



Python

Cechy Pythona

- ❑ Dynamiczny obiektowy język programistyczny, który jest łatwy w nauce i można go wykorzystać do tworzenia różnorakiego oprogramowania.
- ❑ Rozprowadzany jest na otwartej licencji umożliwiając także zastosowanie do zamkniętych komercyjnych projektów.
- ❑ Jest aktywnie rozwijany i posiada szerokie grono użytkowników na całym świecie.

Zastosowania Pythona

- ☐ Programowanie interfejsów
 - ☐ **Interfejsy graficzne:** pyGTK, PyQt, wxPython, pyKDE, pyGNOME, pyFLTK, FxPy, Tkinter
 - ☐ **Interfejsy tekstowe:** curses
- ☐ Programowanie stron internetowych
 - ☐ Django, Flask, Pyramid, Bottle, Zope2, Web2Py, Web.py
- ☐ Programowanie gier
 - ☐ Biblioteka PyGame, pySDL2, Panda3D
- ☐ Programowanie aplikacja mobilnych i multiplatformowych
 - ☐ Kivy

Zastosowania Pythona

- ☐ Analiza tekstu
 - ☐ **PLY**
- ☐ Obliczenia naukowe
 - ☐ **NumPy, SciPy, SimPy**
- ☐ Przetwarzanie grafiki
 - ☐ **Python Imaging Library (PIL)**
- ☐ Programowanie skryptów
 - ☐ **Gimp, Blender, VIM, Dia, XUL**

Kto używa Pythona

- ❑ Google, Yahoo, Nokia, IBM czy NASA wykorzystują Pythona w swoich wartych wiele milionów, czy też miliardów dolarów aplikacjach i projektach.
- ❑ Microsoft jak i Apple oferują pełne wsparcie dla Pythona w swoich systemach operacyjnych i platformach programistycznych.
- ❑ Wiele stron internetowych takich jak YouTube czy Grono napisane jest w Pythonie.



Zagadnienia wstępne

Komendy w Pythonie

- ❑ Zastosowanie linii komend do wykonywania pojedynczych poleceń interpretowanych przez Pythona.
- ❑ Zastosowanie dedykowanego IDE (w naszym przypadku Wing Python IDE) do wykonywania skryptów napisanych w języku Python.

Wiersz poleceń Pythona

- ❑ Aby uruchomić tryb interaktywny Pythona należy w konsoli (cmd.exe) wpisać polecenie *python*.
- ❑ Widoczny po uruchomieniu trybu interaktywnego Pythona znak zachęty `>>>` oznacza gotowość interpretera do wykonywania naszych poleceń.
- ❑ Aby sprawdzić czy interakcja rzeczywiście działa, wpiszmy "credits" i wciśniemy klawisz Enter.

Pierwsze polecenia w Pythonie

- Tryb interaktywny Pythona może być używany jako kalkulator, przykładowo:

```
>>> 2+2
```

```
4
```

```
>>> 4-1
```

```
3
```

```
>>> 2*3
```

```
6
```

```
>>> 4/2
```

```
2
```

```
>>> 2/0
```

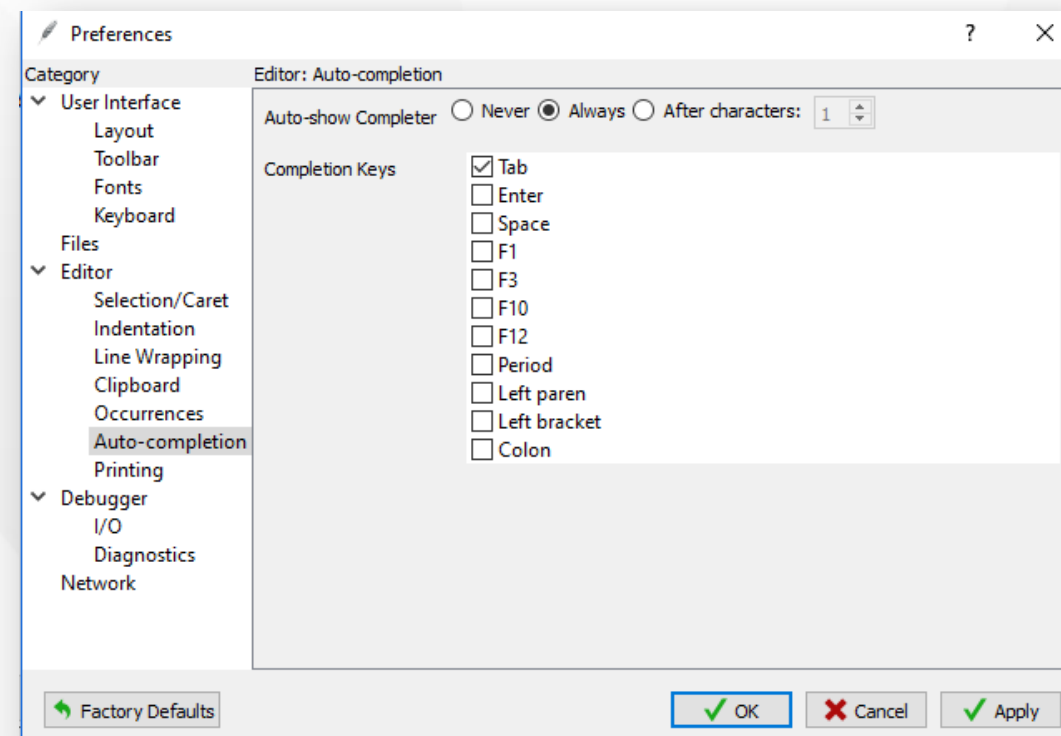
```
Traceback (most recent call last): File "<pyshell#6>", line 1, in -toplevel- 2/0  
ZeroDivisionError: integer division or modulo by zero.
```

Wing Python IDE

- ❑ Wing IDE jest zintegrowanym środowiskiem developerskim firmy Wingware której jest przeznaczone do programowania w języku Python.
- ❑ Wing IDE dostarcza narzędzi do debugowania, edycji, inteligentnego wspomaganie kodowania, refaktoryzacji kodu, testów jednostkowych, kontroli wersji oprogramowania, itp.
- ❑ Od tej pory będziemy głównie korzystali ze środowiska developerskiego Wing IDE.

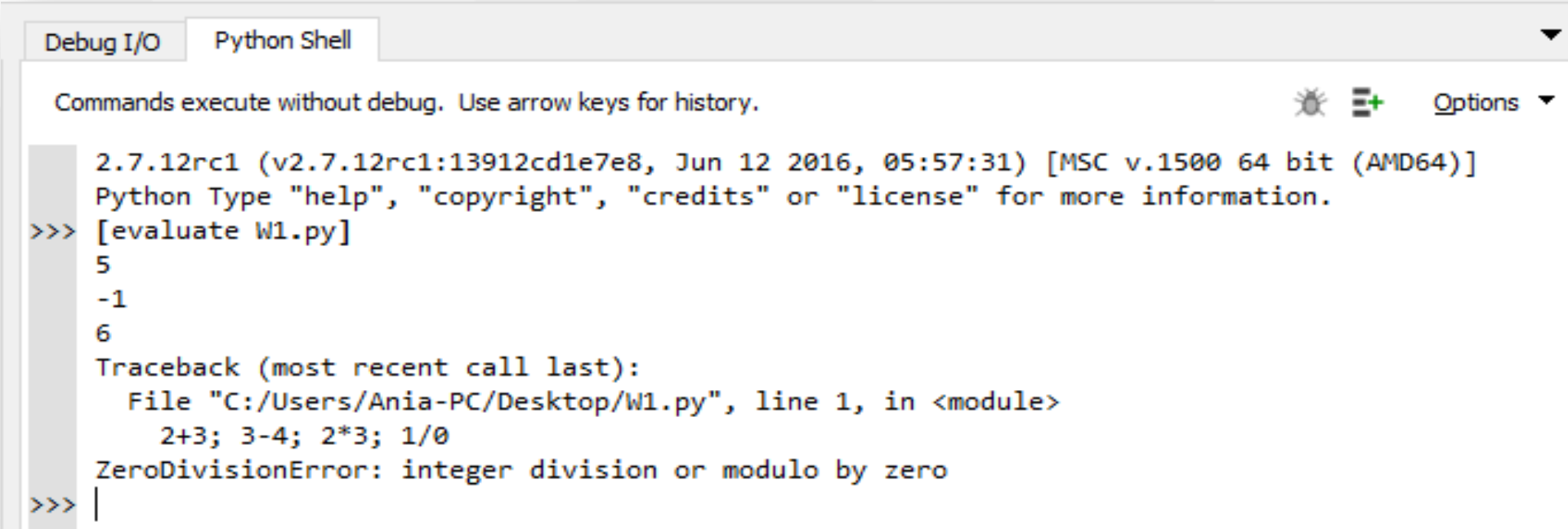
Konfiguracja Wing IDE

- ❑ Istotne jest ustawienie auto-podpowiedzi podczas wpisywania kodu w Wing IDE
 - ❑ Edit
 - ❑ Preferences
 - ❑ Editor -> Auto-completion
 - ❑ Auto-show Completer -> Always
 - ❑ Completion keys -> Tab



Przykładowy program w Wing IDE

2+3; 3-4; 2*3; 1/0



The screenshot shows the 'Python Shell' window in Wing IDE. It has tabs for 'Debug I/O' and 'Python Shell'. The shell contains the following text:

```
Commands execute without debug. Use arrow keys for history.
2.7.12rc1 (v2.7.12rc1:13912cd1e7e8, Jun 12 2016, 05:57:31) [MSC v.1500 64 bit (AMD64)]
Python Type "help", "copyright", "credits" or "license" for more information.
>>> [evaluate W1.py]
5
-1
6
Traceback (most recent call last):
  File "C:/Users/Ania-PC/Desktop/W1.py", line 1, in <module>
    2+3; 3-4; 2*3; 1/0
ZeroDivisionError: integer division or modulo by zero
>>> |
```



Zmienne

Czym jest zmienna w programie?

- ❑ Zmienne to takie wydzielone miejsca w pamięci komputera, gdzie możesz przechowywać dane.
- ❑ Python posiada kilka wbudowanych typów danych jak liczby całkowite, rzeczywiste, itp.
- ❑ Python jest językiem zorientowanym obiektowo i każda zmienna w Pythonie jest obiektem.
- ❑ Python nie wymaga ścisłej kontroli typów zmiennych.

Zasady nazewnictwa zmiennych

- ❑ Nazwy zmiennych w Pythonie mogą być dowolnie długie i mogą zawierać zarówno małe, jak i wielkie litery alfabetu, tak łacińskiego, jak i polskiego.
- ❑ Takie same nazwy, ale napisane małymi bądź dużymi literami, oznaczają różne zmienne.
- ❑ Nazwy zmiennych mogą zawierać znak podkreślenia (znak podkreślenia może rozpoczynać nazwę zmiennej).
- ❑ Nazwy zmiennych mogą zawierać cyfry.

Zasady nazewnictwa zmiennych

- ❑ Nazwy zmiennych nie mogą zawierać spacji.
- ❑ Zmiennym nie można nadawać nazw zastrzeżonych dla instrukcji języka Python (SyntaxError: invalid syntax).
- ❑ Cyfry nie mogą rozpoczynać nazwy zmiennej.

and	del	for	is	raise	assert	elseif	from
lambda	return	break	else	global	not	try	class
except	if	or	while	continue	exec	import	pass
yield	def	finally	In	print			

Deklaracja i przypisanie wartości do zmiennej

- ❑ Podczas deklaracji zmiennej nie trzeba podawać typu danych, jakie będzie przechowywać - z tego powodu mówi się, że Python jest językiem typowanym dynamicznie.
- ❑ Aby zadeklarować zmienną należy podać jej nazwę i przypisać jej wartość.

nazwaZmiennej = wartośćZmiennej

Np.:

```
a = 2
```

```
pi = 3.14
```

```
name = 'Michał'
```

Modyfikowanie wartości zmiennych

- ❑ W każdej chwili możemy zmienić wartość zmiennej.
- ❑ Wykonujemy to dokładnie tak samo jak byśmy przypisywali wartość do zmiennej z tą różnicą, że musimy podać taką samą nazwę istniejącej już zmiennej aby ją zmodyfikować.

`nazwaStarejZmiennej = nowaWartośćStarejZmiennej`

Np.:

```
a = 1
```

```
print(a)
```

```
a = 12
```

```
print(a)
```

Przypisywanie do zmiennej wartości innej zmiennej

- ❑ Do zmiennej można podstawić wartość innej zmiennej:

Np.:

```
a = 1
```

```
b = a      # b = 1
```

- ❑ Do zmiennej możemy również przypisać dowolne inne wyrażenie.

Usuwanie zmiennej

- ❑ Istniejącą zmienną można skasować za pomocą następującej instrukcji:

```
del nazwaZmiennej
```

Np.:

```
a = 5
```

```
del(a)
```

```
print(a)
```

```
NameError: name 'a' is not defined
```

- ❑ Podstawowym elementem programowania są instrukcje, czyli polecenia dla kompilatora określające w jaki sposób ma się zachować.
- ❑ W instrukcjach wykorzystuje się zmienne, wartości, funkcje, itp.
- ❑ Efektem działania instrukcji, jeśli nie zawiera błędów jest wynik zwracany przez kompilator.

Np.:

```
a = 1 + 2
```

```
b = 'Michał'
```

Wypisywanie wartości zmiennej na ekran

Aby wyświetlić jakąś wartość na ekranie należy wykorzystać wbudowaną funkcję Pythona:

```
print(nazwaZmiennej)
```

Np.:

```
a = 2
```

```
print(a)      # 2
```

Można też wyświetlić efekt jej działania jakiejś instrukcji:

```
print(instrukcja)
```

Np.:

```
print(2 - 1) # 1
```



Podstawowe typy zmiennych

Podstawowe typy zmiennych

- ❑ Zmienne możemy podzielić na typy w zależności od typów przechowywanych przez nie wartości, tj.:
 - ❑ Liczbowe,
 - ❑ Logiczne,
 - ❑ Napisowe,
 - ❑ None – brak typu/wartości.

Sprawdzanie typu

- Aby sprawdzić jakiego typu jest zmienna lub wartość należy wykonać instrukcję wykorzystującą wbudowaną funkcję Pythona:

`type(nazwaZmiennej)`

Np.:

```
a = 1.3
```

```
type(a)
```

```
type(21)
```

Typy liczbowe

- ❑ W Pythonie definiujemy następujące typy zmiennych liczbowych:
 - ❑ Liczby całkowite,
 - ❑ Długie liczby całkowite,
 - ❑ Liczby rzeczywiste,
 - ❑ Liczby zespolone.

