
d = Dog()  
d.grow()  
d.bark()
```



Nadpisanie metody przykład polimorfizmu

```
class Animal :  
    def move(self) :  
        print("Zwierz się porusza")  
  
class Dog(Animal) :  
    def move(self) :  
        print("Pies chodzi i biega")  
  
  
a = Animal()  
d = Dog()  
a.move()  
d.move()
```



Funkcja **super()**

dostęp do nadpisanej metody

```
class Animal :  
    def move(self) :  
        print("Zwierz się porusza")  
  
class Dog(Animal) :  
    def move(self) :  
        print("Pies chodzi i biega")  
    def activate(self) :  
        self.move()  
        super().move()
```

```
d = Dog()  
d.activate()
```



Funkcja `super()`

dostęp do konstruktora nadklasy

```
class Animal :  
    def __init__(self, ileWazy) :  
        self.waga = ileWazy  
  
class Dog(Animal) :  
    def __init__(self, ileWazy, czyGryzie) :  
        super().__init__(ileWazy)  
        self.czyGryzie = czyGryzie  
  
d = Dog(15.5, False)  
print(d.waga, d.czyGryzie)
```



Programowanie obiektowe

HERMETYZACJA



UNIwersytet
EKonomiczny
W POZNANIU

Hermetryzacja

- Klasa jako „black box” (*czarna skrzynka*)
- Programista korzystający z zewnątrz z kodu klasy **nie ma bezpośredniego dostępu do wewnętrznych pól i metod**
- Dostęp do wewnętrznych pól i metod odbywa się **za pomocą przeznaczonych do tego metod publicznych**
 - sprzyja zachowaniu **poprawności przetwarzania wartości przechowywanych w polach**
 - redukuje ryzyko zewnętrznych **wywołań metod do tego nie przeznaczonych**



Hermetyzacja w języku Python

- Brak mechanizmu gwarantującego 100% hermetyzację
- Dobra praktyka
- **Pojedynczy prefiks _**
 - sugeruje, że pole/metoda jest jedynie do wewnętrznego użycia
 - w ramach modułu – jedynie sugestia
 - pomiędzy modułami – brak importu
- **Podwójny prefiks __**
 - powoduje doklejenie z przodu do nazwy pola/metody łańcucha _NazwaKlasy
 - ...uniemożliwiając nieświadome odwołanie



Hermetyzacja – przykład

```
class Kolo :  
    def __init__(self, pr, x, y):  
        self.__promien = pr  
        self._pozycjaX = x  
        self._pozycjaY = y  
  
