Programowanie Python
Składnia - poziom podstawowy

Piotr Pociask
www.gis-support.pl

MINISTERSTWO
ŚRODOWISKA

Sfinansowano ze środków
Narodowego Funduszu
Ochrony Środowiska
i Gospodarki Wodnej

Agenda

1. Podstawy programowania obiektowego
2. Główne cechy i składnia języka
3. Biblioteki i ich wykorzystanie
4. Omówienie podstawowych typów danych, operatorów oraz operacji na nich
5. Obsługa wyjątków
6. Wyrażenia warunkowe
7. Pętle
8. Kolekcje
9. Funkcje
10. Klasy

Programowanie zorientowane obiektowo (OOP, ang. *object-oriented programming*) operuje na obiektach. Odzwierciedlają one rzeczywiste elementy dzięki czemu ten sposób programowania jest bliski naturalnemu rozumowaniu człowieka, oczywiście w uproszczonej wersji.

Każdy obiekt opisuje zestaw właściwości (**atrybuty**, informacje o obiekcie, jego stanie) oraz zachowań (**metody**, funkcje/działania jakie wykonuje obiekt).

Języki programowania nastawione na programowanie obiektowe to m.in. JAVA, C++, C#, Ruby, ObjectPascal, **Python.**

Cechy Pythona

Język interpretowany - kod źródłowy jest analizowany przez interpreter i wykonywany na bieżąco. W przeciwieństwie do kompilatora kod źródłowy nie jest tłumaczony do kodu maszynowego, stanowią go zwykłe pliki tekstowe z rozszerzeniem *.py*.

Obiektowość – w Pythonie wszystko jest obiektem tzn. ma swoje atrybuty i metody.

Język wysokiego poziomu – składnia języka Python jest zrozumiała przez człowieka na poziomie przeglądania kodu źródłowego (słowa kluczowe to wyrazy w języku angielskim).

Język ogólnego przeznaczenia - pozwala tworzyć zarówno proste skrypty jak i skomplikowane aplikacje, wieloplatformowy, możliwość współdziałania z innymi językami programowania (język skryptowy w aplikacjach).

Cechy Pythona

Wieloplatformowość - aplikacje z powodzeniem mogą być uruchamiane na różnych systemach bez konieczności zmian w kodzie źródłowym (musi być dostępny interpreter).

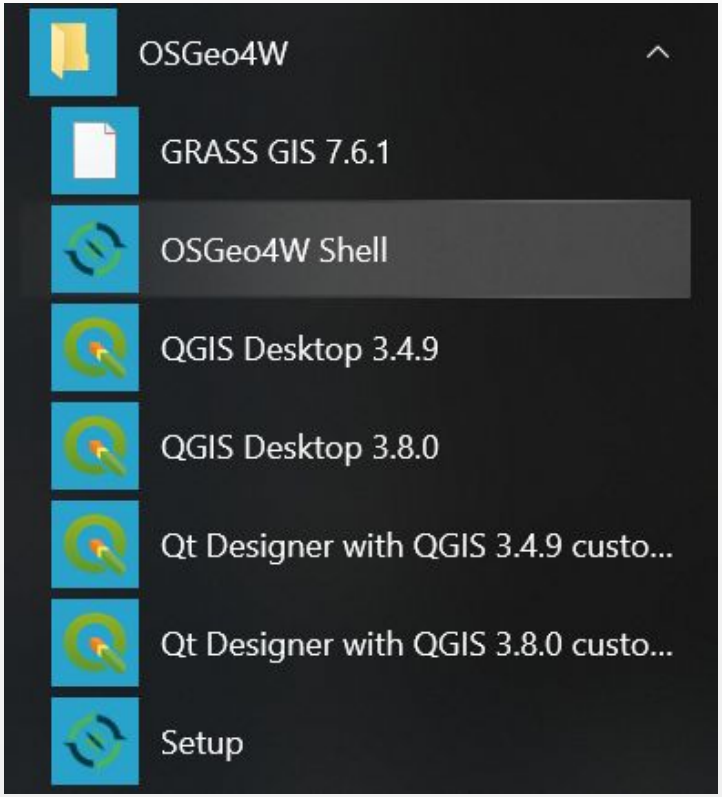
Typowanie dynamiczne - nie wymaga deklarowania typów zmiennych, są one określane przy przypisywaniu wartości do zmiennej.

Biblioteki - bogata biblioteka modułów dostarczanych wraz ze standardowym interpreterem języka (*Batteries included*), możliwość instalacji nowych modułów z repozytorium PyPi (>190 tysięcy).

Wcięcia - konieczność stosowania wcięć do wydzielania bloków kodu tj. funkcje, pętle, instrukcje warunkowe. Jest to cecha unikatowa tego języka. Stosowanie wcięć wymusza na programiście jednorodny sposób pisania kodu, dzięki czemu czytelność kodu źródłowego jest bardzo duża.