- ❑ Liczby całkowite mają rozmiar 32 bitów, stąd największą wartością jaką mogą przyjąć jest 2147483647.
- ❑ Większe wartości zapamiętywane są jako długie liczby całkowite, rozpoznajemy je po umieszczonej na końcu literze L:
- ❑ Możemy też wymusić zapisanie liczby jako długiej liczby całkowitej dodając po jej wartości L.
- ❑ Jeżeli w wyrażeniu występuje choć jedna długa liczba całkowita, również rezultat jest długą liczbą całkowitą.

Liczby rzeczywiste

- ❑ Liczby rzeczywiste są to liczby zmiennoprzecinkowe (z separatorem w postaci kropki, nie przecinka!), czyli popularne ułamki dziesiętne.

- ❑ Mogą być również przedstawione w notacji naukowej (mantysa e + wykładnik):

Np.: **1e+3** **# 1000**

- ❑ Aby odrzucić z wyniku dzielenia część ułamkową należy użyć podwójnego znaku dzielenia:

Np.: **3.0//2.0** **# 1.0**

- ❑ Jeżeli w wyrażeniu występuje choć jedna liczba rzeczywista to rezultat jest liczbą rzeczywistą.

Zaokrąglanie wartości

- Zaokrąglanie liczb zmiennoprzecinkowych:

round(liczba, precyzja)

- **round()** – zaokrągla w górę

`round(3.14159, 2)`

`round(1.2), round(1.5), round(1.8)`

`round(-1.2), round(-1.5), round(-1.8)`

- **int()** – rzuca w dół

`int(1.2+0.5), int(1.5+0.5), int(1.8+0.5)`

`int(-1.2+0.5), int(-1.5+0.5), int(-1.8+0.5)`

Liczby zespolone

- ❑ Liczby zespolone są w Pythonie zapisywane jako suma części rzeczywistej (Re) i części urojonej (Im) oznaczonej przez dostawioną na jej końcu literę j (Re + Imj):

Np.: `-1+1j` `# (-1+1j)`

- ❑ Jeżeli w wyrażeniu występuje choć jedna liczba urojona, rezultat jest liczbą zespoloną:

Ósemkowy system liczbowy

- ❑ Ósemkowy system liczbowy – pozycyjny system liczbowy o podstawie 8.
- ❑ System ósemkowy jest czasem nazywany oktalnym od słowa octal. Do zapisu liczb używa się w nim ośmiu cyfr, od 0 do 7.

Np. 144 w systemie ósemkowym to dziesiętna liczba:

$$1 \times 8^2 + 4 \times 8^1 + 4 \times 8^0 = 64 + 32 + 4 = 100.$$

Szesnastkowy system liczbowy

- ❑ Szesnastkowy system liczbowy znany również pod nazwą system heksadecymalny – pozycyjny system liczbowy, w którym podstawą jest liczba 16.
- ❑ Do zapisu liczb w tym systemie potrzebne jest szesnaście znaków (cyfr szesnastkowych).

Np. E38 w systemie szesnastkowym to dziesiętna liczba:

$$3 \times 16^2 + 14 \times 16^1 + 8 \times 16^0 = 768 + 224 + 8 = 1000$$

Systemy liczbowe

- ❑ Liczby całkowite mogą być w Pythonie zapisywane w systemach pozycyjnych innych niż dziesiętny tj.: ósemkowym i szesnastkowym.
- ❑ Liczbę w systemie ósemkowym zapisujemy poprzedzając jej wartość dwuznakiem "0o":

Np.:

0o11

9

Uwaga! Różnica względem Python 2.X

- ❑ Liczbę w systemie szesnastkowym zapisujemy poprzedzając jej wartość dwuznakiem "0x":

Np.:

0x100

256

Zmienne logiczne

- ❑ Wartości logiczne (boolowskie) stanowią dwie stałe:
 - ❑ ***False***
 - ❑ ***True***
- ❑ Przypisywanie wartości logicznej do zmiennej odbywa się dokładnie tak samo jak w przypadku wartości arytmetycznych:

```
nazwaZmiennej = True [False]
```

Np.:

```
a = True           # True
```

Konwersja liczby do wartości logicznej

- ❑ W celu przekształcenia dowolnej wartości na wartość logiczną można wykorzystać funkcję wbudowaną `bool()`.

```
bool(nazwaZmiennej)
```

Np.:

```
a = 1
```

```
bool(a)
```

```
# True
```

- ❑ Konwersja do wartości:

• False	0	""	()	[]	{ }	None
• True	pozostałe					

Zmienne napisowe

- ❑ Zmienne przechowujące ciągi znaków.
- ❑ Przypisywanie napisów do zmiennej odbywa się podobnie jak w przypadku innych typów z tą różnicą, że napisy ograniczamy cudzysłowami bądź apostrofami.

```
nazwaZmiennej = "napis"
```

lub

```
nazwaZmiennej = 'inny napis'
```

Np.:

```
a = 'Michał'
```

```
b= 'Kruczkowski'
```

Znaki specjalne w napisie

- ❑ W napisach ograniczonych apostrofami możemy używać cudzysłowów i odwrotnie:

```
' "Tekst w cudzysłowiu" tekst poza '
```

- ❑ Napisy mogą ciągnąć się przez wiele linii jeżeli ograniczymy je potrójnymi cudzysłowami:

```
"""Ten
```

```
napis ma
```

```
wiele
```

```
linii"""      # 'Ten napis\nma\nwiele\nlinii'
```

Polskie znaki

- ❑ Domyślnie interpreter przyjmuje kodowanie ASCII dla plików *.py. Jeżeli stosujemy polskie znaki to musimy podać odpowiednie kodowanie. Zaleca się stosowanie kodowania utf-8 i zapisywanie plików z tym kodowaniem. Po zapisaniu pliku z tym kodowaniem należy dodać na początku pliku:

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

Operacje na napisach

- ❑ Polecenie `print` wypisuje swój argument (i przechodzi do nowej linii).

```
print('Jestem programem w Pythonie.')
```

```
print("Wypisuje na ekran różne napisy.")
```

- ❑ Jeśli podamy mu kilka argumentów (rozdzielonych przecinkami), wypisze je rozdzielone spacjami na wyjściu:

```
print('Jestem programem w Pythonie.', 'Wypisuje na ekran różne  
napisy.')
```

Powielanie i łączenie napisów

- ❑ Powielanie wyświetlenia wartości zmiennej napisowej odbywa się podobnie jak przemnożenie wartości arytmetycznej przez zwielokrotnienie:

```
nazwaZmiennej*3
```

- ❑ Łączenie (konkatenacja) napisów:

```
nazwaZmiennej + " " + nazwaZmiennej
```

Np.:

```
a = 'tekst'
```

```
print(a*3)
```

```
print(a + " " + a)
```

```
print ("To jest mój " + a)
```


Konwersja liczb na napisy

- Aby skonwertować liczbę na napis posługujemy się odwróconym apostrofem (klawisz nad tabulatorem):

```
a = 1.1
nazwaZmienej = `a`
lub
nazwaZmienej = str(a)
```

Np.:

```
a = 3.14
```

```
pi = `Pi = ` + `a`
```

```
type(pi)
```

Konwersja napisów na liczby

- ❑ Jeżeli liczby przechowywane są w postaci napisów można je skonwertować aby móc wykonać na nich operacje arytmetyczne.
- ❑ Służy do tego jedna z czterech wbudowanych funkcji:
 - ❑ ***int(x)***
 - ❑ ***long(x)***
 - ❑ ***float(x)***
 - ❑ ***complex(x)***

Np.

```
a = '3'
```

```
b = int(a)
```

```
type(a)
```

```
type(b)
```



Operator

Operatory arytmetyczne

<input type="checkbox"/> Dodawanie	+	(2 + 3)
<input type="checkbox"/> Odejmowanie	-	(6 - 5)
<input type="checkbox"/> Mnożenie	*	(2 * 2)
<input type="checkbox"/> Dzielenie	/	(3 / 3)
<input type="checkbox"/> Modulo – reszta z dzielenia	%	(4 % 3)
<input type="checkbox"/> Potęgowanie	**	(4 ** 2)

UWAGA! Kolejność wykonywania działań.

Błąd dzielenia przez zero

- ❑ UWAGA! Próba dzielenia przez zero kończy się w Pythonie wyświetleniem komunikatu o błędzie:

```
>>> 2/0
```

```
ZeroDivisionError: integer division or modulo by zero
```

Operatory logiczne

- ❑ Dostępne operatory
 - ❑ **AND**
 - ❑ **OR**

- ❑ Operatory AND i OR odpowiadają boolowskim operacjom logicznym, jednak nie zwracają one wartości logicznych. Zamiast tego zwracają którąś z podanych wartości.

UWAGA! Operacje logiczne mają najniższy priorytet spośród wszystkich operacji w Pythonie.

Operator AND

- ❑ Podczas używania AND wartości są oceniane od lewej do prawej.
- ❑ 0, "", [], (), {} i None są fałszem w kontekście logicznym, natomiast wszystko inne jest prawdą.
- ❑ Jeśli jakaś wartość jest fałszywa w kontekście logicznym, and zwraca pierwszą fałszywą wartość.

Np.:

```
>>> 'a' and 'b'      # 'b'
```

```
>>> '' and 'b'       # ''
```

```
>>> 'a' and 'b' and 'c'  # 'c'
```

Operator OR

- ❑ Wartości są oceniane od lewej do prawej.
- ❑ Jeśli jakaś wartość jest prawdą - OR zwraca tą wartość natychmiast.
- ❑ Jeśli wszystkie wartości są fałszem, OR zwraca ostatnią wartość.
- ❑ Zauważmy, że OR ocenia kolejne wartości od lewej do prawej, dopóki nie znajdzie takiej, która jest prawdą w kontekście logicznym, a pozostałą resztę ignoruje.

Np.:

`'a' or 'b'` `# 'a'`

`' ' or 'b'` `# 'b'`

`' ' or [] or {}` `# {}`

Operatory relacji

□ Dostępne operatory:

- > większy niż
- < mniejszy niż
- == równy względem
- >= większy lub równy względem
- <= mniejszy lub równy względem
- <> różny względem
- != różny względem
- is[not] sprawdzenie przynależności
- [not]in sprawdzenie tożsamości

Uwaga! Niedostępne w Python3.X

□ Wynikiem porównania jest wartość logiczna True dla prawdy lub False dla fałszu.

Porównywanie wartości

- ❑ Porównanie wartości tego samego typu zależy od typu:
 - ❑ Liczby są porównywane arytmetycznie,
 - ❑ Napisy są porównywane leksykograficznie,
 - ❑ Krotki i listy są porównywane leksykograficznie poprzez porównanie odpowiadających sobie elementów,
 - ❑ Słowniki są uznawane za równe, tylko, jeśli odpowiadające im listy par (klucz, wartość) również są równe,
 - ❑ W przypadku większości pozostałych typów wynikiem porównania jest nierówność, chyba, że po obu stronach porównania występuje ten sam obiekt.



Pierwszy interaktywny program

Komentarz jednoliniowy

- ❑ Komentarz to fragment kodu, który nie jest interpretowany przez kompilator, a który zazwyczaj służy do opisanie wykonywanych w programie operacji, stosowanych zmiennych, itp.
- ❑ Komentarze w języku Python poprzedzamy znakiem # (SHARP).

Przykładowy komentarz

- ❑ Komentarze mogą występować w dowolnej linii programu w Pythonie: cokolwiek pojawia się za znakiem # traktowane jest jako komentarz, aż do końca linii.

Komentarz blokowy

- ❑ Komentarze blokowe w języku Python poprzedzamy i kończymy znakami `"""`.

```
"""to również jest komentarz,  
który może ciągnąć  
się wiele linii"""
```

Wybrane znaki specjalne

Znak specjalny	Znaczenie
\\	Odwrócony ukośnik (\)
\'	Apostrof (')
\"	Cudzysłów (")
\a	ASCII Brzęczyk (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Przesunięcie wiersza (LF)
\r	ASCII Powrót kursora (CR)
\t	ASCII Tabulacja pozioma (TAB)
\v	Pionowa tabulacja (VT) w zestawie ASCII

Operacje wejścia

- ❑ Operacja wejścia umożliwia wprowadzenie danych z klawiatury przez użytkownika programu.
- ❑ W tym celu należy wykorzystać wbudowaną funkcję `input()`.
- ❑ Uwaga! W Python 2.X stosuje się funkcję `raw_input()`!

```
imie = input("Jak masz na imię?")  
  
print("Witaj", imie, "!")
```



Typy sekwencyjne

Czym są typy sekwencyjne

- ❑ Sekwencje są to specjalne zmienne, które pod jedną nazwą przechowują wiele uporządkowanych wartości.
- ❑ Przechowywane wartości są umieszczone w osobnych komórkach zmiennej i aby się do nich odwołać wykorzystuje się np. indeksy, czyli pozycje tych wartości.



Typy sekwencyjne (typ napisowy)

Typ napisowy

- ❑ Typ napisowy jest typem sekwencyjnym ze względu na wartości znakowe składające się na cały napis.
- ❑ Typ napisowy należy do typów niezmiennych, czyli nie może zmieniać swoich wartości.
- ❑ Konsekwencją niezmienności typu napisowego jest niemożność zmiany jego elementu.

Np.:

```
txt = 'napis'
```

```
txt[2]='w'
```

Indeks	0	1	2	3	4
Wartość	n	a	p	i	s

Metody napisowe

<code>s.capitalize()</code>	zmienia pierwszą literę na wielką
<code>s.center(długość)</code>	centruje napis w polu o podanej długości
<code>s.count(sub)</code>	zlicza wystąpienie podciągu sub w napisie s
<code>s.encode(kodowanie)</code>	zwraca zakodowaną wersję napisu ('utf-8', 'ascii', 'utf-16')
<code>s.isalnum()</code>	sprawdza czy wszystkie znaki są znakami alfanumerycznymi
<code>s.isdigit()</code>	sprawdza czy wszystkie znaki są cyframi
<code>s.islower()</code>	sprawdza czy wszystkie litery są małe

Metody napisowe

<code>s.isspace()</code>	sprawdza czy wszystkie znaki są białymi znakami
<code>s.isupper()</code>	sprawdza czy wszystkie litery są duże
<code>s.join(t)</code>	łączy wszystkie napisy na liście t używając s jako separatora
<code>s.lstrip()</code>	usuwa początkowe białe znaki
<code>s.replace(old, new)</code>	zastępuje stary podciąg nowym
<code>s.rstrip()</code>	usuwa końcowe białe znaki
<code>s.split(separator)</code>	dzieli napis używając podanego separatora
<code>s.strip()</code>	usuwa początkowe i końcowe białe znaki



Typy sekwencyjne (lista)

Lista

- ❑ Listę liczb można sobie wyobrazić jako zmienne ustawione w szereg.
- ❑ Każdej zmiennej jest przyporządkowany indeks, czyli oznaczenie jej miejsca w szeregu.
- ❑ Listy mogą zawierać dane dowolnego typu.
- ❑ Poza tym nie ma ograniczeń (oprócz tych sprzętowych) dla rozmiaru list.
- ❑ Dostęp do danych w liście uzyskujemy poprzez indeks każdego elementu.