    def liczPole(self):  
        return math.pi * self.__promien**2  
  
    def liczObwod(self):  
        return 2 * math.pi * self.__promien  
  
    def zwrocPromien(self):  
        return self.__promien  
  
k = Kolo(2.0, 100, 100)  
print(k.liczPole())  
print(k._pozycjaX) #niezalecane  
print(k.zwrocPromien()) #zalecane  
print(k.__promien) #błąd
```





UNIWERSYTET
EKONOMICZNY
W POZNANIU

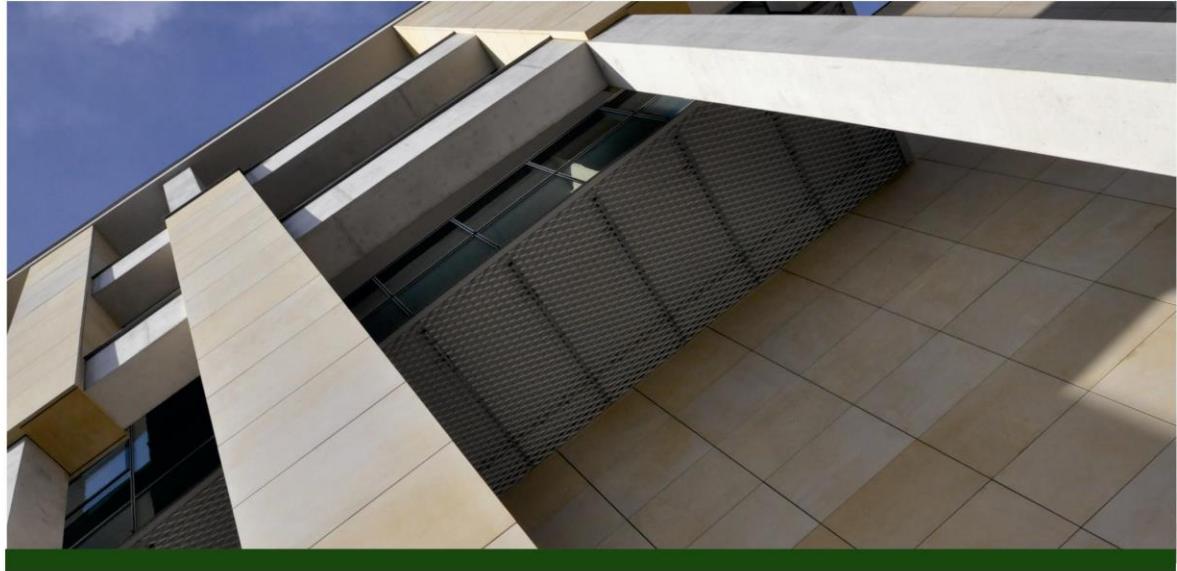


Dziękuję!

Do zobaczenia za tydzień



UNIWERSYTET
EKONOMICZNY
W POZNANIU



Obsługa sytuacji wyjątkowych

Programowanie obiektowe

dr hab. inż. Adam Wójtowicz, prof. UEP

Ariane 5

- 4 czerwca 1996, pierwszy lot
 - 40 sekund
 - \$500 000 000 strat
- Powód?
 - Błąd w oprogramowaniu
 - Zdarzyło się coś **wyjątkowego**



Wyjątek jest zdarzeniem
występującym w trakcie działania programu,
które uniemożliwia programowi
dalsze normalne działanie



Przykład wyjątku (1/4)

```
lista = [4, 6, 2]  
element = lista[3]
```

Traceback (most recent call last):

```
  File "E:/dydaktyka/PO/wyklad06/wyklad06.py",  
line 2, in <module>  
      element = lista[3]  
IndexError: list index out of range
```



Przykład wyjątku (2/4)

x = 4.5/0.0

Traceback (most recent call last):

```
  File "E:/dydaktyka/PO/wyklad06/wyklad06.py",  
line 4, in <module>
```

```
    x = 4.5/0.0
```

```
ZeroDivisionError: float division by zero
```



Przykład wyjątku (3/4)

```
lancuch = "wynik to: " + 3
```

Traceback (most recent call last):

```
  File "E:/dydaktyka/PO/wyklad06/wyklad06.py",  
line 6, in <module>
```

```
    lancuch = "wynik to: " + 3
```

```
TypeError: can only concatenate str (not "int") to str
```



Przykład wyjątku (4/4)

$y = x + 5$

Traceback (most recent call last):