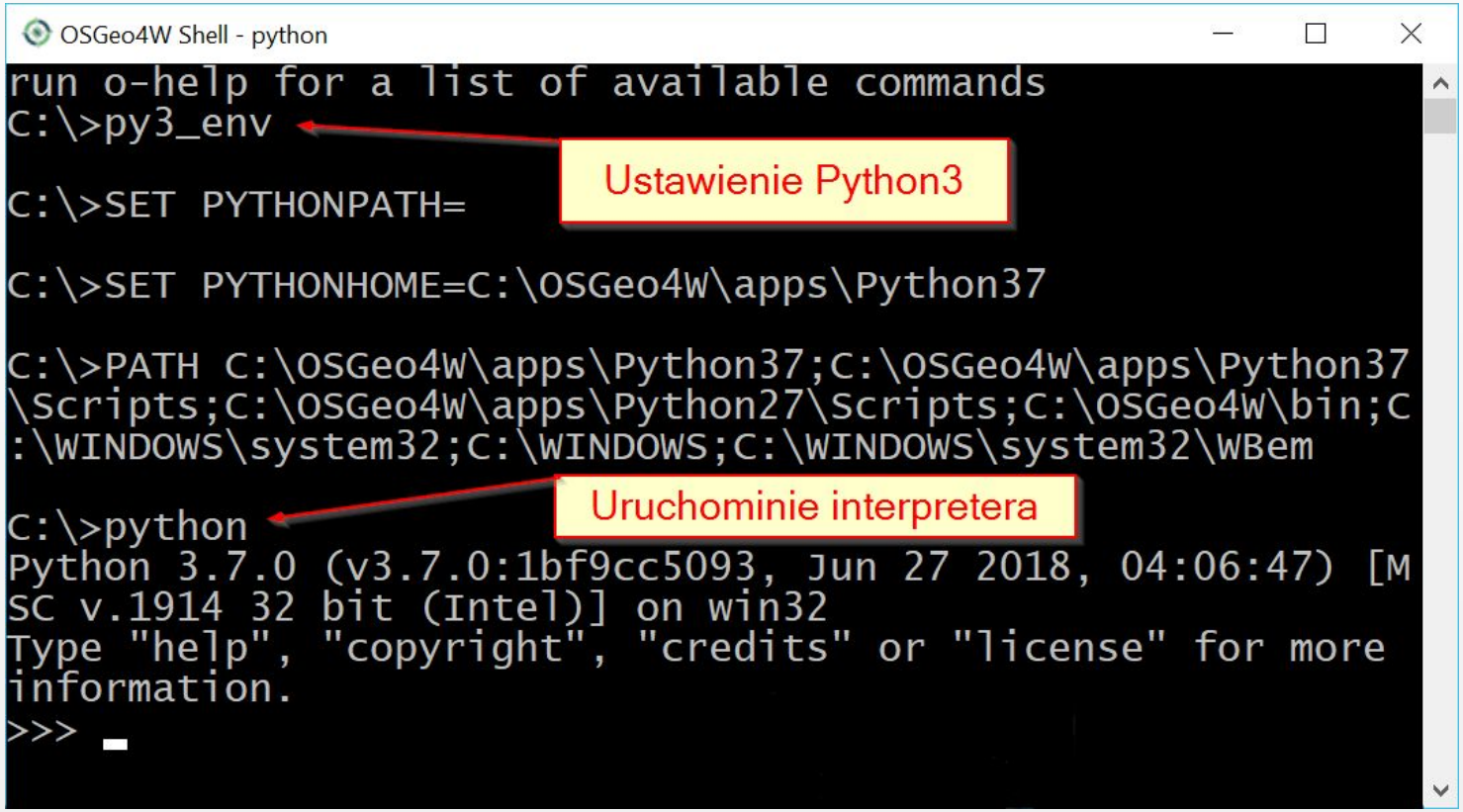
Instalacja interpretera

- Oficjalny strona: **https://www.python.org/downloads**
- **QGIS** - dostęp poprzez konsolę *OSGeo4W Shell*



OSGeo4W Shell

Domyślnie uruchamiany jest Python 2, aby skorzystać z nowszej wersji należy po uruchomieniu konsoli wpisać polecenie ***py3_env***, a następnie **python**.



Podstawowe typy danych

- Wartości logiczne – prawda/fałsz (**True**/**False**)
- Typy numeryczne – liczby
- Kolekcje – tekst, lista, słownik, tupla, …

Typy numeryczne

- **int** – liczby całkowite o dowolnej wielkości

```
>>> type( 5 )
<class 'int'>
```

- **float** – liczby zmiennoprzecinkowe (część dziesiętna jest zawsze za kropką)

```
>>> type( 1.23 )
<class 'float'>
```

- **complex** – liczby zespolone np. 3+2.7j

Zmienne

Przypisanie wartości do zmiennej odbywa się pojedynczym znakiem równości:

```
a = 5
```

Nazwy zmiennych:

- nie mogą zawierać specjalnych znaków np. diakrytycznych, spacji, tabulatorów itp.;
- nie mogą rozpoczynać się od liczby np. 1liczba;
- wielkość liter ma znaczenie, tzn. zmienna X i x są traktowane jako różne obiekty.

print

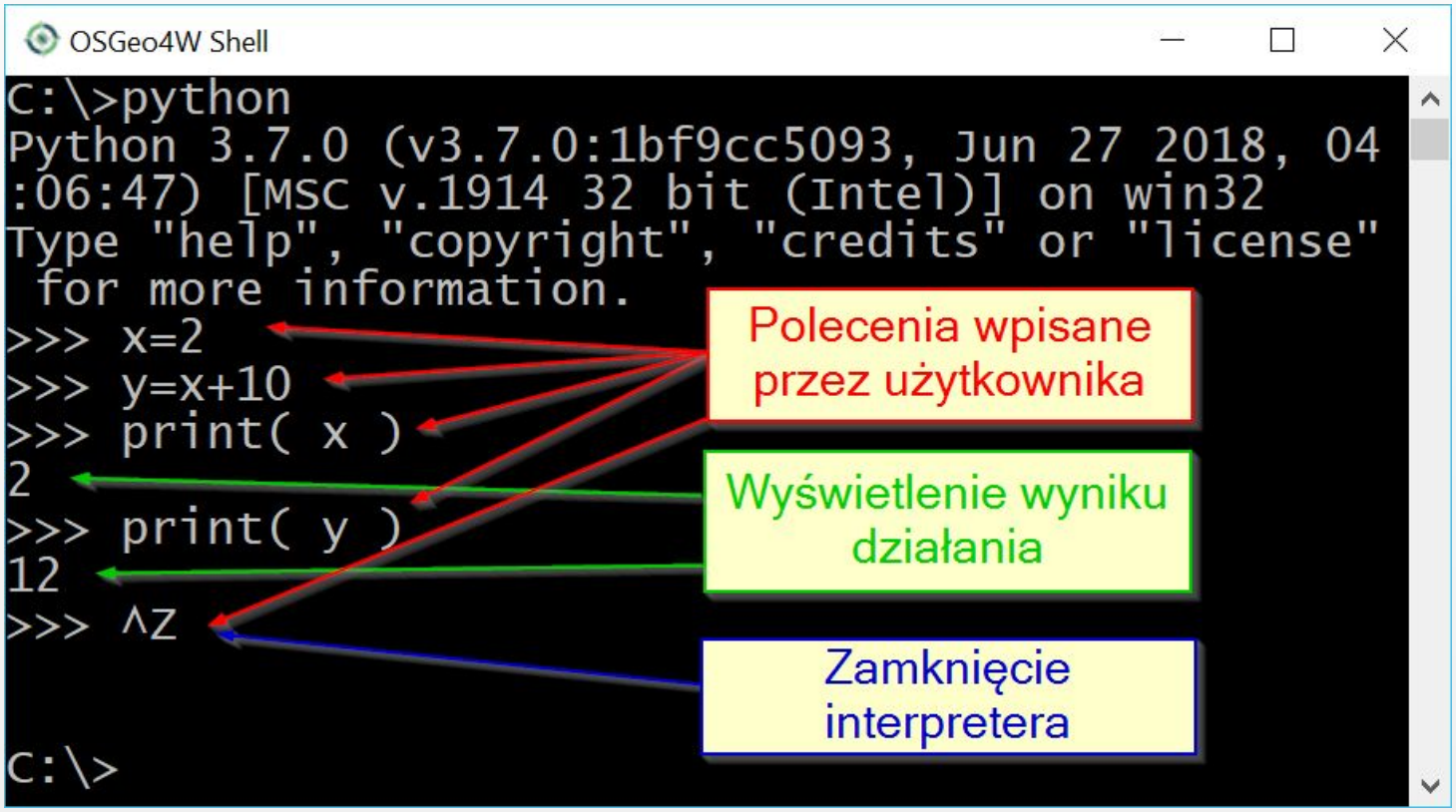
Funkcja **print** pozwala wydrukować tekst w konsoli. Możliwe jest podanie dowolnego obiektu Pythona, przecinkami można oddzielić kolejne elementy do wydrukowania.

```
>>> print( 'tekst' )
'tekst'

>>> print( 1,2,3 )
1 2 3

>>> print( 1,'tekst' )
1 'tekst'
```