Indeks	0	1	2	3	4
Wartość	2	K	Michał	0	-3.14

Wpisywanie wartości do listy

- ❑ Aby wpisać wartości do listy najpierw należy ją zadeklarować:

```
list = []
```

- ❑ Następnie stosujemy instrukcję pozwalającą wpisać wartość do kolejnych komórek listy

```
list.append(wartość)
```

lub do wybranych

```
list[indeks] = wartość
```


Wpisywanie wartości do listy

- ❑ Jeśli z góry wiemy jakie wartości mają się znaleźć w liście możemy je od razu wpisać podczas deklaracji

```
list = [1, 2, 3.14, 'tekst']
```

- ❑ Wypisanie wartości dla poszczególnych indeksów listy wymaga odwołania się do nich za pomocą indeksu

```
print list[1]
```

Podsumowanie tworzenia list

- ❑ Tworzenie listy:
 - ❑ `List = [wart1, wart2, ..., wartN]`
 - ❑ `ListInt = [1,2,3]`
 - ❑ `ListTxt = ["pierwszy", "ostatni"]`
 - ❑ `ListVar = [1.0, 2, "trzy"]`
 - ❑ `ListEmp = []`
 - ❑ `ListList = [[1, 2, 3], ["Nocny", "Dzienny"]]`
 - ❑ `ListOne = [1]`

Podsumowanie odczytywania list

- ❑ Odczytywanie elementów listy:
 - ❑ `List = [indeksElementu]`
 - ❑ `List = [indeksStart : indeksStop]`
 - ❑ `List = [indeksStart :: wielokrotność]`
 - ❑ **Wielopoziomowe?**
- ❑ Powielanie elementów listy:
 - ❑ `List *= 2`
 - ❑ `List = List * 3`
 - ❑ `List *= 0`
 - ❑ `[] *= 100`
- ❑ Określenie długości listy:
 - ❑ `len(List)`

Modyfikowanie list

- ❑ Skracanie:

- ❑ `List = List[indeksStart : indeksStop]`

- ❑ `List = List [: 2]`

- ❑ Wydłużanie:

- ❑ `List += [wartość]`

- ❑ `List += ['ostatni']`

- ❑ Zmiana wartości wybranych elementów:

- ❑ `List[indeksElementu] = nowaWartośćElementu`

- ❑ `List[0:2] = ["jeden", "dwa"]`

- ❑ Usuwanie wybranych elementów - del:

- ❑ `del List[indeksElementu]`

- ❑ `del List[1]`

Porównywanie list

- ☐ Porównywanie list:
 - ☐ jeżeli elementy obu list są sobie równe, listy są równe
 - ☐ jeżeli listy różnią się choć jednym elementem, to są nierówne
 - ☐ jeżeli pierwszy element pierwszej listy jest większy od pierwszego elementu drugiej listy, to pierwsza lista jest większa od drugiej
 - ☐ jeżeli pierwszy element pierwszej listy jest taki sam jak pierwszy element drugiej listy, decyduje porównanie drugich elementów, itd.
 - ☐ element nieistniejący jest zawsze mniejszy od każdego innego elementu

Operatory do porównywania list

- ❑ Czy dwie listy są równe? (==)
 - ❑ `nazwaListy1 == nazwaListy2`
 - ❑ `nazwaListy1 == [wart1, wart2, ..., wartN]`
- ❑ Czy dwie listy są różne? (!=)
- ❑ Czy pierwsza lista jest większa / mniejsza od drugiej? (> / <)
- ❑ Czy pierwsza lista jest większa / mniejsza bądź równa od drugiej? (>= / <=)

Sprawdzanie zawartości list

- ❑ Sprawdzanie zawartości odbywa się za pomocą operatorów `in` lub `not in` – sprawdzanie czy określona wartość znajduje się w liście:

- ❑ `wartość in List`
- ❑ `wartość not in List`
- ❑ `1 in List`
- ❑ `2 not in List`

Listy jako typ zmienny

- ❑ W Pythonie wszystkie sekwencje zmienne nie odnoszą się do określonych danych, ale do miejsca w pamięci, w którym te dane się znajdują.
- ❑ W związku z tym przypisanie listy do listy nie kopiuje wartości, a jedynie wskaźnik do nich!

```
Lista1 = Lista2
```

```
Lista1[1] = 12
```

- ❑ Jeżeli listę utworzono z innych list to każda zmiana ich wartości będzie przenoszona na listę nadrzędną. Co więcej, zmiana wartości listy nadrzędnej będzie przenoszona na listę podrzędną.

Podstawowe metody list

<code>list(s)</code>	konwertuje sekwencję <code>s</code> na listę
<code>s.append(x)</code>	dodaje nowy element <code>x</code> na końcu <code>s</code>
<code>s.extend(t)</code>	dodaje nową listę <code>t</code> na końcu <code>s</code>
<code>s.count(x)</code>	zlicza wystąpienie <code>x</code> w <code>s</code>
<code>s.index(x)</code>	zwraca najmniejszy indeks <code>i</code> , gdzie <code>s[i] == x</code>
<code>s.pop([i])</code>	zwraca <code>i</code> -ty element i usuwa go z listy.
<code>s.remove(x)</code>	odnajduje <code>x</code> i usuwa go z listy <code>s</code>
<code>s.reverse()</code>	odwraca w miejscu kolejność elementów <code>s</code>
<code>s.sort([funkcja])</code>	sortuje w miejscu elementy. "funkcja" to funkcja porównawcza



Typy sekwencyjne (krotki)

Krotki

- ❑ Krotki pod wieloma względami przypominają listy, w podobny sposób tworzymy je i sprawdzamy ich wartości.
- ❑ Krotki są *sekwencjami niezmiennymi*, co powoduje różnice w sposobie ich modyfikacji.
- ❑ W związku z tym przypisanie krotki do krotki kopiuje faktyczne wartości, a nie jedynie wskaźnik do nich.
- ❑ Krotki mogą zawierać elementy różnych typów w tym sekwencyjnych, również mogą być puste.

Tworzenie krotek

- ❑ Tworzenie krotki:
 - ❑ `Krotka = (wart1, wart2, ..., wartN)`
 - ❑ `Krotka = wart1, wart2, ..., wartN`
 - ❑ `Krotka = (1, 2, 3)`
 - ❑ `Krotka = (1.0, 2, "trzy")`
 - ❑ `Krotka = ()`
 - ❑ `Krotka = (12,)`
 - ❑ `Krotka2 = (Krotka1, Lista1)`

Odczytywanie krotek

- ❑ Odczytywanie elementów krotki:
 - ❑ `Krotka = [indeksElementu]`
 - ❑ `Krotka = [indeksStart : indeksStop]`
 - ❑ `Krotka = [indeksStart :: wielokrotność]`
 - ❑ **Wielopoziomowe?**
- ❑ Powielanie elementów krotki:
 - ❑ `Krotka *= 2`
 - ❑ `Krotka = Krotka * 3`
 - ❑ `Krotka *= 0`
 - ❑ `[] *= 100`
- ❑ Określenie długości krotki:
 - ❑ **`Len(Krotka)`**

Modyfikowanie krotek

- ❑ Skracanie:
 - ❑ `Krotka = Krotka[indeksStart : indeksStop]`
 - ❑ `Krotka = Krotka[: 2]`
- ❑ Wydłużanie:
 - ❑ `Krotka += (wartość)`
 - ❑ `Krotka += ('ostatni')`
- ❑ Zmiana wartości wybranych elementów:
 - ❑ krotki są sekwencjami niezmiennymi więc nie możemy modyfikować i usuwać elementów krotek



Typy sekwencyjne (słowniki)

- ❑ Kolejny specjalny typ zmiennej który zawiera *pary klucz-wartość*.
- ❑ Aby odwołać się do odpowiedniej wartości należy zamiast numeru indeksu, jak to było w listach podać klucz odpowiadający tej wartości.

Zapisywanie danych do słownika

- ❑ W słowniku dostęp do dowolnej przechowywanej wartości możliwy jest poprzez podanie klucza do niej.
- ❑ Słownik składa się zatem ze zbioru kluczy i zbioru wartości, gdzie każdemu kluczowi przypisana jest pojedyncza wartość.
- ❑ Zależność między kluczem a jego wartością nazywana bywa odwzorowaniem.
- ❑ Klucz nie musi być liczbą, tak jak jest nią indeks w listach czy krotkach, ale musi być typu niezmiennego.

Słownik = {klucz1 : wartość1, ..., kluczN : wartośćN}

Metody słownikowe

`vars()`

Zwraca słownik zawierający wszystkie dostępne zmienne

`len(Słownik)`

Zwraca ilość kluczy w słowniku

`Słownik[klucz]`

Zwraca wartość podanego klucza

`Słownik[klucz] = wartość`

Dopisuje nowy klucz

`Słownik[klucz] = nWartość`

Modyfikuje istniejącą wartość

`Słownik1 = Słownik2.copy()`

Kopiuje zawartość Słownik1 do Słownik2

`del Słownik[klucz]`

Usuwa wartość ze słownika

`Słownik.clear()`

Czyści cały słownik

Metody słownikowe

`Slownik1.update(Slownik2)`

Aktualizuje Slownik1 w oparciu o Slownik2

`Slownik.pop('taxi')`

Odczytuje i usuwa wartość ze słownika

`Slownik.popitem()`

Odczytuje i usuwa nieokreśloną wartość ze słownika

`'taxi' in Slownik`

Sprawdza występowanie określonego klucza w słowniku

`Slownik.has_key(„k1”)`

Sprawdza występowanie określonego klucza w słowniku

Słowniki właściwości

`Słownik.keys()`

Zwraca listę kluczy w słowniku

`Słownik.values()`

Zwraca listę wartości w słowniku

`999 in Słownik.values()`

Występowanie określonych wartości w słowniku

`str(Słownik)`

Konwertuje słownik na napis

`del Sownik`

Usuwa cały słownik



Typy sekwencyjne (zbiory)

- ❑ Nowy zbiór tworzymy używając słowa kluczowego **set()**, jako parametr możemy podać listę lub krotkę z elementami, które mają zostać na starcie dołączone do zbioru.
- ❑ Pierwszą różnicą jest numerowanie/indeksowanie. Zbiór jest tylko workiem na elementy, w przeciwieństwie do uporządkowanych krotek i list. Wartości **nie posiadają** swoich indeksów.
- ❑ Jest to największa zaleta zbiorów, ponieważ dzięki temu każda wartość występuje **dokładnie jeden raz**.
- ❑ Różnica druga, bezpośrednio wynikająca z pierwszej, to operacje na zbiorach.
- ❑ Mamy jednak dodatkowe metody, których nie omawialiśmy wcześniej, jak np.: metody pozwalające na działania boolowskie, czyli sumę zbiorów, różnicę i iloczyn.

Zapis danych do zbioru

- ❑ W Pythonie możemy tworzyć zbiory zmienne i niezmiennie:

```
A = set([1,2,3])           # zmienny  
B = frozenset([2,3,4])     # niezmienny
```

- ❑ Aby stworzyć zbiór pusty napiszemy:

```
C=set()                   # pusty
```

Zbiory zmienne - właściwości

- ❑ Zbiory zmienne mogą być powiększane i zmniejszane:

`A.discard(wartośćUsuwana)`

`A.add(wartośćDodawana)`

- ❑ Zbiory zmienne nie mogą być ani kluczami w słownikach ani elementami innych zbiorów.

Zbiory niezmiennie właściwości

- ❑ Zbiory niezmiennie nie mogą być ani zmniejszane ani powiększane.
- ❑ Zbiory niezmiennie mogą być kluczami w słownikach i elementami innych zbiorów.

Metody zbiorów

`len(a)`

Zwraca liczbę elementów zbioru

`2 [not]in A`

Sprawdza czy dany obiekt [nie]jest elementem zbioru

`set([1,3])<=A`

Sprawdza czy dany zbiór jest podzbiorem innego zbioru

`A.issubset(B)`

Sprawdza czy dany zbiór jest podzbiorem innego zbioru

`A>=set([1,3])`

Sprawdza czy dany zbiór jest nadzbiorem innego zbioru

`A.issuperset(B)`

Sprawdza czy dany zbiór jest nadzbiorem innego zbioru

Operacje na zbiorach

$$D = A \mid B$$

Połączenie dwóch zbiorów A i B

$$E = A \& B$$

Część wspólna dwóch zbiorów A i B

$$F = A - B$$

Różnica dwóch zbiorów A i B (typ wynikowy jest typem pierwszego zbioru)

$$G = A \wedge B$$

Różnica symetryczna dwóch zbiorów A i B

Konwersja typów sekwencyjnych

- ❑ W konwersji typów sekwencyjnych używamy następujących instrukcji:
 - ❑ list zamienia typ sekwencyjny na listę
 - ❑ `Lista = list(krotka)`
 - ❑ tuple zamienia typ sekwencyjny na krotkę
 - ❑ `Krotka =tuple(lista)`
 - ❑ str zamienia typ sekwencyjny na napis
 - ❑ `str(Krotka)`
 - ❑ `str(Lista)`

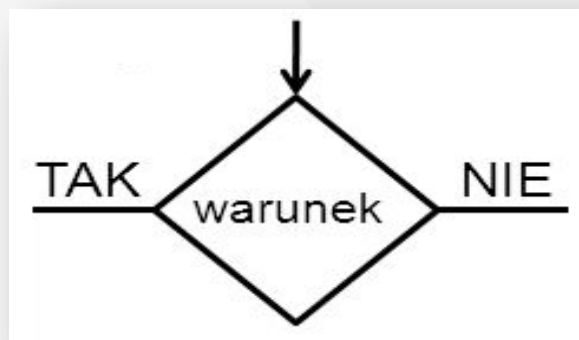


Instrukcje warunkowe

Reaktor

Do czego są potrzebne instrukcje warunkowe

- ❑ Rozwiązują problemy decyzyjne polegające na wyborze jednej opcji – która spełnia lub nie spełnia dany warunek.
- ❑ Brak możliwości wielokrotnego sprawdzania tego samego warunku – brak sprzężeń zwrotnych.



Instrukcje warunkowe w Pythonie

❑ Dostępne instrukcje warunkowe:

❑ **if**

❑ **if-else**

❑ **if-elif-[else]**

Uwaga! W wersji 2.X – elseif!