```
  File "E:/dydaktyka/PO/wyklad06/wyklad06.py",  
line 8, in <module>  
    y = x + 5  
NameError: name 'x' is not defined
```



Błędy składniowe a wyjątki

- Błędy składniowe (ang. syntax error) – są identyfikowane i zgłasiane na etapie parsingu kodu źródłowego
 - np. brak dwukropka w instrukcji warunkowej
 - brak błędów składniowych warunkiem uruchomienia programu
 - ale brak błędów składniowych nie gwarantuje braku błędów w trakcie uruchomienia programu
- Wyjątki (ang. exception) – są identyfikowane i obsługiwane w czasie uruchomienia programu (ang. runtime errors)
 - wyjątki wbudowane w język
 - wyjątki definiowane przez programistę

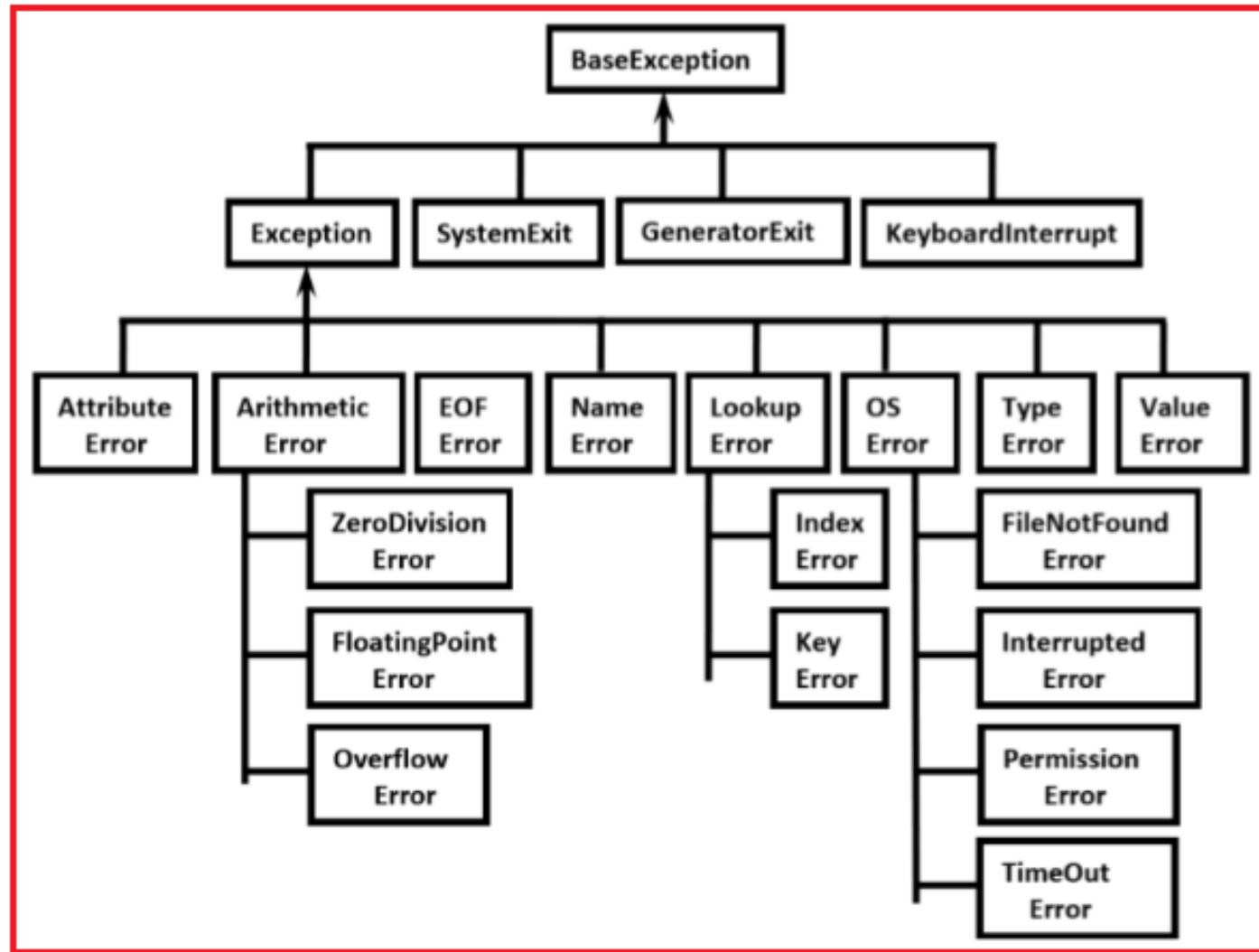


Wyjątki w środowisku Python

- Środowisko Python wychwytuje sytuacje wyjątkowe
 - tworzy obiekty z opisem przyczyny wyjątku
- W momencie wystąpienia sytuacji wyjątkowej
 - tworzony jest obiekt zawierający
 - komunikat błędu
 - wskazanie gdzie w kodzie błąd wystąpił (tzw. stack trace)
 - czasami również kod błędu
 - typ obiektu zazwyczaj identyfikuje dany typ sytuacji



Hierarchia wyjątków w języku Python



Programowanie obiektowe

WYJĄTKI – OBSŁUGA



UNIwersytet
EKONOMICZNY
W POZNANIU

Obsługa wyjątków

- Wystąpienie sytuacji wyjątkowej
 - kończy działanie instrukcji i całego bloku...
 - ...i uruchamia „blok obsługi wyjątku”

Blok obsługi wyjątku jest blokiem kodu uruchamianym w reakcji na wystąpienie wyjątku danego typu

- Można zadeklarować wiele bloków obsługi wyjątku...
- ...ale tylko co najwyżej jeden zostanie uruchomiony



Obsługa wyjątków w języku Python (1)

- Składnia – wariant 1: **try** – **except**