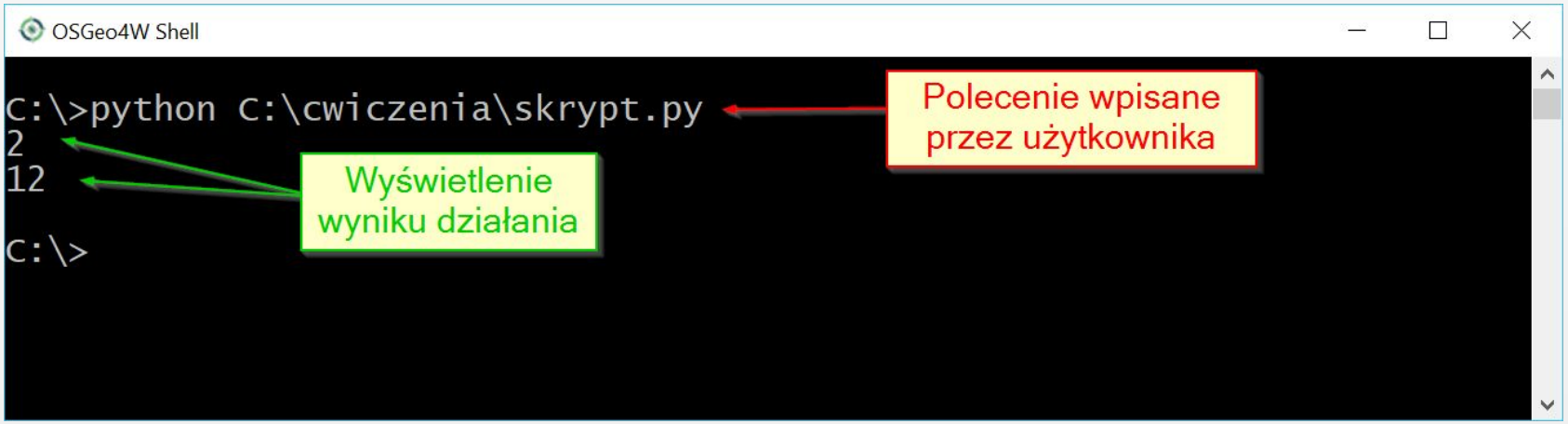
Po uruchomieniu interpretera Pythona w konsoli polecenia wpisujemy po znakach >>>. Wynik działania operacji jest wyświetlany na bieżąco. Aby zamknąć interpreter należy wpisać *Ctrl+Z* i zatwierdzić klawiszem *Enter*.



Skrypty i moduły

Kod źródłowy w języku Python piszemy w plikach tekstowych z rozszerzeniem .py. Taki plik może zostać uruchomiony bezpośrednio w konsoli, mówimy w takim wypadku o *skrypcie.*

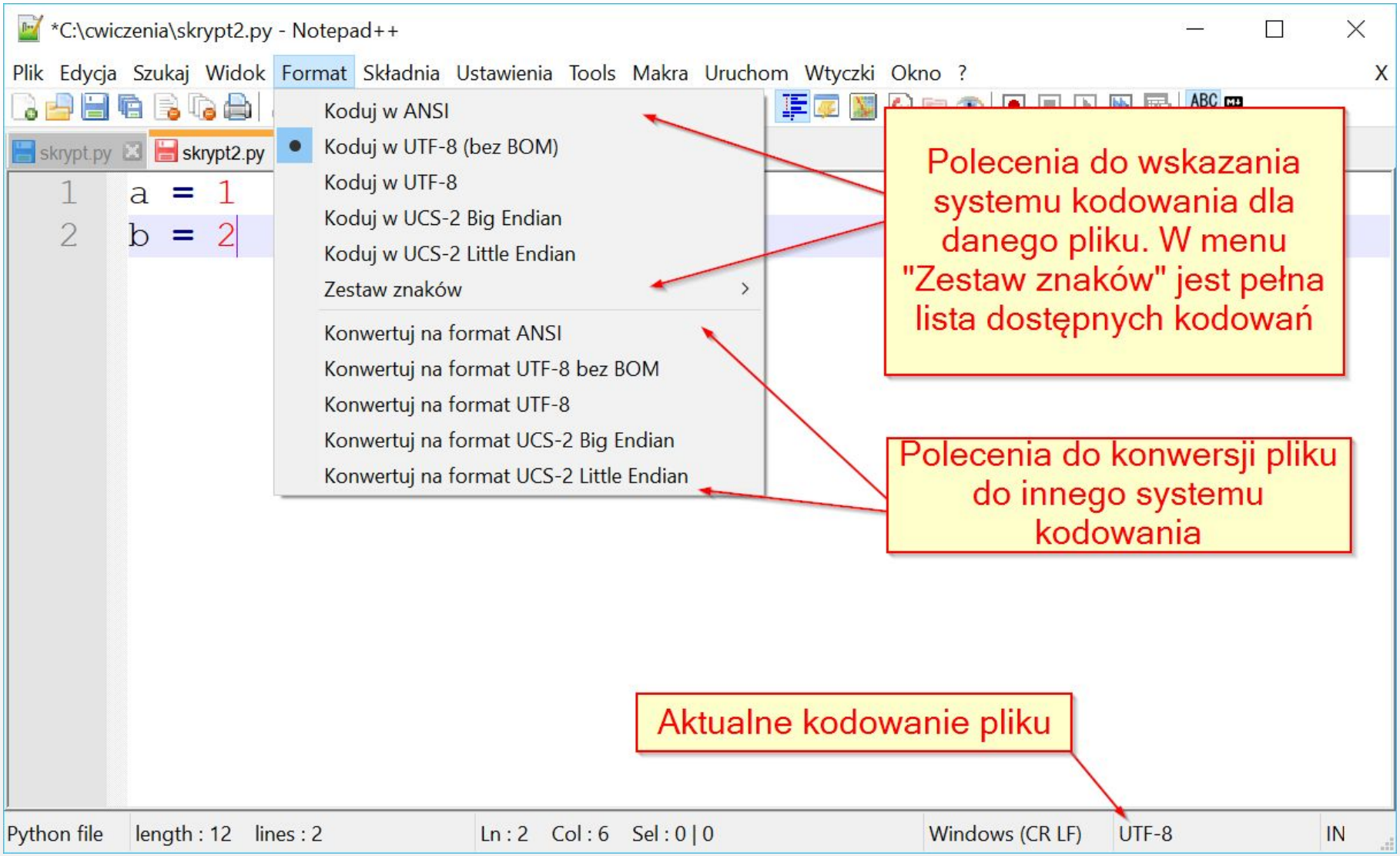
```
1  x = 2
2  y = x+10
3  print( x )
4  print( y )
```



Pliki tekstowe nie mają informacji o tym w jakim systemie kodowania zapisane są znaki. Interpreter Pythona wymaga jednak takiej informacji. W tym celu na początku pliku należy go określić:

```
# coding: utf-8
# coding: windows-1250
# coding: iso-8859-2
```