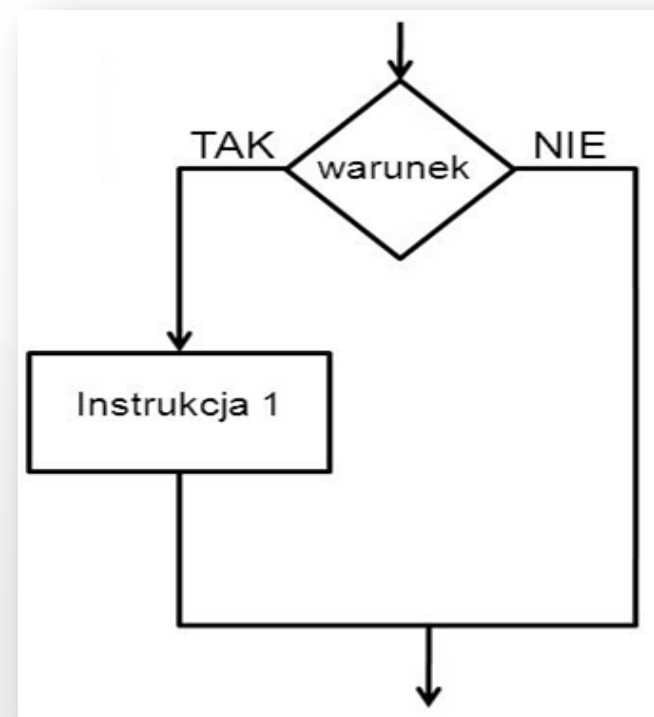
❑ Podobne działanie do instrukcji warunkowych możemy osiągnąć za pomocą wyrażeń trójargumentowych.

Uwaga! W Pythonie nie ma instrukcji switch ... case, ale można ją zastąpić zagnieżdżonymi instrukcjami if ... elif.

Instrukcja if

□ Instrukcja if:

```
if [warunek]:  
    [Instrukcja1]
```

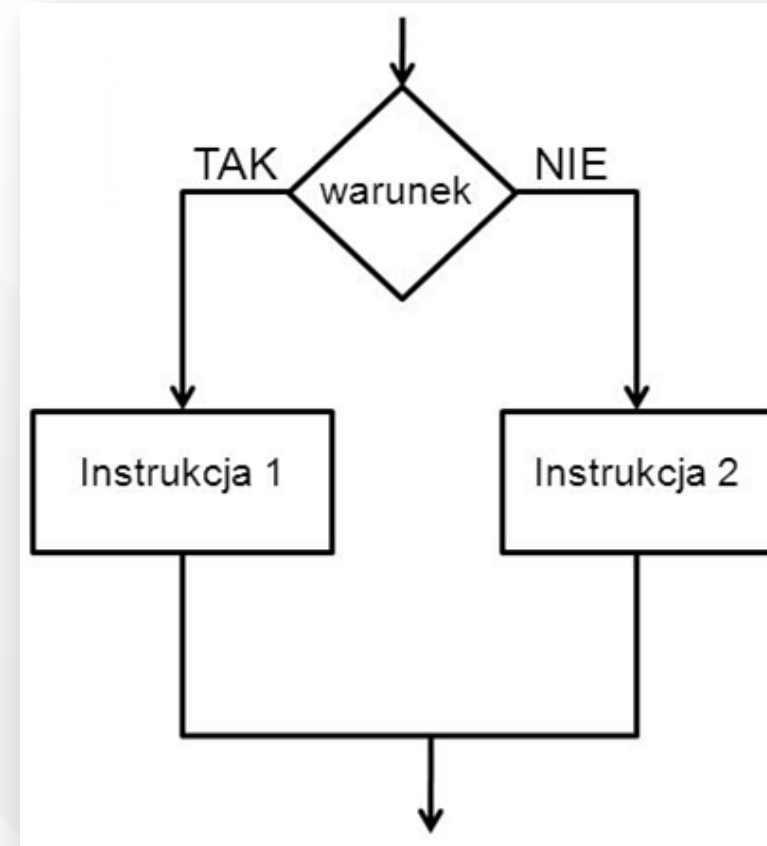


Uwaga! W Pythonie wcięcia mają znaczenie – blok kodu, który ma być wykonany, jeśli warunek jest spełniony, oznaczamy wcięciem. Koniec wcięcia to koniec bloku, a koniec wiersza kończy instrukcję.

Instrukcja if-else

- ❑ Instrukcja if -else:

```
if [warunek]:
    [Instrukcja1]
else:
    [Instrukcja2]
```



Instrukcja if-elif i if-elif-else

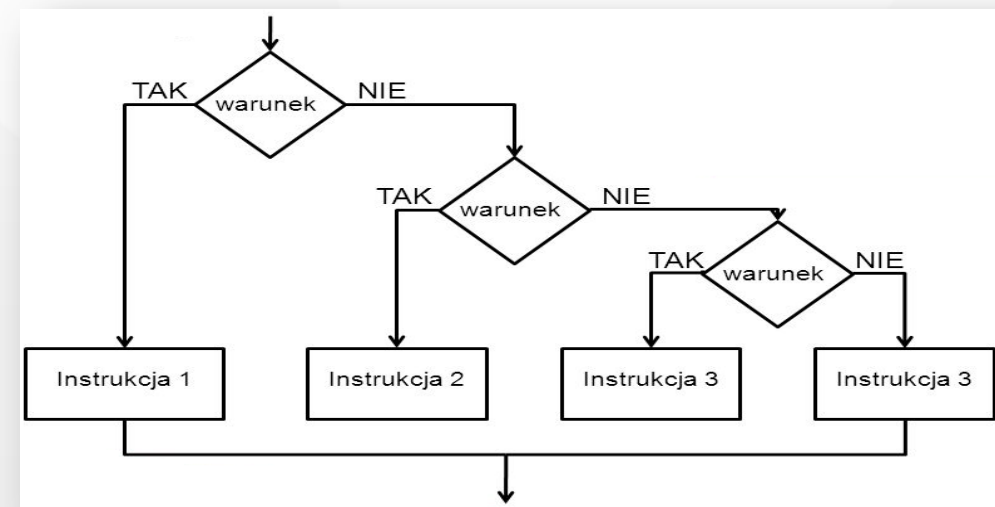
❑ Instrukcja if-elif

```
if [warunek1]:
    [Instrukcja1]
elif [warunek2]:
    [Instrukcja2]
```

❑ Instrukcja if-elif-else

```
if [warunek1]:
    [Instrukcja1]
elif [warunek2]:
    [Instrukcja2]

else:
    [instrukcja3]
```



Wyrażenie trójargumentowe

- ❑ W wyrażenie trójargumentowe if-else przypomina nieco operator trójargumentowy z języka C.
- ❑ Daje dokładnie taki sam efekt działania jak instrukcja if-else.

`[działanie war-tak] if [war] else [działanie war-nie]`

Np.:

```
a = 25 ; b = 30
```

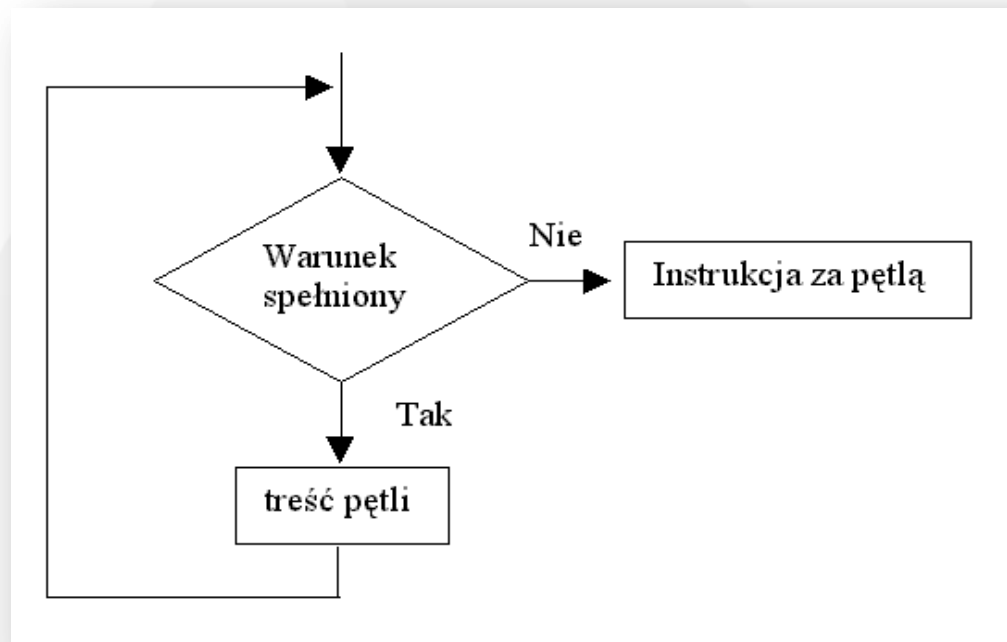
```
print(b) if (a < b) else print(a)
```



Petle

Do czego są potrzebne pętle?

- ❑ Pętle są wykorzystywane do iterowania (sprawdzania tego samego warunku) dla kolejnych przypadków – aż do ich wyczerpania.



Pętla while i while-else

- ❑ Składnia pętli while:

```
while [warunek]:  
    [instrukcje1]
```

- ❑ Składnia pętli while-else:

```
while [warunek]:  
    [instrukcje1]  
  
else:  
    [instrukcje2]
```

Przykład użycia pętli while

□ Pętla while:

```
licznik = 10
wartość = 15
while licznik <= wartość:
    licznik += 1
    print("Jestem w while.")
```

Przykład użycia pętli while-else

- ❑ Pętla typu while może również zawierać blok po else, wykonywany po ostatnim obiegu pętli:

```
a=7

while a:
    a-=1
    print(a)

else:
    print("koniec")
```


Pętla for-in stosowana do list

- ❑ Pętla pozwala wykonać operacje na wszystkich lub wybranych elementach sekwencji

```
for a in lista:  
  
    print(a)
```

- ❑ gdzie a jest przykładową zmienną, która w danym powtórzeniu pętli przyjmuje wartość kolejnych elementów sekwencji.

```
for indeks, wartość in enumerate(lista):  
  
    print(indeks, wartość)
```

Pętla for-in stosowana do słowników

- ❑ Pętla również pozwala wykonać operacje na wszystkich lub wybranych elementach słowników

```
Slownik = {"key1": "var1", "key2": "var2", "key3": "var3"}  
  
for key in Slownik:  
    print(key, Slownik[key])
```

Zagnieżdżenia

- ❑ Instrukcja warunkowa wewnątrz pętli
- ❑ Pętla wewnątrz pętli
- ❑ Pętla wewnątrz instrukcji warunkowej
- ❑ Instrukcja warunkowa wewnątrz instrukcji warunkowej, itd.

```
for i in lista:  
    if i>0:  
        print(i, i**0.5)
```

Szybkie tworzenie sekwencji

- ❑ Do tworzenia sekwencji, których elementy należą do ciągu arytmetycznego, używamy funkcji `range()`:

```
range(10)                # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- ❑ Aby wyświetlić kwadraty liczb od 0 do 9, napiszemy:

```
for x in range(10):  
    print(x, '** 2 =', x*x)
```

Szybkie tworzenie sekwencji

- Aby zmienić pierwszy element tworzonej sekwencji używamy funkcji range z dwoma parametrami (początek i koniec):

```
range(1,10)          # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Aby zmienić krok pomiędzy elementami tworzonej sekwencji używamy funkcji range z trzema parametrami (początek, koniec i krok):

```
range(1,10,2)        # [1, 3, 5, 7, 9]
```

Formatowanie liczb

- ❑ Aby liczby wyświetlane były w należyty sposób i w pożądanym miejscu używamy operatora formatowania % w połączeniu z ciągiem formatującym.
- ❑ Ciąg formatujący składa się ze znaku %, po którym następują opcje formatowania, ilość znaków przeznaczonych do wyświetlenia oraz typ danej do wyświetlenia (przy czym tylko trzeci element – tj. typ danych jest wymagany).
- ❑ Typ danej sygnalizujemy pojedynczą literą:
 - ❑ s oznacza napis (konwertuje każdy typ danych na tekst),
 - ❑ c oznacza pojedynczy znak w kodzie ASCII,
 - ❑ i oznacza dziesiętną liczbę całkowitą (konwertuje kompatybilny typ danych na liczbę całkowitą),
 - ❑ x oznacza szesnastkową liczbę całkowitą (konwertuje kompatybilny typ danych na liczbę całkowitą),
 - ❑ o oznacza ósemkową liczbę całkowitą (konwertuje kompatybilny typ danych na liczbę całkowitą),
 - ❑ e oznacza liczbę zmiennopozycyjną w postaci wykładniczej,
 - ❑ f oznacza liczbę zmiennopozycyjną w postaci ułamka dziesiętnego,

Formatowanie długości wyjścia z programu

- ❑ Formatowanie wyjścia:

```
for x in range(5,100,10):  
    print("%4i%6i%8i" % (x,x**2,x**3))
```

- ❑ Formatowanie liczb zmiennoprzecinkowych:

```
for x in range(5,100,10):  
    print("Pierwiastkiem liczby %2i jest %5.3f" % (x,x**0.5))
```

Wybrane opcje formatowania

- ❑ + wymusza wyświetlanie znaku liczby, także dla liczb nieujemnych:

```
for x in range (-10,11):  
    print("%+i" % x)
```

- ❑ # powoduje, że liczby ósemkowe i szesnastkowe będą poprzedzane właściwym prefiksem:

```
for x in range(5,100,10):  
    print("%3i%#6o%#5x" % (x,x,x))
```


Wybrane opcje formatowania

- ❑ - powoduje, że liczby będą wyrównywane do lewej, a nie prawej krawędzi swojego pola:

```
for x in range(5,100,10):  
    print "%-3i%#-6o%#-5x" % (x,x,x)
```

- ❑ 0 spowoduje, że pole przeznaczone na liczby będzie wypełniane nie spacjami, lecz zerami:

```
for x in range(5,100,10):  
    print "%3i %#04o %#04x" % (x,x,x)
```

Instrukcja continue

- continue – pomijanie instrukcji

```
liczby = [2,1,0,-1,-2]
for x in liczby:
    if x<0:
        continue
    print("Pierwiastkiem liczby %2i jest %5.3f" % (x,x**0.5))
```

Instrukcja break

- ❑ break – przerwanie instrukcji

```
liczby = input("Podaj kilka liczb:")  
  
szukana = input("Podaj liczbę do znalezienia:")  
  
for p,x in enumerate(liczby):  
    if x != szukana:  
        continue  
  
    print("Znaleziono liczbę %i na pozycji %i" % (x,p+1))  
  
    break  
  
else:  
  
    print("Liczby %i nie ma na liście" % szukana)
```



Funkcje

Czym są funkcje

- ❑ Dotąd korzystaliśmy z wbudowanych funkcji Pythona nie wymagających implementacji – wystarczyło je jedynie wywołać i poprawnie zastosować.
- ❑ Jak w każdym języku programowania Python również pozwala tworzyć programiście własne funkcje.