```
try :  
    <blok-testowanego-kodu>  
except <krotka-wyjątków> :  
    <blok-obsługi-wyjątków>  
    ...
```

- blok **try**
 - <blok-testowanego-kodu> dowolny kod, który potencjalnie generuje wyjątki
- jeden lub więcej bloków **except** zawierających:
 - opcjonalnie <krotka-wyjątków> – jedna klasa wyjątku albo krotka zawierająca nazwy klasy wyjątków
 - <blok-obsługi-wyjątków> instrukcje, które zostaną wykonane w przypadku pojawienia się w bloku **try** dowolnego wyjątku wskazanego w <krotka-wyjątków>



Przykład obsługi wyjątku (1)

```
lista = [4, 6, 2]
n=0
try :
    print(lista[n])
    print("Nie było błędu.")
except IndexError :
    print("Nie da się pobrać. Sprawdź czy lista ma tyle elementów.")
print("Koniec programu.")
```

4

Nie było błędu.
Koniec programu.



Przykład obsługi wyjątku (2)

```
lista = [4, 6, 2]
n=3
try :
    print(lista[n])
    print("Nie było błędu.")
except IndexError :
    print("Nie da się pobrać. Sprawdź czy lista ma tyle elementów.")
print("Koniec programu.")
```

Nie da się pobrać. Sprawdź czy lista ma tyle elementów.
Koniec programu.



Przykład obsługi wyjątku (3)

```
lista = [4, 6, 2]
n=3
try :
    print(lista[n])
    print("Nie było błędu.")
except TypeError :
    print("Nie da się pobrać. Sprawdź czy lista ma tyle elementów.")
print("Koniec programu.")
```

Traceback (most recent call last):

```
File "E:/dydaktyka/PO/wyklad06/wyklad06.py", line 31, in <module>
    print(lista[n])
```

IndexError: list index out of range



Przykład obsługi wyjątku (4)

```
lista = [4, 6, 2]
n=3
try :
    print(lista[n])
    print("Nie było błędu.")
except TypeError :
    print("Błędny typ danych.")
except IndexError :
    print("Nie da się pobrać. Sprawdź czy lista ma tyle elementów.")
print("Koniec programu.")
```

Nie da się pobrać. Sprawdź czy lista ma tyle elementów.
Koniec programu.



Przykład obsługi wyjątku (5)

```
lista = [4, 6, 2]
n="0"
try :
    print(lista[n])
    print("Nie było błędu.")
except TypeError :
    print("Błędny typ danych.")
except IndexError :
    print("Nie da się pobrać. Sprawdź czy lista ma tyle elementów.")
print("Koniec programu.")
```

Błędny typ danych.

Koniec programu.



Przykład obsługi wyjątku (6)

```
lista = [4, 6, 2]
n="0"
try :
    print(lista[n])
    print("Nie było błędu.")
except (TypeError, IndexError) :
    print("Jakiś błąd w przetwarzaniu listy.")
print("Koniec programu.")
```

Jakiś błąd w przetwarzaniu listy.

Koniec programu.



Przykład obsługi wyjątku (7)

```
lista = [4, 6, 2]
n="0"
try :
    print(lista[n])
    print("Nie było błędu.")
except :
    print("Jakiś błąd.")
print("Koniec programu.")
```

Jakiś błąd.

Koniec programu.



Obsługa wyjątków w języku Python (2)

- Składnia – wariant 2: **try** – **except** – **else**