Kodowanie znaków w Notepad++



Konsola Pythona w QGIS

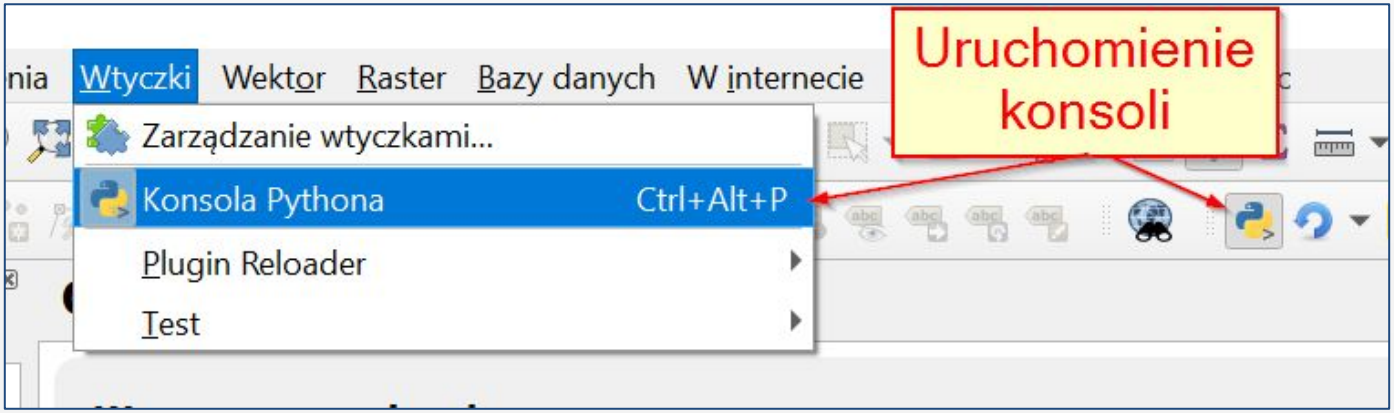
Aplikacja QGIS wykorzystuje Pythona jako język skryptowy. Wbudowana *Konsola Pythona* to nic innego jak nakładka graficzna na interpreter tego języka. Umożliwia ona pisanie kodu i wywoływanie go w interpreterze w dwóch trybach:

- wiersz poleceń umożliwiający wykonywanie poleceń po wpisaniu ich w konsoli,
- *Edytor skryptów* umożliwiający pisanie i uruchamianie plików z kodem źródłowym.

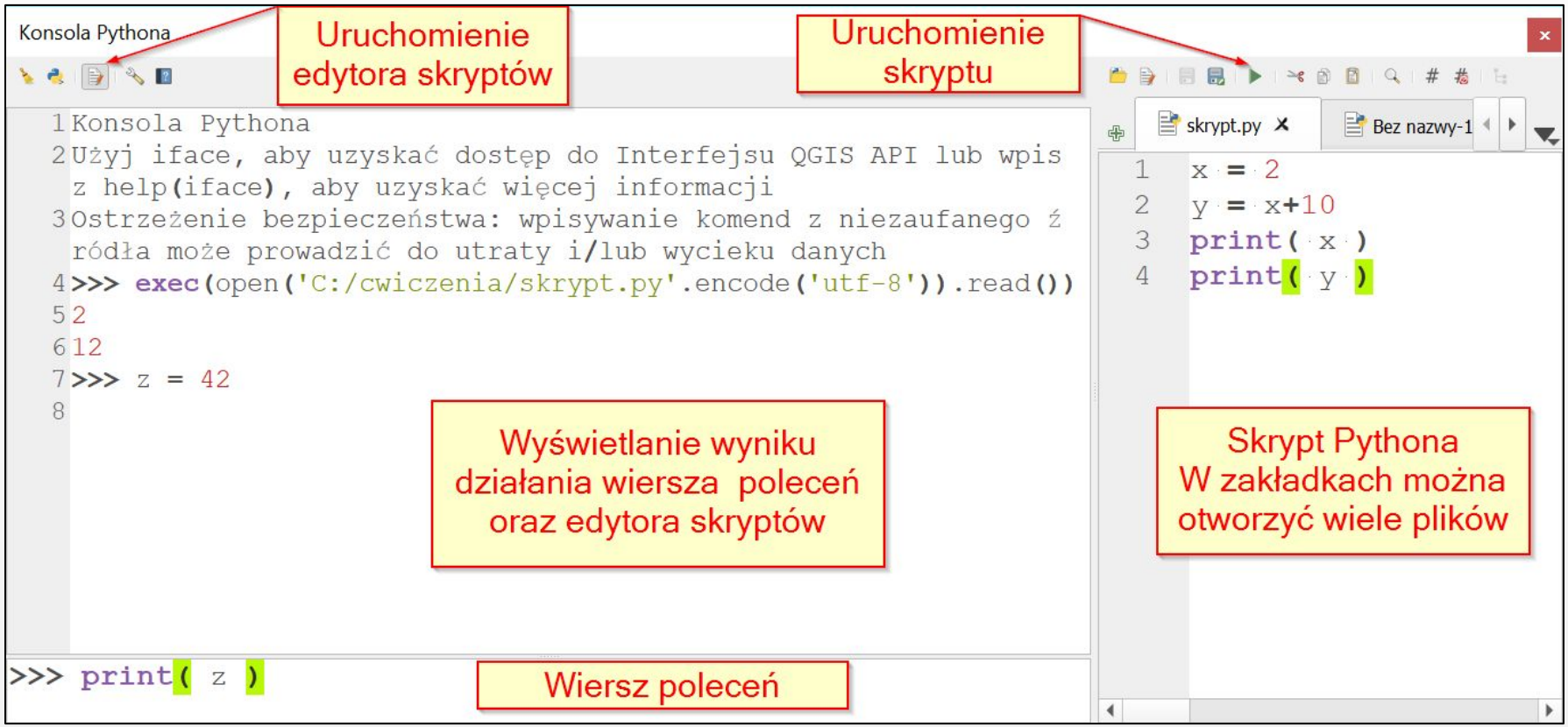
Oba narzędzia są ze sobą zintegrowane i uruchomione we wspólnym środowisku. Oznacza to, że zmienne, importy itp. zdefiniowane w jednym z tych miejsc są również widoczne w drugim.

Konsola Pythona w QGIS

Aby uruchomić konsolę należy w QGIS z menu *Wtyczki* wybrać polecenie *Konsola Pythona* lub kliknąć odpowiedni przycisk na pasku narzędzi.



Konsola Pythona w QGIS



Komentarze

Fragmenty kodu oznaczone jako komentarze są ignorowane przez interpreter.