Definicja funkcji

- ❑ Definicja funkcji musi zawierać:
 - ❑ nagłówek funkcji obejmujący o nazwę funkcji, która pozwoli zidentyfikować funkcję w pozostałej części programu o listę argumentów, która funkcja otrzymuje na początku działania programu
 - ❑ ciało funkcji, zawierające instrukcje, które zostaną wykonane w momencie wywołania (użycia) funkcji o jeżeli funkcja ma zwracać jakiś rezultat, musi zawierać odpowiednią instrukcję
- ❑ Składnia funkcji:

```
def nazwaFunkcji ([parametry]):  
    [instrukcje]
```

Wywoływanie funkcji

- ❑ Wywoływanie funkcji odbywa się poza ciałem funkcji (w dowolnym miejscu w programie):

`nazwaFunkcji ([parametry])`

- ❑ Parametry do funkcji, możemy przekazywać albo podając ich wartości w kolejności podanej w nagłówku definicji funkcji, albo w dowolnej kolejności, wykorzystując ich nazwy.
- ❑ Parametry znajdujące się na początku listy i na właściwych sobie pozycjach, nie muszą mieć podanej nazwy.

Typy funkcji

- ❑ Zwracające wartość:

```
def nazwaFunkcji (parametry):  
    [instrukcje]  
  
    return wartośćZwracana
```

- ❑ Nie zwracające żadnej wartości:

```
def nazwaFunkcji (parametry):  
    [instrukcje]
```


Usuwanie i redefiniowanie funkcji

- ❑ Zdefiniowaną uprzednio funkcję możemy w dowolnym miejscu usunąć, posługując się instrukcją `del`:

```
del nazwaFunkcji
```

- ❑ Redefinicja funkcji nie wymaga jej usuwania – wystarczy od nowa ją zdefiniować pod tą samą nazwą.

Parametry formalne funkcji

- ❑ Występujący w nagłówku funkcji identyfikator n nazywamy parametrem formalnym. Jest to nazwa, pod którą przekazana do funkcji wartość widziana jest wewnątrz ciała funkcji.
- ❑ Parametr formalny jest szczególnym rodzajem zmiennej lokalnej (szczególnym, bo inicjalizowanym wartością podaną przy wywołaniu w nawiasach). Zmienna lokalna to każda zmienna inicjowana w obrębie funkcji. Tj.:
 - ❑ dostępna jest tylko w obrębie funkcji,
 - ❑ "przykrywa" zmienną o tej samej nazwie istniejącą poza funkcją,

Parametry aktualne funkcji

- ❑ Parametry aktualne to faktyczne wartości prze
- ❑ W momencie wywołania funkcji wszystkie operacje przewidziane do wykonania na parametrze formalnym, wykonywane są na parametrze aktualnym przekazanym do funkcji.

Funkcje wielo-parametrowe

- ❑ Funkcja może przyjmować więcej niż jeden argument i zwracać więcej niż jeden rezultat.

```
def nazwaFunkcji(p1, p2, ..., pN):  
    [instrukcje]  
  
    return wynik1, wynik2, ..., wynikN
```

- ❑ Wywołanie funkcji:

```
nazwaFunkcji(p1, p2, ..., pN)
```

Domyślne wartości parametrów funkcji

- ❑ Wywołanie funkcji wymaga podania wartości dla wszystkich parametrów - jeżeli nie podamy wartości dla wszystkich parametrów formalnych, wystąpi błąd.
- ❑ Możemy tego uniknąć, podając domyślne wartości argumentów.

```
def nazwaFunkcji(p1, p2 = "domyślny", p3 = 0):  
  
    print p1, p2, p3
```

- ❑ Parametry do funkcji, możemy przekazywać albo podając ich wartości w kolejności podanej w nagłówku definicji funkcji, albo w dowolnej kolejności, wykorzystując ich nazwy:

```
nazwaFunkcji(p1, p2 = "war2", p3 = 12)  
  
nazwaFunkcji(p1)
```

Funkcje z nieznaną liczbą parametrów

- ❑ Jeżeli w momencie definiowania funkcji nie jesteśmy w stanie określić liczby argumentów, które będą do niej przekazywane, poprzedzamy nazwę parametru formalnego oznaczającego wszystkie pozostałe argumenty funkcji gwiazdką:

```
def suma(*arg):  
    s=0  
    for x in arg:  
        s+=x  
    return print(s)
```

- ❑ Taka funkcja zadziała dla dowolnej liczby argumentów:

```
suma(1,2,3,4,5)
```



Programowanie obiektowe

Python a obiektowość

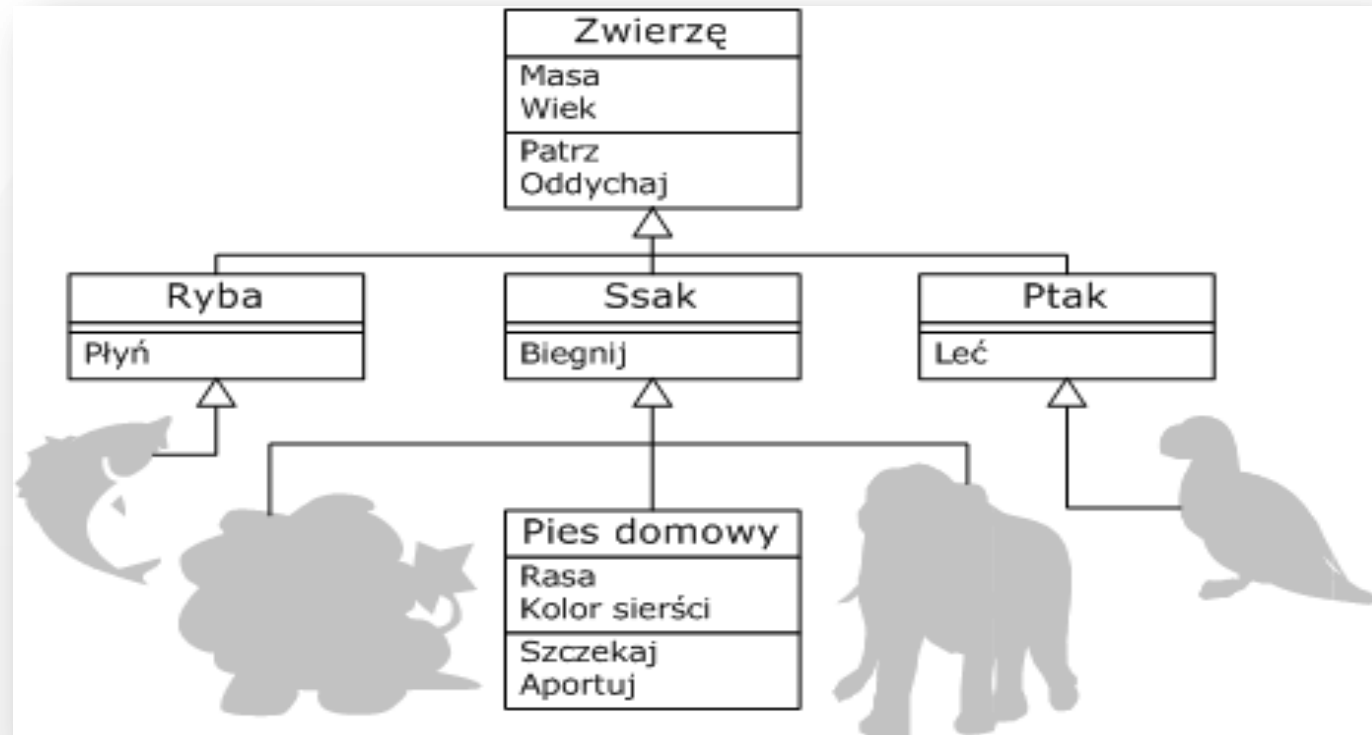
- ❑ Python jest językiem zorientowanym obiektowo i aby w pełni korzystać z jego możliwości należy poznać programowanie obiektowe.
- ❑ Zasadniczą koncepcją w podejściu obiektowym do programowania jest połączenie w całość danych oraz algorytmów, które na tych danych operują.
- ❑ Obiekt posiada pewne własności, czyli dane oraz pewne metody, czyli algorytmy do przetwarzania tych danych.
- ❑ Zbiór obiektów o tych samych własnościach i metodach nazywamy klasą.

Czym jest programowanie obiektowe

- ❑ Programowanie obiektowe (ang. object-oriented programming)
 - ❑ Programy definiuje się za pomocą obiektów – elementów łączących stan (czyli dane, nazywane najczęściej polami) i zachowanie (czyli procedury, tu: metody).
 - ❑ Obiektowy program komputerowy wyrażony jest jako zbiór takich obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań.
- ❑ Podejście to różni się od tradycyjnego programowania proceduralnego, gdzie dane i procedury nie są ze sobą bezpośrednio związane. Programowanie obiektowe ma ułatwić pisanie, konserwację i wielokrotne użycie programów lub ich fragmentów.
- ❑ Największym atutem programowania, projektowania oraz analizy obiektowej jest zgodność takiego podejścia z rzeczywistością.

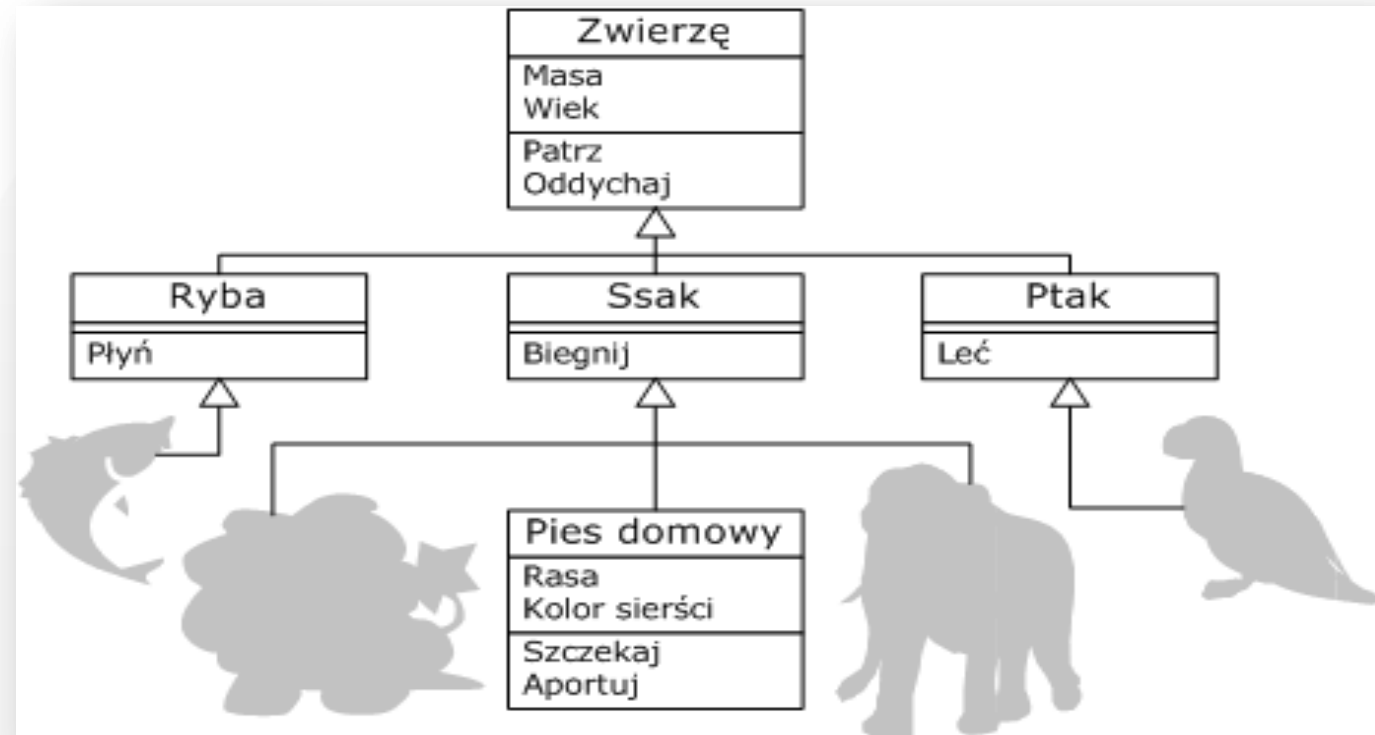
Obiektowość

- ☐ Abstrakcja
- ☐ Hermetyzacja
- ☐ Polimorfizm
- ☐ Dziedziczenie



Zagadnienia obiektowe

- ☐ Klasa
- ☐ Konstruktor
- ☐ Pola
- ☐ Metody
- ☐ Obiekty



Klasa

- ❑ Klasy są podstawowym narzędziem do tworzenia struktur danych i nowych obiektów. Instrukcja **class** pozwala definiować własne klasy:

```
class NazwaKlasy:
```

```
    [opis klasy]
```

- ❑ Klasa definiuje wspólne własności i zachowanie obiektów. Klasa zawiera funkcje przeznaczone do używania z obiektami tej klasy — metody.
- ❑ Klasa jest opisem, definicją.
- ❑ Obiekt jest konkretną zmienną zbudowaną i zachowującą się zgodnie z definicją klasy.

- ❑ Obiekt to instancja (reprezentacja) klasy:

```
NazwaObiektu = NazwaKlasy(argumenty)
```

- ❑ Wyprowadzanie zawartości obiektu na ekran:

```
print(NazwaObiektu)
```

- ❑ Zmienne przechowujące dane zawarte w obiekcie nazywa się polami, natomiast funkcje związane z obiektem, nazywa się metodami.
- ❑ Ważną cechą obiektów jest to, że dzielą się one na grupy charakteryzujące się podobnym zachowaniem. Wynika to z tego, że każdy obiekt ma swoją klasę, lub bardziej poprawnie gramatycznie, jest pewnej klasy.

Notacja obiektowa

- ❑ Ważnym elementem używania obiektów jest notacja obiektowa. Do pól i metod obiektów dostajemy się pisząc nazwę zmiennej dowiązanej do obiektu, kropkę i nazwę atrybutu obiektu.
- ❑ Na przykład dla klasy list istnieje metoda `append`, która pozwala na dopisywanie elementów na końcu każdej listy. Wywołanie `mojalista.append('rzecz')` doda napis 'rzecz' na koniec listy `mojalista`.

```
a = 1  
  
print('a = ', a.numerator, '/', a.denominator)  
  
print('a = ', a.real, ' + i * ', a.imag)  
  
print('a jest typu', a.__class__.__name__)
```

Zmienna self

- ❑ W definicji funkcji podany jest parametr — self.
- ❑ Niemniej, w momencie wywołania, piszemy tylko nazwę funkcji, nie podając żadnych argumentów – chyba że istnieją poza self.
- ❑ Wewnątrz metod, zmienna self odnosi się do samego obiektu.
- ❑ Dzięki temu możliwy jest dostęp do pól obiektu.
- ❑ W momencie wywołania metody obiektu, zostaje on automatycznie wstawiony jako pierwszy argument metody i użytkownik podaje o jeden mniej argument niż metoda wymaga.