```
try :  
    <blok-testowanego-kodu>  
except <krotka-wyjątków> :  
    <blok-obsługi-wyjątków>  
  
...  
else :  
    <blok-obsługi-braku-wyjątku>
```

- blok **try**
- jeden lub więcej bloków **except**
- blok **else**
 - <blok-obsługi-braku-wyjątku> instrukcje, które zostaną wykonane tylko jeżeli żaden wyjątek nie pojawił się w bloku **try**



Przykład obsługi wyjątku (8)

```
lista = [4, 6, 2]
n=0
try :
    print(lista[n])
except IndexError :
    print("Nie da sie pobrać. Sprawdź czy lista ma tyle elementów.")
else :
    print("Nie było błędu.")
print("Koniec programu.")
```

4

Nie było błędu.

Koniec programu.



Obsługa wyjątków w języku Python (3)

- Składnia – wariant 3: **try** – **except** – **finally**

```
try :  
    <blok-testowanego-kodu>  
except <krotka-wyjątków> :  
    <blok-obsługi-wyjątków>  
  
...  
finally :  
    <blok-zakończenia>
```

- blok **try**
- zero, jeden lub więcej bloków **except**
- blok **finally**
 - <blok-zakończenia> instrukcje, które zostaną wykonane jednokrotnie, na zakończenie, niezależnie od ewentualnego pojawienia się wyjątku w bloku **try**
 - instrukcja zakończenia (break, continue, return) w bloku **try** nie zapobiega wykonaniu bloku **finally**



Przykład obsługi wyjątku (9)

```
lista = [4, 6, 2]
n="0"
try :
    print(lista[n])
except IndexError :
    print("Nie da sie pobrać. Sprawdź czy lista ma tyle elementów.")
finally :
    print("Sprzątanie...")
print("Koniec programu.")
```

Sprzątanie...

Traceback (most recent call last):

```
  File "E:/dydaktyka/PO/wyklad06/wyklad06.py", line 92, in <module>
    print(lista[n])
```

TypeError: list indices must be integers or slices, not str



Obsługa wyjątków w języku Python (4)

- Obiekty klasy wyjątku mogą być przekazywane do bloku **except** za pomocą konstrukcji

```
except <klasa-wyjątku> as <zmienna-objektu>  
      <blok-obsługi-wyjątku>
```

- W <blok-obsługi-wyjątku> można uzyskać dostęp do danych obsługiwanyego wyjątku (w tym do jego parametrów) za pomocą zmiennej <zmienna-objektu>



Przykład obsługi wyjątku (10)

```
lista = [4, 6, 2]
n="0"
try :
    print(lista[n])
    print("Nie było błędu.")
except Exception as blad:
    print("Typ błędu: " + str(type(blad)))
    for arg in blad.args :
        print("Parametr: " + arg)
print("Koniec programu.")
```

```
-----  
Typ błędu: <class 'TypeError'>  
Parametr: list indices must be integers or slices, not str  
Koniec programu.
```



Obsługa wyjątku poza funkcją

- Wyjątek nie musi być „złapany” i obsłużony w kodzie funkcji/metody, w której się pojawił
- Jeżeli nie został obsłużony w kodzie funkcji, jest przekazywany „wyżej”, tj. do bloku, który zawiera wywołanie tej funkcji
 - i dalej, jeżeli tam też nie jest obsługiwany a wywołanie funkcji też jest w funkcji...



Przykład obsługi poza funkcją

```
def dziel(dzielna,dzielnik) :  
    return dzielna/dzielnik  
  
x=5.0  
y=0.0  
  
try :  
    dziel(x/y)  
except ZeroDivisionError :  
    print("Pamiętaj cholero nie dzielić przez zero")
```



Programowanie obiektowe

WYJĄTKI – GENEROWANIE



UNIWERSYTET
EKONOMICZNY
W POZNANIU

Generowanie wyjątków

- Poza wyjątkami generowanymi przez środowisko są również wyjątki generowane jawnie przez programistę w kodzie źródłowym
- Składnia
`raise <Wyjątek>`
- Gdzie `<Wyjątek>` oznacza
 - nazwę **klasy wyjątku**
 - lub wyrażenie, którego wartością jest **obiekt klasy wyjątku**
- Powstaje **obiekt**



Przykład generowania wyjątku