Komentarze jednolinijkowe rozpoczynamy znakiem # (hash):

```
#Cała ta linia jest komentarzem
x = 5   #Komentarz w linii z kodem
```

Komentarze wielolinijkowe rozpoczynamy i kończymy potrójnym apostrofem lub cudzysłowem:

```
""" Pierwsza linijka
Druga linijka """
```

Funkcje wbudowane

https://docs.python.org/3/library/functions.html

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

type

Funkcja **type** zwraca typ danych danego obiektu. Wszystkie obiekty Pythona mogą być użyte jako argument funkcji, również inne funkcje.

```
>>> type('tekst')
<class 'str'>

>>> a = 2
>>> type( a )
<class 'int'>

>>> type( a ) is int
True
```

dir

Funkcja **dir** zwraca listę atrybutów i metod danego obiektu, wszystkie obiekty Pythona mogą być użyte jako argument funkcji. Nazwy otoczone podwójnym podkreśleniem dotyczą metod systemowych (tzw. metody magiczne).

```
>>> dir( 'tekst' )
['__add__', '__class__', '__contains__',
'__delattr__', '__dir__', '__doc__',...,
'capitalize', 'center', 'count', ...]
```

Moduły

Moduły to pliki z rozszerzeniem .py, które można zaimportować z poziomu skryptu lub innych modułów. Zawiera on obiekty (zmienne, funkcje, klasy, itp.), które mogą być dowolnie wykorzystywane w różnych aplikacjach.

Importowanie należy wykonywać jak najwcześniej w kodzie źródłowym.

Importowanie

- **import** *nazwa_modulu*

Importowany jest cały moduł jako pojedynczy obiekt.

```
>>> import math
>>> print( math.pi )
3.141592653589793
>>> print( math.cos( math.pi ) )
-1
```

Funkcje trygonometryczne z modułu `math` wykonują operacje na kątach podawanych w radianach.

Importowanie

- **from** *nazwa_modułu* **import** *nazwa_obiektu / **

Importowane są tylko wskazane obiekty z modułu lub wszystko co w nim jest (*).

```
>>> from math import pi, cos
>>> cos( pi )
-1.0

>>> from math import *
>>> cos( pi )
-1.0
```

Paczki

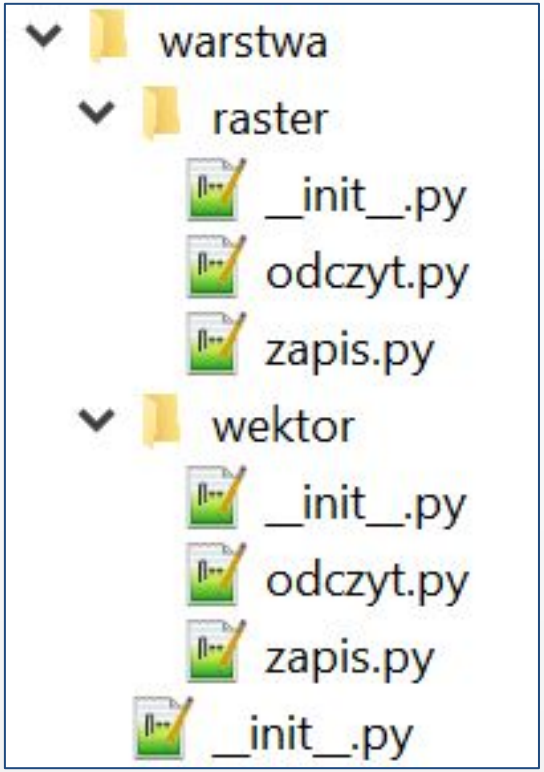
Paczki (ang. *package*) pozwalają na hierarchiczne usystematyzowanie wielu modułów. Dzięki temu kod jest rozdzielony na osobne pliki zawierające logiczne części całości.

Sama paczka to katalog, w którym znajdują się pliki z rozszerzeniem .py. Może on zawierać podkatalogi, zawierające kolejne pliki.

W każdym katalogu paczki powinien znajdować się dodatkowy, pusty plik *__init__.py*. Są one jednak opcjonalne od Pythona 3.3, we wcześniejszych wersjach są one obowiązkowe.

Importowanie obiektów z paczki jest rozbudowaną wersją importowania obiektów z pojedynczego modułu. Możliwe jest zaimportowanie całej paczki, jej części, pojedynczego modułu lub wybranych obiektów z danego pliku. Niezależnie od wybranego sposobu poszczególne katalogi/pliki należy rozdzielić kropką.

Paczki




```
import warstwa
warstwa.raster.odczyt.TIFF('C:\plik.tif')

from warstwa.raster import odczyt
odczyt.TIFF('C:\plik.tif')

from warstwa.raster.odczyt import TIFF
TIFF('C:\plik.tif')

from warstwa.wektor import *
odczyt.SHP('C:\dzialki.shp')
```

Instalacja nowych bibliotek

Python posiada bogatą bibliotekę dodatkowych modułów, które można pobrać z oficjalnego repozytorium *Python Package Index* (**https://pypi.org/**). Najprostszym sposobem instalacji jest wykorzystanie narzędzia **pip**, które uruchamia się podając je jako parametr przy uruchamianiu interpretera z dodatkowym argumentem *-m*.

Instalacja nowej biblioteki

```
> python -m pip install nazwa
```

Aktualizacja biblioteki

```
> python -m pip install --upgrade nazwa
```

Usunięcie biblioteki

```
> python -m pip uninstall nazwa
```

Konwersja między typami danych

Funkcje do konwersji podstawowych typów danych mają takie same nazwy jak typ, do którego chcemy przekonwertować dane. Jeśli konwersja nie jest możliwa to wystąpi błąd.

```
>>> a = 1
>>> b = str( a )
>>> type(b)
<class 'str'>
>>> c = float( b )
>>> print( c )
1.0
```

```
>>> lista = [ 1, 2, 3 ]
>>> tupla = tuple(lista)
>>> print( tupla )
(1, 2, 3)
>>> list( 'abc' )
[ 'a', 'b', 'c' ]
```

Wartości logiczne (bool)

`True` – logiczna prawda
`False` – logiczny fałsz

Dla podstawowych typów danych Pythona fałszem logicznym są:

- **`False`**,
- **`None`** – specjalny typ reprezentujący w Pythonie brak wartości,
- liczba zero (`0, 0.0, 0L, 0j`),
- puste kolekcje (`[], tuple(), {}, ""`, itp.).

Prawda logiczna – pozostałe przypadki

Operatory logiczne

P = **True**
F = **False**

Operator	Znaczenie	Przykład
and	Koniunkcja, wyrażenie jest prawdziwe gdy oba zdania są prawdziwe.	P **and** F → **False**
or	Alternatywa, wyrażenie jest fałszywe gdy oba zdania są fałszywe	P **or** F → **True**
not	Zaprzeczenie, odwraca wartość logiczną zdania	**not** P → **False**

Operatory arytmetyczne

Operator	Znaczenie	Przykład
+	suma	`10 + 2.5 → 12.5`
-	różnica	`3.6 - 7 → -3.4`
*	iloczyn	`10 * 5 → 50`
/	iloraz (dzielenie zawsze zwraca typ **float**)	`12 / 3 → 4.0` `10 / 3 → 3.3333`
//	zwraca podstawę ilorazu	`10 // 3 → 3`
%	reszta z dzielenia (modulo)	`10 % 3 → 1`
**	potęgowanie	`3 ** 2 → 9`