Konstruktor klasy

- ❑ Definiujemy atrybuty za pomocą inicjalizatora - jego zadaniem jest zapisanie odpowiednich wartości w polach obiektu i zostaje on wywołany automatycznie w momencie tworzenia obiektu.
- ❑ Jest to metoda zdefiniowana wewnątrz klasy zawierająca atrybut `self` pozwalający na odwołanie się do bieżącej instancji klasy:

```
class NazwaKlasy:  
    def __init__(self, atr1, ..., atrN)  
        self.atr1 = atr1  
        ...  
        self.atrN = atrN  
    [dalszy opis klasy]
```

- ❑ Wywołanie instancji klasy:

```
NazwaObiektu = NazwaKlasy(atr1, ..., atrN)
```

Metoda `__init__`

- ❑ Ważną funkcją specjalną jest metoda `__init__`.
- ❑ Python sam wywołuje ją automatycznie kiedy tworzymy nowy obiekt danego typu.
- ❑ Programista nie musi podać nazwy funkcji do wywołania, bo interpreter wie, że do konstrukcji obiektu służy metoda o tej właśnie nazwie.
- ❑ W tym właśnie sensie metoda `__init__` jest specjalna.
- ❑ Metoda ta powinna wykonywać wszystkie operacje potrzebne do zainicjowania nowego obiektu, w szczególności powinna ona nadawać wartości jego polom.
- ❑ Pod innymi względami jest ona zupełnie zwyczajna, w szczególności można ją wywołać drugi i trzeci raz podając explicite jej nazwę.

Metoda `__str__`

- ❑ Oprócz wcześniej wspomnianej metody `__init__` przydatna jest metoda `__str__`.
- ❑ Służy ona do wytwarzania tekstowej reprezentacji obiektu.
- ❑ Jest automatycznie wywoływana np. przez polecenie `print()`.

Destruktor klasy

- ❑ Destruktor wywoływany wtedy, gdy dany egzemplarz ma być usunięty w celu oczyszczenia wszelkich zasobów pamięci używanych przez egzemplarz.
- ❑ Metoda `__del__()` jest rzadko stosowana w praktyce, ponieważ nie ma gwarancji, że będzie wywołana w określonym odstępie czasu (może się zdarzyć, że nie zostanie wywołana nigdy).
- ❑ Lepsze wyniki można uzyskać definiując jakąś jawnie wywoływaną metodę `close()`, która wykona także operacje oczyszczające. □ Instrukcja `del` nie wywołuje metody `__del__()` bezpośrednio, lecz zmniejsza o jeden liczbę odwołań do obiektu.

```
def __del__(self):  
    [ciało destruktora]
```

Metody klasy

- ❑ Metoda jest to funkcja zdefiniowana wewnątrz klasy.

```
class NazwaKlasy:
    def __init__(self, atr1, ..., atrN):
        self.atr1 = atr1
        ...
        self.atrN = atrN
    def NazwaMetody(self, atrybuty):
        [ciało metody]
```

- ❑ Wywołanie metody:

```
NazwaObiektu = NazwaKlasy(atr1, ..., atrN)
```

```
NazwaObiektu.NazwaMetody(atorybuty)
```

Zmienne klasowe

- ❑ Zmienne definiowane wewnątrz klasy:
- ❑ Zasięg zmiennych obejmuje całą klasę.

```
class NazwaKlasy:

    zmienna = wartość

    def __init__(self, atr1, ..., atrN):
        self.atr1 = atr1
        ...
        self.atrN = atrN

    def NazwaMetody(self, atrybuty):
        [ciało metody]
```

Przykład klasy

```
class Wektor(object):  
    def __init__(self, x, y):  
        self.a = x  
        self.b = y  
        print("wektor został stworzony!")  
    def dlugosc(self):  
        return (self.a ** 2 + self.b ** 2) ** 0.5  
    def __str__(self):  
        return "Wektor x={0.a} y={0.b}".format(self)  
  
w1 = Wektor(5, 7) print w1
```

Operatory dwuargumentowe

- ❑ Operatory dwuargumentowe są kolejnym typem metod specjalnych.
 - ❑ Ich nazwy zawierają po dwa znaki podkreślenia na początku i na końcu.
 - ❑ Wywoływane są gdy w kodzie pomiędzy dwoma obiektami pojawia się symbol operatora, np. +
 - ❑ Z każdym operatorem związana jest domyślna nazwa metody, która definiuje co programista przewidywał zrobić gdy np. dodawał do siebie dwa wektory.
- ❑ Wracając do przykładowego Wektora: jeśli zechcemy dodać dwa Wektory korzystając ze znaku +, nie będzie to wykonalne, otrzymamy TypeError. Stanie się to możliwe dopiero kiedy napiszemy metodę `__add__`, czyli implementację operatora + dla klasy Wektor.

Przykład działania operatora dwuargumentowego

```
class Wektor(object):  
  
    def __add__(self, other):  
        return Wektor(self.a + other.a, self.b + other.b)  
  
w1=Wektor(5,7)  
print(w1 + w1)
```

Inne funkcje specjalne

- ❑ Funkcji specjalnych jest sporo, bo tabela operatorów jest długa, a należy pamiętać, że dochodzą jeszcze operatory skrócone (`+=`, `-=`, itp.).
- ❑ Każdy z nich ma odpowiadającą mu funkcję specjalną, którą można napisać by umożliwić wykonywanie tej operacji na obiektach danej klasy.
- ❑ Wszystkie funkcje specjalne są opisane w dokumentacji języka, <http://docs.python.org/dev/reference/datamodel.html#special-method-names>.
- ❑ W trakcie pracy w interaktywnym interpreterze przydatny jest moduł `operator`, który zawiera funkcje specjalne do arytmetyki na liczbach. Można skorzystać z jego dokumentacji, która zawiera opisy funkcji specjalnych do operacji matematycznych (`import operator; help(operator)`).

Cykl życia obiektu

1. `nazwa = Obiekt('niebieski')`



2. `nowa_nazwa = nazwa`



3. `nazwa = Obiekt('zielony')`



4. `del nowa_nazwa`



Dziedziczenie

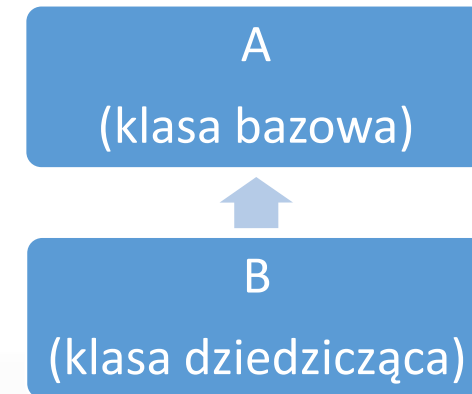
- ❑ Dziedziczeniem nazywamy sytuację, w której definiujemy nową klasę jako rozwinięcie istniejącej. Tę nową klasę nazywamy dziedziczącą, a starą bazową.
- ❑ Klasa bazowa jest klasą zawierającą pewne właściwości, które będą dziedziczone przez klasę dziedziczącą

```
class NazwaKlasyBazowej:
```

```
...
```

```
class NazwaKlasyDziedziczącej(NazwaKlasyBazowej):
```

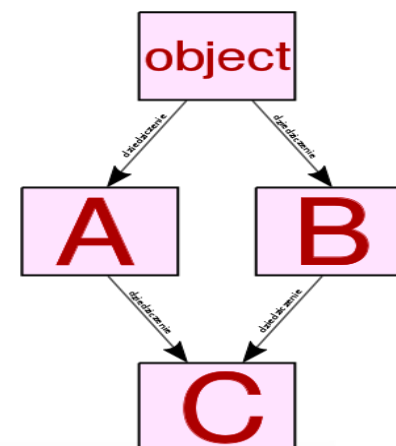
```
...
```



- ❑ Załóżmy, że mamy klasę B, która dziedziczy po klasie A.
- ❑ Dziedziczenie oznacza, że metody i pola dostępne w klasie-bazowej A, są również dostępne w klasie-dziedziczącej B.
- ❑ Innymi słowy, wszystkie obiekty klasy B mają takie same atrybuty jak obiekty klasy A. Powoduje to, że w ogólności, tam gdzie mogliśmy użyć obiektu klasy A, możemy również użyć obiektu klasy B. Oznacza to, że obiekty typu B są też obiektami typu A, czyli B jest podtypem czy też podklasą A. Symetrycznie, A jest nadklasą B.

Polimorfizm

- ❑ Sytuację kiedy obiektów dwóch różnych klas mają wspólny zestaw metod i pól i można je używać zamiennie nazywamy **polimorfizmem**.
- ❑ Polimorfizm jest ważny, gdyż znacznie ułatwia wykorzystanie obiektów różnych klas. Wygodniej jest myśleć o operacji dodawania zdefiniowanej dla wszystkich liczb, niż o operacji dodawania liczb zmiennie-przecinkowych, operacji dodawania liczb całkowitych, operacji dodawania liczb zespolonych, itd.
- ❑ Dziedziczenie jest mechanizmem, który pozwala na uzyskanie polimorfizmu w bardzo łatwy sposób. Ponieważ klasy potomne na wstępie otrzymują komplet metod i pól rodzica, to możliwość podstawienia obiektu klasy dziecka za rodzica otrzymuje się automatycznie.



- ❑ W sytuacji gdy klasa dziedzicząca posiada więcej niż jedną klasę rodzicielską to obiekty tej klasy odziedziczy atrybuty po wszystkich rodzicach.
- ❑ W przypadku gdy zażądamy dostępu do metody czy pola, które występuje tylko w jednym z rodziców, to sytuacja jest prosta — przeszukiwany jest najpierw pierwszy rodzic, potem drugi, itd., dopóki nie natrafimy na atrybut o żądanej nazwie.
- ❑ W sytuacji kiedy zażądamy dostępu do atrybutu, dostępnego u więcej niż jednego rodzica, to decyduje kolejność dziedziczenia. Np. C, potem A, potem B, a na końcu object.

Funkcja super

- ❑ W sytuacji gdzie chcielibyśmy wywołać metodę nadrzędną, czyli tą odziedziczoną po rodzicu możemy to zrobić na dwa sposoby:
 - ❑ Wykorzystując **super** gdzie podajemy nazwę bieżącej klasy.
 - ❑ Wykorzystując **bezpośrednie** gdzie odwołanie podajemy nazwę klasy macierzystej.
- ❑ W przypadku gdy zmieniamy hierarchię dziedziczenia i stosujemy funkcję `super()` nie musimy zmieniać wszystkich odwołań do odziedziczonych metod. Jeśli odwoływaliśmy się do metod rodzica przez `super`, to automatycznie zaczniemy się odwoływać do metod nowego rodzica.
- ❑ Dodatkowo funkcja `super` zachowuje się w szczególny sposób w sytuacji wielokrotnego dziedziczenia (czyli w sytuacji, gdy dana klasa ma więcej niż jednego rodzica). O ile każda z metod w hierarchii wywołuje metodę "nadrzędną" przez `super`, to wszystkie metody w hierarchii zostaną wywołane w pewnym ściśle określonym porządku.

Odwołanie do klasy nadrzędnej

- ❑ W pierwszym rozwiązaniu, do metody rodzicielskiej odwołujemy się tak samo jak do każdej innej funkcji, podając ścieżkę do niej.

```
class Welcome(object):  
    def hello(self):  
        return 'Hello'  
  
class WarmWelcome(Welcome):  
    def hello(self):  
        return Welcome.hello(self) + ", you're welcome"  
  
print(Welcome().hello())  
print(WarmWelcome().hello())
```

Zastosowanie funkcji super

- ❑ W drugim rozwiązaniu, do metody rodzicielskiej odwołujemy się poprzez pomocniczy obiekt zwracany przez super.
- ❑ Funkcja super bierze dwa argumenty — klasę od której zaczynamy poszukiwania (czyli generalnie klasę wewnątrz której definicji wywołujemy super) oraz argument self.

```
class HeartyWelcome(Welcome):  
    def hello(self):  
        return super(HeartyWelcome, self).hello() + ", you're  
heartily welcome"  
  
print(HeartyWelcome().hello())
```

Przestrzenie nazw

- ❑ Nazwy nie mogą się powtarzać wewnątrz jednej przestrzeni nazw, ale to ograniczenie nie dotyczy różnych przestrzeni nazw.
- ❑ Przestrzeń nazw tworzy po prostu odwzorowanie z nazw do obiektów.
- ❑ W Pythonie przestrzenie nazw są zaimplementowane po prostu jako słowniki, w których kluczami są napisy - nazwy zmiennych, a wartościami obiekty.
- ❑ Każdy moduł ma swoją przestrzeń nazw. Każda klasa ma oddzielną przestrzeń nazw. Każdy obiekt ma swoją własną przestrzeń nazw. Każde wywołanie funkcji tworzy nową przestrzeń nazw.

Przestrzeń nazw funkcji

- ❑ Każda funkcja posiada własną przestrzeń nazw, nazywaną lokalną przestrzenią nazw, a która śledzi zmienne funkcji, włączając w to jej argumenty i lokalnie zdefiniowane zmienne.
- ❑ W momencie kiedy funkcja zaczyna wykonywanie, to w tej lokalnej przestrzeni nazw znajdują się tylko parametry określone w nagłówku funkcji. W miarę wykonywania funkcji, lokalnie zdefiniowane zmienne są dopisywane do tej przestrzeni nazw. Wraz z zakończeniem wykonywania funkcji ta przestrzeń nazw znika.
- ❑ Dostęp do najbardziej wewnętrznej przestrzeni nazw można uzyskać (pomijając bezpośrednie użycie zmiennych), poprzez funkcję `locals`.
- ❑ Funkcja ta zwraca słownik, który jest właśnie tym słownikiem który pamięta co lokalna przestrzeń nazw zawiera.