```
flagaOpuszczona = True
try :
    if flagaOpuszczona :
        raise ConnectionError("Flaga opuszczona",
                              "The flag is down")
except ConnectionError as e :
    for arg in e.args :
        print("Parametr: " + arg)
```



Wyjątki łańcuchowe

- Generowanie wyjątku w kodzie obsługującym wystąpienie innego wyjątku
- Inaczej: „opakowanie” wyjątku kolejnym
- Przykład:

```
lista = [4, 6, 2]
n="0"
try :
    print(lista[n])
    print("Nie było błędu.")
except TypeError as blad:
    lan=""
    for arg in blad.args :
        lan += (arg + " ")
    raise Exception("Zły typ", lan)
print("Koniec programu.")
```



Programowanie obiektowe

WYJĄTKI – DEFINIOWANIE



UNIWERSYTET
EKONOMICZNY
W POZNANIU

Definiowanie własnego typu wyjątku

- Wyjątki są generowane w oparciu o klasy wyjątków
- Można definiować **własne klasy wyjątków**
 - specyficzne dla danej aplikacji
 - z dodatkowymi danymi (polami)
 - z dodatkowym „zachowaniem” (metodami)
- Deklaracja przez dziedziczenie z klasy **Exception** (bezpośrednio lub pośrednio)



Przykład własnego wyjątku

```
class BladTrojkata (Exception) :  
    opis = "Dla dowolnego trójkąta miara każdego boku \n  
musi być mniejsza lub równa sumie miar dwóch pozostałych"  
  
a=1  
b=5  
c=10  
  
try :  
    if (a+b<c or a+c<b or b+c<a) :  
        raise BladTrojkata  
except BladTrojkata as e :  
    print(e.opis)
```



Programowanie obiektowe

WYJĄTKI – DLACZEGO



UNIWERSYTET
EKONOMICZNY
W POZNANIU

Dlaczego wyjątki?

- Zalety korzystania z mechanizmu wyjątków:
 - **Przekazywanie błędów w górę** stosu wywołań
 - **Grupowanie i rozróżnianie** różnych typów błędów
 - **Oddzielenie obsługi błędów** od zwykłego kodu



Separacja kodu i błędów

```
readFile :  
    open the file  
    determine its size  
    allocate that much memory  
    read the file into memory  
    close the file
```

- Potencjalne błędy:
 - Co się stanie, jeżeli nie mogę otworzyć pliku?
 - Co się stanie, jeżeli nie mogę określić jego wielkości?
 - Co się stanie, jak nie będę mógł przydzielić pamięci?
 - Co się stanie, jak nie będę mógł odczytać pliku?
 - Co się stanie, jeżeli nie będę mógł zamknąć pliku?



Separacja kodu i błędów

```
def readFile () :
    errorCode = 0

    open the file
    if theFileIsOpen :
        determine the length of the file
        if gotTheFileLength :
            allocate that much memory
            if gotEnoughMemory :
                read the file into memory
                if readFailed : errorCode = -1;
                else : errorCode = -2
            else : errorCode = -3
        close the file
        if (theFileDidntClose and (errorCode == 0)) :
            errorCode = -4
        else : errorCode = errorCode and -4
    else errorCode = -5
return errorCode
```



Separacja kodu i błędów

```
def readFile () :  
    try :  
        open the file  
        determine its size  
        allocate that much memory  
        read the file into memory  
        close the file  
    except fileOpenFailedException :  
        doSomething  
    except sizeDeterminationFailedException :  
        doSomething  
    except memoryAllocationFailedException :  
        doSomething  
    except readFailedException :  
        doSomething  
    except fileCloseFailedException :  
        doSomething
```





UNIWERSYTET
EKONOMICZNY
W POZNANIU



Dziękuję!

Do zobaczenia za tydzień