Operatory przypisania

Operator	Znaczenie	Przykład
=	Przypisanie wartości	`a = 1`
+=	Dodanie i przypisanie nowej wartości	`a += 1` jest równoważne `a = a + 1`
-=	Odjęcie i przypisanie nowej wartości	`a -= 1` jest równoważne `a = a - 1`
*=	Mnożenie i przypisanie nowej wartości	`a *= 1` jest równoważne `a = a * 1`
/=	Dzielenie i przypisanie nowej wartości	`a /= 3` jest równoważne `a = a / 3`
//=	Dzielenie bez reszty i przypisanie nowej wartości	`a //= 3` jest równoważne `a = a // 3`
%=	Reszta z dzielenia i przypisanie nowej wartości	`a %= 3` jest równoważne `a = a % 3`
**=	Potęgowanie i przypisanie nowej wartości	`a **= 3` jest równoważne `a = a ** 3`

Operatory porównania

```
a = 1
b = 2
```

Operator	Znaczenie	Przykład
`<`	mniejszy, zwraca prawdę jeśli lewa strona wyrażenia jest mniejsza od prawej	`a < b` → **`True`**
`<=`	mniejszy lub równy, zwraca prawdę jeśli lewa strona wyrażenia jest mniejsza lub równa prawej	`a <= b` → **`True`**
`>`	większy, zwraca prawdę jeśli lewa strona wyrażenia jest większa od prawej	`a > b` → **`False`**
`>=`	większy lub równy, zwraca prawdę jeśli lewa strona wyrażenia jest większa lub równa prawej	`a >= b` → **`False`**
`==`	równy, zwraca prawdę jeśli obie strony są takie same	`a == b` → **`False`**
`!=`	nierówny, zwraca prawdę jeśli obie strony różnią się	`a != b` → **`True`**

Wcięcia kodu

- Wcięciem może być spacja lub tabulator.
- Standardem jest definiowanie jako pojedynczego poziomu wcięcia 4 spacji.
- W pojedynczym pliku wcięcia muszą być takie same, nie można np. mieszać spacji i tabulatorów, ponieważ może to wywołać błąd **IndentationError** lub **TabError**.

Wcięcia kodu

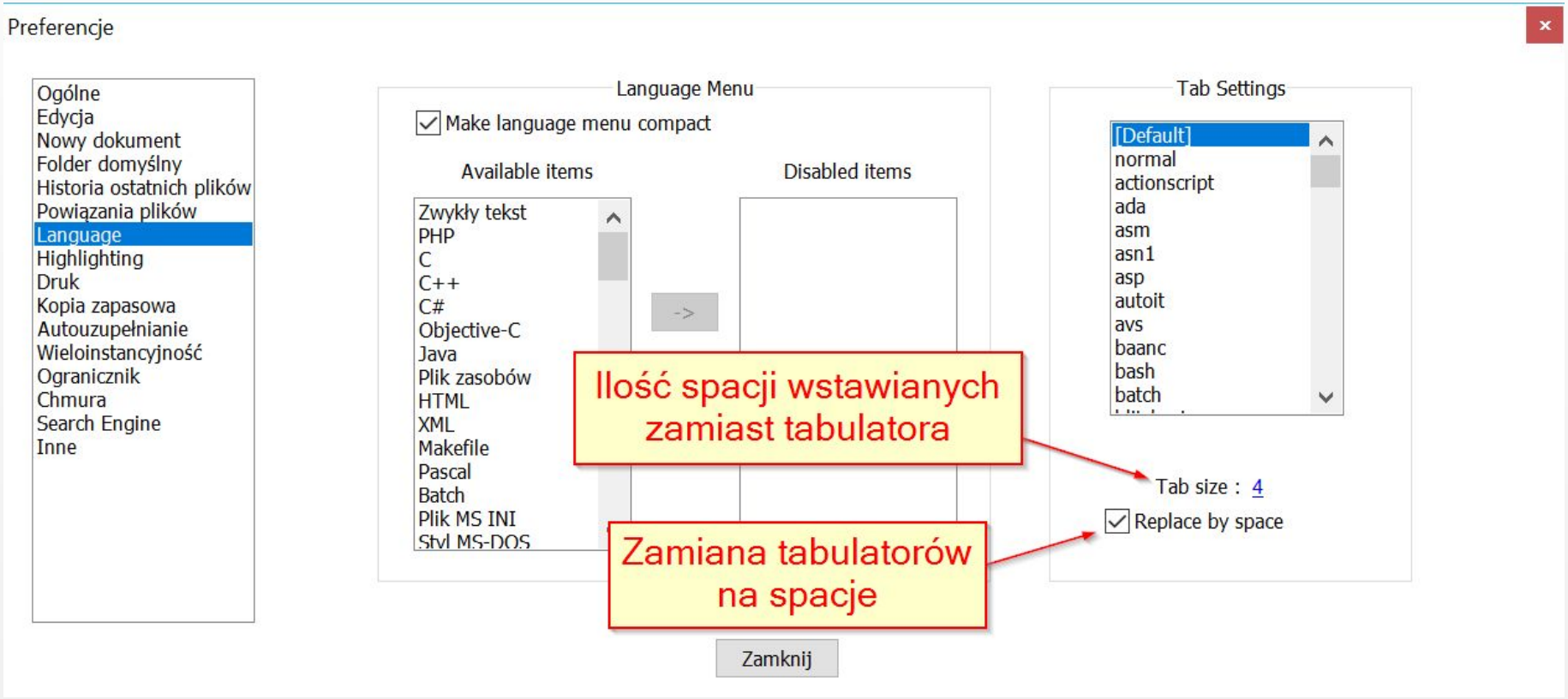
- Wcięcia kodu definiują bloki kodu.
- W Pythonie wcięcia stosuje się w pętlach, warunkach, wyjątkach, funkcjach i klasach.
- Wcięcie następuje jeśli wcześniejsza linijka kończy się dwukropkiem.

```
def funkcja(a):#Dwukropek, nowa linia musi być wcięta
    wynik = a**2#Początek bloku kodu
    return wynik#Koniec bloku kodu

#Po końcu bloku kodu wracamy do poprzedniego poziomu
b = funkcja( 3 )
```

Wcięcia kodu w Notepad++

Ustawienia wcięć w aplikacji Notepad++ znajdują się w menu *Ustawienia* → *Preferencje*, zakładka *Language*.