Przestrzeń nazw w module

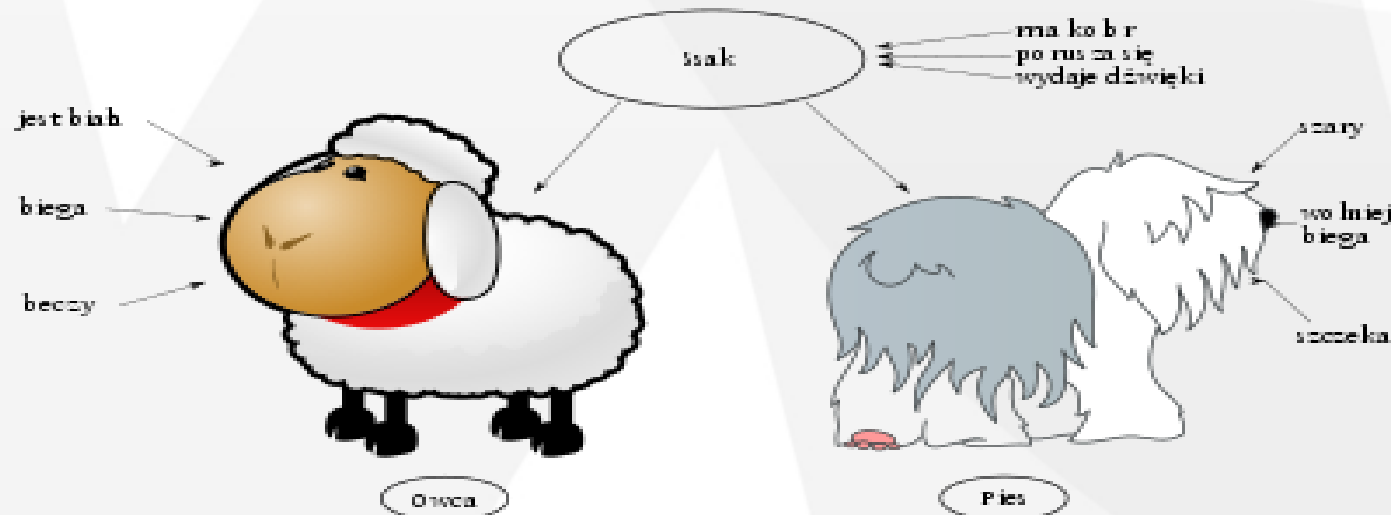
- ❑ Każdy moduł, czyli każdy Pythonowy plik, posiada własną przestrzeń nazw.
- ❑ Na początku jest ona pusta, a w miarę wykonywania poleceń w pliku (czyli wczytywania definicji klas czy funkcji, a nie wykonywania tych funkcji!) odpowiednie nazwy są do niej dopisywane.
- ❑ Ostatecznie znajdują się w niej funkcje, klasy i inne zaimportowane moduły, a także zmienne zdefiniowane na poziomie modułu.
- ❑ Przestrzeń nazw w module jest nazywana globalną przestrzenią nazw, bo nazwy w niej zdefiniowane są widoczne również w funkcjach i klasach zdefiniowanych wewnątrz, o czym za chwilę.
- ❑ Dostęp do przestrzeni nazw modułu można uzyskać poprzez funkcję `globals`.

Przestrzeń nazw w obiekcie

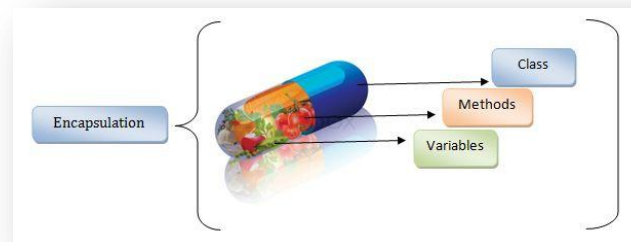
- ❑ Rozpoczęcie definicji klasy słowem `class`, tworzy również nową przestrzeń nazw. W zasadzie jest to przestrzeń nazw jak każda inna, i mogą się w niej znaleźć dowolne rzeczy, ale w praktyce są to głównie funkcje, a rzadziej zmienne klasowe, czyli zmienne wspólne dla wszystkich obiektów tej klasy.
- ❑ Po stworzeniu obiektu danej klasy, tworzymy również nową przestrzeń nazw dla tego indywidualnego obiektu.
- ❑ Żyje ona tak samo długo jak ten obiekt, i zostaje zniszczona razem z nim.

Abstrakcyjne spojrzenie na programowanie obiektowe

- ❑ Abstrakcyjną ideą programowania obiektowego jest powiązanie stanu (danych, które określone są zwykle polami) i zachowania (algorytmy związane ze stanem i całym obiektem, określane słowem metody).
- ❑ Program korzystający z obiektowości wyrażony jest jako zbiór takich obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań.

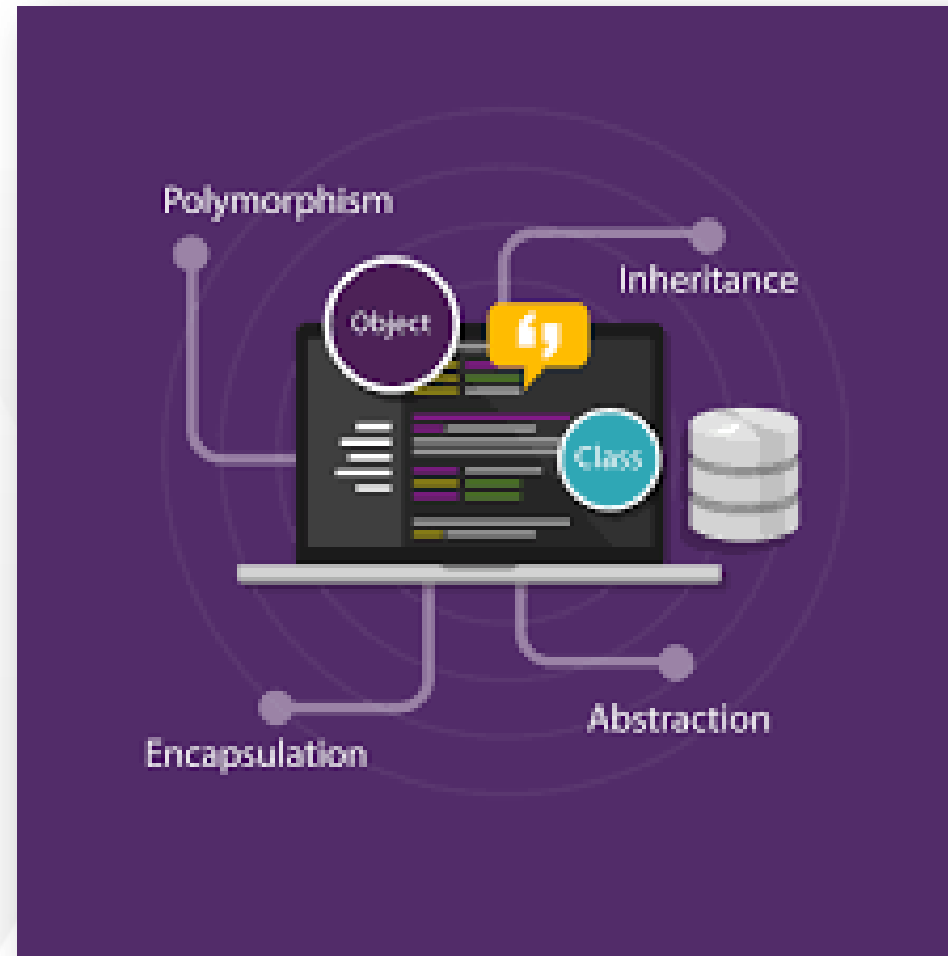


Enkapsulacja (Hermatyzacja)



- ❑ Enkapsulacja polega na tym, że szczegóły implementacji są ukryte. Dzięki temu obiekt nie może zmieniać stanu wewnętrznego innych obiektów w nieoczekiwany sposób. Tylko wewnętrzne metody danego obiektu są uprawnione do zmiany jego stanu. Każdy typ obiektów ma swój interfejs, który określa dopuszczalne metody współpracy.
- ❑ Jesteśmy jakimś obiektem, dla ustalenia A. Widzimy drugi obiekt, B. Obiekt B wie o sobie wszystko, my wiemy o nim tylko tyle, ile on nam udostępnia. W szczególności nie mamy dostępu do wielu zmiennych tego obiektu, możemy natomiast go "poprosić" żeby coś zrobił z tymi zmiennymi, lub podał nam ich wartość, wywołując metodę, jaką obiekt nam udostępnia.
- ❑ Analogia z życia:
 - ❑ Idziemy do apteki, chcemy kupić jakiś lek — nie bierzemy go z półki sami, tylko wywołujemy określoną metodę obiektu Apteka, prosząc Panią Sprzedawczynię, aby ten lek nam podała. Nie interesuje nas w jaki sposób ona to zrealizuje — tzn. czy będzie np. musiała poszukać go w magazynie, czy też wejść na stołek bo lek stoi na górnej półce. My tego sami robić nie musimy, to już nie nasz problem, leży to w gestii drugiego obiektu.

- ☐ Enkapsulacja
- ☐ Dziedziczenie
- ☐ Polimorfizm





Moduły i pakiety

Czym są moduły

- ❑ W Pythonie moduły są po prostu plikami z rozszerzeniem .py, w których zawarto pewien zestaw funkcji.
- ❑ Moduły importujemy do swojego programu głównego za pomocą komendy import.

```
import nazwaModułu
```

- ❑ Najpierw moduł jest ładowany w wykonujący się skrypt Pythona. Potem jego zawartość jest odczytywana tak, jakby załadowany moduł był częścią programu.
- ❑ Jeśli zdarzy się, że interpreter dwa razy napotka polecenie dołączenia tego samego modułu, to drugie polecenie zostanie zignorowane.

Wbudowane moduły Pythona

- ❑ Pełna lista wbudowanych modułów Pythona znajduje się w dokumentacji:

<https://docs.python.org>

Importowanie i użycie modułu

- ❑ Aby zaimportować moduł urllib, który pozwala nam na odczytywanie danych z URL, musisz użyć komendy import:

```
import urllib
```

- ❑ Aby użyć funkcji z zaimportowanej biblioteki:

```
urllib.urlopen(...)
```

Przeszukiwanie modułów

- ❑ Aby przeglądnąć zawartość modułów w Pythonie z pomocą wykorzystujemy dwie funkcje
 - ❑ `dir()`
 - ❑ `help()`
- ❑ Za pomocą funkcji `dir` możemy zobaczyć, jakie funkcje zostały umieszczone w dowolnym module.

Tworzenie własnych modułów

- ❑ Aby stworzyć moduł wystarczy utworzyć nowy plik z rozszerzeniem .py (np.: mod1.py).
- ❑ Następnie wystarczy go zaimportować za pomocą komendy `import mod1` (bez rozszerzenia .py)
- ❑ Od tej pory dostępne są wszystkie metody zawarte w zaimportowanym module mod1.

- ❑ Pakiety są przestrzeniami nazw, które zawierają w sobie wiele modułów, a nawet innych pakietów, czyli są folderami, ale z pewnym znakiem szczególnym.
- ❑ Każdy pakiet w Pythonie jest folderem, który musi zawierać specjalny plik nazwany `init.py`. Ten plik może być pusty i służy informowaniu, że ten folder zawiera pakiet Pythona. Dzięki temu może być importowany tak samo jak moduły.
- ❑ Jeśli utworzymy folder o nazwie `kat1`, która jest tożsama nazwie pakietu, możemy w nim utworzyć moduł nazwany `mod1`.

Uwaga! Nie możemy zapomnieć także dodać pliku `__init__.py` wewnątrz folderu `kat1`.

Używanie modułu z pakietu

- ❑ Aby używać modułu `mod1` możemy go zaimportować na dwa sposoby:
 - ❑ `import kat1.mod1`
 - ❑ `from kat1 import mod1`
- ❑ W pierwszym przypadku będziemy musieli używać przedrostka `kat1` za każdym razem, gdy będziemy chcieli posłużyć się funkcją oferowaną przez `mod1`.
- ❑ W drugim przypadku nie musimy, ponieważ, zaimportowaliśmy moduł do naszej przestrzeni nazw modułów.



Operacje na plikach

Cele zapisu do pliku

- ☐ Do tej pory zajmowaliśmy się wyłącznie przechowywaniem danych w pamięci operacyjnej komputera.
- ☐ Pamięć operacyjna komputera jest jednak ulotna.
- ☐ Aby zabezpieczyć dane należy zapisać je w pliku dyskowym.

Tworzenie i otwarcie pliku

- ❑ Polecenie stworzenia i otwarcia do zapisu pliku "plik.txt" w aktualnym katalogu dyskowym (domyślnie C:\Python...):

```
F = open("plik.txt", "w")
```

- ❑ Obiekty plikowe mają trzy podstawowe atrybuty:
 - ❑ **F.name** # określa nazwa pliku
 - ❑ **F.mode** # określa tryb w jakim otwarto plik
 - ❑ **F.closed** # określa czy plik jest zamknięty

Metody obsługi plików

- ❑ Zapis do pliku

```
F.write("Początek pliku")
```

```
F.writelines(["\n3 linia", "\n4 linia", "\n5 linia"])
```

- ❑ Zapis danych z bufora do pliku

```
F.flush()
```

- ❑ Zapis danych z bufora do pliku i zamknięcie pliku

```
F.close()
```

Metody obsługi plików

- ❑ Odczyt z pliku

```
print (F.read())  
print (F.read(14))           # odczyt fragmentu o określonej długości  
print (F.readlines())       # odczyt sekwencji napisów
```

- ❑ Odczyt aktualnej pozycji w pliku

```
F.tell()
```

- ❑ Ustawienie pozycję w pliku

```
F.seek(0)  
F.seek(-14,1)                # przesunięcie względem aktualnej pozycji  
F.seek(0,2)                  # przesunięcie względem końca pliku
```

Metody obsługi plików

- ❑ Skracanie pliku względem podanej pozycji

F.truncate(26)

- ❑ Metoda `write` nie kończy zapisanych danych znakiem końca linii. By przejść do kolejnej linii, należy dodać znak `(\n)`.
- ❑ Normalne zakończenie programu powoduje automatyczne zamknięcie wszystkich otwartych plików, jednak należy samodzielnie zamykać wszystkie pliki.

Operacje na plikach

- ❑ Jednym z podstawowych zadań systemu operacyjnego jest obsługa dyskowego systemu plików.
- ❑ Zostaną omówione funkcje służące do manipulacji plikami w całości, ich przenoszenia i usuwania oraz funkcje obsługujące katalogi dyskowe.

Operacje na plikach i katalogach

- ❑ Import modułów pakietu os pozwalających na operacje na plikach i katalogach

```
from os import *

getcwd()           # sprawdzenie aktualnego katalogu
chdir('tcl')       # zmiana bieżącego katalogu na tcl
listdir('.')       # wyświetlenie zawartości katalogu bieżącego
listdir('tcl')     # wyświetlenie zawartości katalogu tcl
listdir(r'C:\Python24\tcl')
```

Filtrowanie plików

- ❑ Aby filtrować pliki i katalogi według określonego wzorca, musimy posłużyć się funkcją `fnmatch(nazwa, wzorzec)` z modułu `fnmatch`.

```
import fnmatch
```

- ❑ Funkcja ta zwraca prawdę, wtedy i tylko wtedy, gdy nazwa odpowiada wzorcowi:

```
fnmatch.fnmatch('Python', 'P*n')           # True  
fnmatch.fnmatch('Python', 'P*e')          # False
```

Filtrowanie plików

- ❑ Lista plików z rozszerzeniem 'tcl':

```
[x for x in listdir(r'C:\Python24\tcl\tcl8.4') if \
fnmatch.fnmatch(x, '*.tcl')]
```

- ❑ Lista plików o nazwach kończących się na 't' lub 'y':

```
[x for x in listdir(r'C:\Python24\tcl\tcl8.4') if \
fnmatch.fnmatch(x, '*[ty]*')]
```

- ❑ Sprawdzanie, czy dany obiekt dyskowy istnieje:

```
path.exists('C:\\Python24\\tcl\\tcl8.4\\history.tcl')
```

Operacje na katalogach

- ❑ Tworzenie na dysku nowego katalogu

```
mkdir('nazwaKatalogu')
```

- ❑ Zmiana nazwy pliku lub katalogu

```
rename('staraNazwa', 'nowaNazwa')
```

- ❑ Sprawdzanie czy dany obiekt dyskowy jest plikiem

```
path.isfile('nazwaPliku')
```

- ❑ Sprawdzenie czy dany obiekt dyskowy jest katalogiem

```
path.isdir('nazwaKatalogu')
```

- ❑ Sprawdzenie czy dany obiekt dyskowy jest dyskiem:

```
path.ismount('nazwaDysku')
```

Operacje na katalogach

- ❑ Zwracanie długość pliku w bajtach

```
path.getsize('plik')      # dla katalogów jest zwracane 0
```

- ❑ Zwracanie czasu stworzenia i czasu ostatniej modyfikacji obiektu dyskowego:

```
from time import ctime
```

- ❑ Usuwanie z dysku pliku o podanej nazwie:

```
remove('nazwaPliku')
```

- ❑ Usuwanie z dysku katalogu o podanej nazwie:

```
rmdir('nazwaKatalogu')
```



Połączenie Pythona z bazą danych

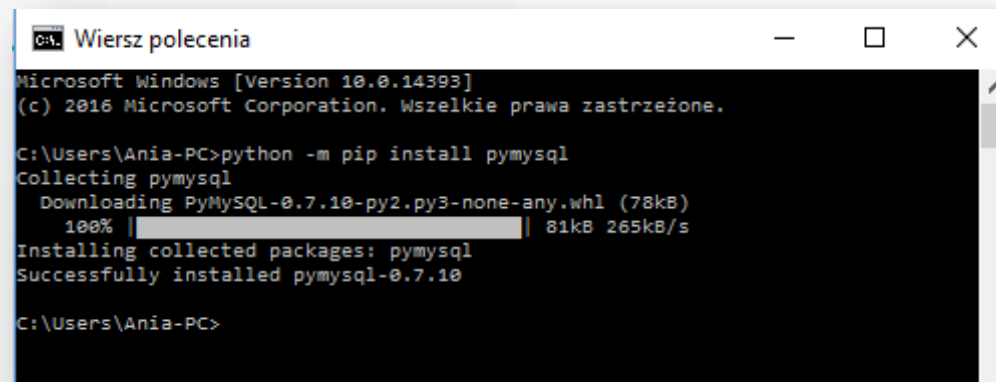
Biblioteka Pythona do obsługi pymysql

- ❑ Pobieramy bibliotekę do obsługi MySQL w terminalu:

```
python -m pip install pymysql
```

- ❑ Importujemy bibliotekę:

```
import pymysql
```



```
CA: Wiersz polecenia
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\Users\Ania-PC>python -m pip install pymysql
Collecting pymysql
  Downloading PyMySQL-0.7.10-py2.py3-none-any.whl (78kB)
    100% |#####| 81kB 265kB/s
Installing collected packages: pymysql
Successfully installed pymysql-0.7.10

C:\Users\Ania-PC>
```


Połączenie Pythona z MySQL

- ❑ Do połączenia z bazą danych służy polecenie `connect`, które można wykonać na kilka sposobów:

```
conn = pymysql.connect("host", "user", "hasło", "baza")
```

```
conn = pymysql.connect(host="host", user="user", passwd="hasło", db="baza")
```

```
conn = pymysql.connect(read_default_file="/etc/mysql/myapp.cnf")
```

- ❑ Dodatkowe opcje połączenia:

- ❑ `compress=1` #włącza kompresję gzip,
- ❑ `use_unicode=1` #zwraca dane jako obiekty unicode,

Pobieranie rekordów z bazy danych

- ❑ Do pobierania rekordów z bazy danych służy polecenie execute:

```
import pymysql

conn = pymysql.connect("localhost", "user", "haslo", "testy")

c = conn.cursor()                # ustawienie kursora na początku pliku

c.execute("?zapytanie?")         # wykonanie zapytania

print(c.fetchall())             # wyświetlenie wyników
```

Odwoływanie się do wartości wynikowych rekordów

- ❑ Wypisanie całej zawartości:

```
print (c.fetchall())
```

- ❑ Wypisanie pierwszego rekordu:

```
print (c.fetchall()[0][0])
```

lub

```
print (c.fetchone()[0])
```

- ❑ Wypisanie drugiego rekordu:

```
print (c.fetchall()[1][0])
```

itd.

Odwoływanie się do liczebności wynikowych rekordów

- ❑ Wyświetla wszystkie rekordy:

```
print (c.fetchall())
```

- ❑ Wyświetla N-pierwszych rekordów:

```
print (c.fetchmany(N) )
```

- ❑ Zwraca liczbę pobranych wierszy:

```
print (c.rowcount)
```

Złożone zapytania

- ❑ Bardziej złożone zapytania również wykonuje się za pomocą metody `execute`:

```
c.execute("SELECT nick FROM users WHERE pass = %s", ("xxx"))
```

Wprowadzanie danych do bazy

- ❑ Wykorzystanie instrukcji INSERT, UPDATE, ALTER itd. również odbywa się za pomocą metody execute:

Np.:

```
c = conn.cursor() c.execute("INSERT INTO users VALUES(' ', 'albin', 'yyy')")  
c.execute("INSERT INTO users VALUES(' ', %s, %s)", ('user1', 'haslo1'))  
c.execute("INSERT INTO users VALUES(' ', %s, %s)", ('user2', 'haslo2'))
```

Wprowadzanie wielu danych do bazy

- ❑ Istnieje możliwość wykorzystania metody `executemany` w celu wprowadzenia wielu rekordów jednocześnie do bazy danych:

```
c.executemany("INSERT INTO users VALUES(' ',%s,%s)",  
              ( ('user1', 'haslo1'), ('user2', 'haslo2') ))
```

- ❑ Dostępne są też transakcje:
 - ❑ `conn.begin()`
 - ❑ `conn.commit()`
 - ❑ `conn.rollback()`

Szablon połączenia z bazą danych

```
import pymysql

conn = pymysql.connect("localhost", "root", "miki123", "pracownicy")

cursor = conn.cursor()

sql = "SELECT * FROM dane"

try:

    ....

except:

    print("Error: Unable to fetch data")

conn.close()
```


Szablon pobierania danych z bazy

```
cursor.execute(sql)

results = cursor.fetchall()

print("ID \t Name \t Surname \t Phone" )

for row in results:

    lp = row[0]

    name = row[1]

    last = row[2]

    phone = row[3]

    #print ("lp=%s,\t name=%s,\t last=%s,\t phone=%s" % (lp, name, last, phone))

    print ("%s \t %s \t %s \t %s" % (lp, name, last, phone))
```



Dziękuję za uwagę!



Dodatek: Przetwarzanie list

Listy są wykorzystywane

- ❑ Programy bardzo często przetwarzają sekwencje danych.
- ❑ Jakkolwiek szczegóły tego przetwarzania bywają rozmaite, istnieje kilka podstawowych typów operacji wykonywanych na sekwencjach danych.

Szybkie generowanie list

- ☐ Wytworniki – narzędzia do generowania list o złożonej zawartości dostępne są w pięciu postaciach:
 - ☐ prostej,
 - ☐ prostej warunkowej,
 - ☐ rozszerzonej,
 - ☐ rozszerzonej z jednym warunkiem,
 - ☐ rozszerzonej z wieloma warunkami.

Postać prosta wytwornika

- ❑ Postać prosta wytwornika daje w wyniku listę, zawierającą wartości wyrażenia obliczone dla elementów sekwencji wejściowej:

`[wyrażenie for zmienna in sekwencja]`

Np.:

<code>[2*x for x in 1]</code>	<code># podwojenie wartości</code>
<code>[(x, x*x) for x in range(1,5)]</code>	<code># stworzenie par (x, kwadrat x):</code>
<code>[(x, ord(x)) for x in "ABCDEF"]</code>	<code># tabela kodowa ASCII:</code>
<code>[[] for x in range(10)]</code>	<code># lista zawierająca 10 pustych list:</code>

Postać prosta warunkowa wytwornika

- ❑ Postać prosta warunkowa pozwala umieszczać na liście tylko takie elementy, które spełniają pewien warunek (operację usuwania z listy elementów niespełniających określonego warunku nazywamy filtrowaniem danych).

[wyrażenie for zmienna in sekwencja if warunek]

Np.:

```
[x for x in l if x>10]      # Tylko liczby większe od 10
```

```
[x for x in range(1,20) if not (x%3) or not (x%5)]  
                        # Tylko liczby podzielne przez 3 lub 5
```

```
[(x, ord(x)) for x in "ABCDEF" if x in "AEIOUY"]  
                        # Tabela kodowa ASCII tylko dla samogłosek:
```

Postać rozszerzona wytwornika

- ❑ Postać rozszerzona pozwala tworzyć nową listę w oparciu o więcej niż jedną istniejącą listę.

```
[wyrażenie for zmienna1 in sekwencja1 for zmienna2 in sekwencja2 ... ]
```

Np.:

```
[(x,y) for x in range(1,5) for y in range(4,0,-1)]  
    # Pary każdy element z każdym:  
[x-y for x in range(1,5) for y in range(4,0,-1)]  
    # Różnica między wartością z pierwszej i drugiej listy  
[`x`+y+`z` for x in [1,2] for y in ['A','B'] for z in [0,3] ]  
    # Sklejenie napisu z wartości pochodzących z trzech list:
```


Postać rozszerzona wytwornika z jednym warunkiem

- Postać rozszerzona z jednym warunkiem pozwala na określenie pojedynczego warunku, który muszą spełniać dane kwalifikujące się na listę wynikową.

```
[wyrażenie for zmienna1 in sekwencja1
    for zmienna2 in sekwencja2
    ...
    if warunek ]
```

Np.:

```
[(x,y) for x in range(1,5) for y in range (6,3,-1) if x<y]
    # Pary każdy element z każdym, tylko jeżeli pierwszy element jest
    # mniejszy od drugiego
[(x,y) for x in range(1,5) for y in range (6,3,-1) if x+y<7]
    # Pary każdy element z każdym, tylko jeżeli suma elementów jest
    # mniejsza od 7
[(x,y) for x in range(1,5) for y in range (6,2,-1) if not (x%2) or y%2]
    # Pary każdy element z każdym, pod warunkiem, że pierwszy element jest
    # parzysty, lub drugi jest nieparzysty:
```

reaktor

Postać rozszerzona z wieloma warunkami

- ❑ Postać rozszerzona z wieloma warunkami pozwala na określenie warunków, które muszą spełniać dane pobierane z poszczególnych list źródłowych. Jej składnia jest następująca:

[wyrażenie

```
for zmienna1 in sekwencja1 if warunek1  
for zmienna2 in sekwencja2 if warunek2  
... ]
```

Np.:

```
[(x,y) for x in range(1,5) if not (x%2) for y in range (6,2,-1) if y%2]  
# Pary każdy element z każdym, pod warunkiem, że pierwszy element jest  
# parzysty a drugi jest nieparzysty (z pierwszej listy bierzemy tylko  
# elementy parzyste, a z drugiej - nieparzyste)
```

reaktor

Funkcje ułatwiające przetwarzanie sekwencji danych

- ❑ W Pythonie dostępnych jest pięć funkcji, których przeznaczeniem jest ułatwienie przetwarzania sekwencji danych:
 - ❑ Funkcja apply,
 - ❑ Funkcja map,
 - ❑ Funkcja zip,
 - ❑ Funkcja filter,
 - ❑ Funkcja reduce.

Funkcja apply

- ❑ Funkcja apply, której działanie polega na wywołaniu funkcji z parametrami uzyskanymi z rozpakowania sekwencji.

Np.:

```
xyz=(3,5,2)
```

```
apply(dziel,xyz)
```

Funkcja map

- ❑ Funkcja map pozwala wywołać określoną funkcję dla każdego elementu sekwencji z osobna. Zwraca listę rezultatów funkcji, o takiej samej długości jak listy parametrów.

Np.:

- ❑ `map(lambda x: x*x, range(5))`

- ❑ `map(dziel, range(5), range(5), [2]*5)`

Funkcja zip

- ❑ Funkcja zip służy do konsolidacji danych, tj. operacji łączenia kilku list w jedną, w której wartość pojedynczego elementu listy wynikowej zależy od wartości pojedynczych elementów list źródłowych.
- ❑ Funkcja zip przyjmuje jako swoje parametry jedną lub więcej sekwencji, po czym zwraca listę krotek, których poszczególne elementy pochodzą z poszczególnych sekwencji.

Np.:

- ❑ `zip("abcdef", [1,2,3,4,5,6])`
- ❑ `zip(range(1,10), range(9,0,-1))`
- ❑ `zip("zip", range(0,9), zip(range(0,9)))` – gdy dł. sekw. są różne

Funkcja filter

- ❑ Funkcja filter służy do filtrowania danych. Przyjmuje jako parametry funkcję oraz sekwencję, po czym zwraca sekwencję zawierającą te elementy sekwencji wejściowej, dla których funkcja zwróciła wartość logiczną True.

Np.:

- `filter(samogloska, "Ala ma kota, kot ma Ale")`
- `filter(lambda x: not samogloska(x), "Ala ma kota, kot ma Ale")`
- `filter(lambda x: x%2-1, range(0,11))`

Funkcja reduce

- ❑ Funkcja reduce służy do agregowania danych, tj. operacji obliczenia pojedynczego wyrażenia, zależnego od wszystkich elementów listy źródłowej.
- ❑ Funkcja reduce przyjmuje jako parametry funkcję oraz sekwencję, zwraca pojedynczą wartość.
- ❑ Na początek wykonuje funkcję dla dwóch pierwszych elementów sekwencji, następnie wykonuje funkcję dla otrzymanego w pierwszym kroku rezultatu i trzeciego elementu sekwencji, następnie wykonuje funkcję dla otrzymanego w drugim kroku rezultatu i czwartego elementu sekwencji, itd., aż dojdzie do końca sekwencji.

Np.:

- `reduce(lambda x,y: x+y, [1,2,3])`
- `reduce(lambda x,y: x*y, [1,2,3,4])`
- `reduce(lambda x,y: x+y, map(lambda x: x*x, range(1,10)))`



Koniec