Wystąpienie wyjątku oznacza błąd programu i przerwanie jego działania. Lista wbudowanych typów błędów: *https://docs.python.org/3/library/exceptions.html*

```
try:
   <blok kodu>
except [TYP_WYJĄTKU [ as KLASA_BŁĘDU ]]:
   print( "Wystąpił problem: ", KLASA_BŁĘDU )
```

Wyjątki **except**

```
try:
   print( 2 / 0 )
   print( int('tekst') )
except: #bez podania typu zostaną wyłapane
wszystkie błędy
   print( "Błąd: ", error )

try:
   print( 2 / 0 )
   print( int('tekst') )
except ZeroDivisionError:
   print( "Błąd, dzieleni przez zero" )
except ValueError:
   print( "Błąd, zła wartość" )
```

Wyjątki **finally**

finally – blok kodu jest wykonywany niezależnie od tego czy błąd wystąpi czy nie.

```
try:
   print( 5 / 2 )
except:
   print( 'błąd' )
finally:
   print( 'koniec' )
```

Wyjątki **else**

else – wykonuje blok kodu tylko jeśli nie wystąpi błąd

```
try:
   print( 5 / 2 )
except:
   print( 'błąd' )
else:
   print( 'bez błędu' )
```

Instrukcja warunkowa **if**

Jeden z podstawowych elementów większości języków programowania. Pozwala wykonać blok kodu jeśli zdanie logiczne (warunek) zwraca prawdę logiczną.

```
if <zdanie logiczne>:
    <instrukcja 1>    #warunek jest prawdziwy

#Moduł random służy do generowania losowych liczb
import random
#randint zwraca losową liczbę całkowitą z podanego zakresu
a = random.randint( 0, 10 )
if a > 5:
   print( 'Wylosowano liczbę większą od 5' )
```

Instrukcja warunkowa **if**

Blok po **`else`** jest wykonywany tylko w przypadku gdy żaden wcześniejszy warunek nie jest spełniony.

```
if <zdanie logiczne>:
    <instrukcja 1>  #warunek jest prawdziwy
else:
   <instrukcja2>    #warunek nie jest prawdziwy
```

```
a = random.randint( 0, 10 )
if a > 5:
   print( 'Liczba większa od 5' )
else:
   print( 'Liczba mniejsza lub równa 5' )
```

Instrukcja warunkowa **if**

```
if (pierwszy warunek):
    <instrukcja 1>#pierwszy warunek jest prawdziwy
elif (drugi warunek):
    <instrukcja 2>#drugi warunek jest prawdziwy
elif (…):           #bloków elif może być dowolna ilość
    <instrukcja 3>
else:     #żaden powyższy warunek nie jest spełniony
    <instrukcja 4>
```

Bloki **elif** i **else** są opcjonalne.
Jeśli dany warunek jest spełniony to następujące po nim nie są już sprawdzane.

Instrukcja warunkowa **if**

```
import random

#Wylosowanie liczby z zakresu 0-10
a = random.randint( 0, 10 )

if a == 0:
   print( 'Wylosowano zero' )
elif a%2 == 0:
   print( a, 'liczba jest parzysta' )
else:
   print( a, 'liczba jest nieparzysta' )
```

Pętla **while**

Pętla **`while`** jest wykonywana dopóki podany warunek jest prawdziwy. Należy uważać na nieskończone wykonywanie pętli jeśli warunek nigdy nie będzie fałszywy.

```
while (warunek jest spełniony):
    <wykonuj instrukcję>
```

```
i = 0
while i < 3:
   print( i )
   i += 1
→ 0
→ 1
→ 2
```

Pętla **for**

Iterator to obiekt zawierający policzalną liczbę elementów, do których możliwy jest sekwencyjny dostęp. W Pythonie jest to możliwe za pomocą pętli **`for`**, gdzie każda iteracja zwraca pojedynczy element.

```
for element in iterator:
   print( element )
```

Iterator może być zbiorem istniejących obiektów (kolekcje) lub generować zwracane elementy na bieżąco.

Funkcja **range**

Funkcja range zwraca iterator, zwracający po kolei liczby całkowite wg podanych kryteriów.

```
for liczba in range( 3 ):
   print( liczba )
→ 0
→ 1
→ 2
```

Funkcja range może przyjąć dodatkowe argumenty:

- `range(2, 6)` - zwraca ciąg liczb od 2 do 5
- `range(3, 10, 2)` - zwraca co drugą liczbę w zakresie od 3 do 9

Kolekcje

- Łańcuch znaków
- Listy
- Tuple (krotki)
- Słowniki

Odwołanie do konkretnego elementu powyższych kolekcji odbywa się wg indeksu podawanego w nawiasie kwadratowym:

```
>>> obiekt[indeks]
wartość
```

Indeksem jest liczba porządkowa elementu w uporządkowanej kolekcji (tekst, lista, tupla; pierwszy element ma indeks 0) lub klucz (słownik)

Łańcuch znaków **str**

- niezmienne – nie można dodawać/usuwać/zmieniać tekstu, jakakolwiek zmiana powoduje utworzenie nowego obiektu,
- kolejność elementów ma znaczenie,
- przechowuje znaki tekstowe,
- definiuje się apostrofem (‘tekst’) lub cudzysłowem (“tekst”)

```
#stworzenie nowego łańcucha typu str
s = 'tekst'
#pierwszy znak ma indeks 0, drugi 1 itd.
print( s[2] )
→ 'k'
```

Formatowanie łańcucha znaków

```
imie = 'Jan'
nazwisko = 'Nowak'

tekst = „Imię: “ + imie + “, nazwisko: ” + nazwisko
```

lub

```
tekst = „Imię: %s, nazwisko: %s” % ( imie, nazwisko )
```

lub

```
tekst = „Imię: {}, nazwisko: {}”.format( imie, nazwisko )
```

lub

```
tekst = „Imię: {1}, nazwisko: { 0 }”.format( nazwisko, imie )
```

lub

```
tekst = „Imię: { surname }, nazwisko: { name }”.format(
    surname=nazwisko, name=imie )
```

Wynik:

```
print( tekst )
→ 'Imię: Jan, nazwisko: Nowak'
```

Formatowanie liczb w tekście

```
>>> „pozycja {} z {:d}”.format(2, 12)
‘pozycja 2 z 12’

#domyślnie liczby rzeczywiste wyświetlenie są z
dokładnością do sześciu miejsc po przecinku
>>> 'Cena towaru: {:f}'.format( 3.4 )
'Cena stara: 3.400000'

>>> 'Cena towaru: {:.2f}'.format(3.4)
'Cena nowa: 3.40'

>>> 'Zmiana: {:+d}'.format( 42 )   #można
określić znak pokazywany przed liczbą
'Zmiana: +42'
```

Znaki ucieczki

Znak \ w łańcuchu znaków umożliwia wstawianie znaków specjalnych takich jak znak nowej linii czy tabulatury. Litera po znaku \ informuje interpreter o tym jaki znak ma zostać wprowadzony.

Główne znaki ucieczki:

- \n - znak nowej linii
- \t - tabulator
- \’ - apostrof
- \” - cudzysłów
- \\ - ukośnik

Znaki ucieczki

```
>>> print( 'pierwsza linijka\ndruga linijka' )
pierwsza linijka
druga linijka

>>> print( 'Apostrof \' i cydzysłów " ' )
Apostrof ' i cydzysłów "

>>> print( 'C:\testy' )
C:    esty  #Niepoprawny tekst
>>> print( 'C:\\testy' )
C:\testy
```

Można "wyłączyć" znaki ucieczki dodając przed tekstem literę *r* :

```
>>> print( r'C:\testy' )
C:\testy
```

Dzielenie i łączenie tekstu

- **`split(sep)`** – tworzy listę z tekstu poprzez jego rozdzielenie podanym separatorem `sep`

```
tekst = 'jeden,dwa,trzy'
tekst.split(',')
→ ['jeden', 'dwa', 'trzy']
```

- **`join(seq)`** – łączy elementy kolekcji `seq` za pomocą separatora, którym jest łańcuch wywołujący metodę

```
lista = ['jeden', 'dwa', 'trzy']
';'.join(lista)
→ 'jeden;dwa;trzy'
```

Lista **list**

- edytowalna – można dodawać i usuwać elementy,
- kolejność elementów ma znaczenie,
- może przechowywać dowolny obiekt Pythona (także inne listy).
- definiuje się kwadratowymi nawiasami `[]` lub funkcją `list()`

```
>>> lista = [1, 2.5, 'trzy']
>>> lista
[1, 2.5, 'trzy']
```

Indeksacja

```
lista = [1,2,3,4,5,6]
print( lista[1] ) #zwraca drugi element listy (pierwszy ma indeks 0)
→ 2

print( lista[10] )   #błąd - brak elementu o tym indeksie
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

print( lista[-1] )   #ujemny indeks - elementy liczone są od końca (indeks -1 ma ostatni element, -2 przedostatni itd.)
→ 6

lista[1] = 'dwa'  #Aby zmienić konkretny element należy przypisać wartość podając jego indeks
print( lista )
→ [1,'dwa',3,4,5,6]
```

Przeszukiwanie listy

```
lista = [1, „gis”, [45, 2], 'z', 'gis']
```

- **index -** wyszukanie indeksu podanego elementu, zwraca indeks pierwszego wystąpienia danej wartości w liście

```
print( lista.index('gis') )
→ 1

#jeśli elementu nie ma w liście zwracany jest błąd
print( lista.index('tekst') )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'tekst' not in list
```

- sprawdzenie czy dany element jest w liście, zwracana jest wartość logiczna: **`True`** – element jest na liście lub **`False`** – nie znaleziono.

```
print( 'tekst' in lista )
→ False
```

Dodawanie nowych elementów

```
lista1 = [1,3]
lista2 = [5, 6, 7, 8, 9, 10]
```

- **append(x)** - dodanie pojedynczego elementu `x` na koniec listy:

```
lista1.append( 4 )
print( lista1 )
→ [1, 3, 4]
```

- **insert(index, x)** - dodanie obiektu `x` na pozycji `index`:

```
lista1.insert(1, '2')
print( lista1 )
→ [1, '2', 3, 4]
```

- **extend(x)** - rozszerzenie listy elementami z innej listy `x`

```
lista1.extend(lista2)
print( lista1 )#zmieniona lista
→ [1, '2', 3, 4, 5, 6, 7, 8, 9, 10]
print( lista2 )#lista2 nie jest modyfikowana
→ [5, 6, 7, 8, 9, 10]
```

Usuwanie elementów listy

```
lista = [‘jeden’, ‘dwa’, ‘pięć’, ‘cztery’,
‘pięć’, ‘sześć’]
```

- **`remove`** - usuwanie elementu wg wartości:

```
lista.remove(‘pięć’)
print( lista )
→ [ ‘jeden’, ‘dwa’, ‘cztery’, ‘pięć’, ‘sześć’ ]
```

Jeśli elementu nie ma na liście to wystąpi błąd *ValueError*.

- **`del`** - usuwanie elementu wg indeksu:

```
del lista[0]
lista
→ [ ‘dwa’, ‘cztery’, ‘pięć’, ‘sześć’ ]
```

Jeśli indeks jest poza zasięgiem listy to zostanie zwrócony błąd *IndexError*.

Tupla, krotka **tuple**

- niezmienne – w utworzonej tupli nie można modyfikować elementów,
- kolejność elementów ma znaczenie,
- może przechowywać dowolny obiekt Pythona
- definiowana jest okrągłymi nawiasami lub funkcją `tuple()`

```
tupla = (1, 'dwa', 3., 4, 'dwa')
print( tupla[0] )
→ 1
print( tupla[2:5] )
→ (3.0, 4, 'dwa')
print( tupla.index('dwa')
→ 1
print( tupla.count('dwa')
→ 2
```

Słownik **dict**

- pojedynczy element słownika stanowi relacja *klucz:wartość*,
- obiekty (wartości) indeksowane są za pomocą kluczy,
- edytowalna – można dodawać i usuwać elementy,
- kolejność elementów nie ma znaczenia – dostęp do elementów odbywa się poprzez podanie klucza,
- wartością może być dowolny obiekt Pythona, kluczem tylko obiekty niezmienne m.in. łańcuch znaków, liczba, tupla

Dostęp do elementów słownika

Dostęp do elementów polega na podaniu nazwy klucza. W słownikach pary elementów nie zachowują kolejności przy ich dodawaniu, więc nie ma do nich dostępu przez podanie indeksu.

```
slownik = {1:'a', 2:'b', 'c':3.5}
print( slownik['c'] )
→ 3.5

print( slownik[3.5] )    #podanie nieistniejącego
klucza spowoduje wywołanie błędu
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 3.5

print( 'c' in slownik )  #czy słownik posiada
podany klucz
→ True
```

Edycja elementów słownika

```
slownik = {'a':1, 'b':2, 'c':3.5}
```

- Edycja wartości podanego klucza

```
slownik['a'] = 11
→ {'a':11, 'b':2, 'c':3.5}
```

- Dodanie nowego elementu

```
slownik[3] = 'c'
→ {'a':11, 'b':2, 'c':3.5, 3: 'c'}
```

- Usunięcie elementu

```
del slownik['c']
→ {'a':11, 'b':2, 3: 'c'}
```

Widoki słownika

Poniższe metody zwracają tzw. widoki (*views*), które można przekonwertować na listę

```
slownik = { 1:'aa', 2:'b', 'c':3.5 }
```

- **keys()** – lista z kluczami słownika w losowej kolejności

```
print( list( slownik.keys() ) )
→ [2, 'c', 1]
```

- **values()** – zwraca listę z wartościami słownika

```
print( list( slownik.values() ) )
→ ['aa', 3.5, 'b']
```

- **items()** – zwraca listę elementów (klucz, wartość)

```
print( list( slownik.items() ) )
→ [(1, 'aa'), (2, 'b'), ('c', 3.5)]
```

len

Funkcja **len** zwraca długość (liczbę elementów) danego obiektu, jako argument można podać m.in. dowolną kolekcję (lista, tupla, słownik) lub łańcuch znaków.

```
len('tekst')
→ 5

lista = [1, 2, 3, 'cztery']
print( len(lista) )
→ 4

print( len({1:'a', 2:'b', 3:'c'}) )
→ 3
```

Iteracja po kolekcji

```
lista = [1, 2, 3]
for element in lista:
   print( „Liczba {}”.format( element ) )
```

→ ‘Liczba 1’

c‘Liczba 2’

→ ‘Liczba 3’

Rozpakowanie kolekcji

Istnieje możliwość jednoczesnego przypisania każdego elementu kolekcji (listy, tupli) do osobnej zmiennej (tzw. rozpakowanie):

```
x, y = (1, 2)
print( x )
→ 1
print( y )
→ 2
```

Liczba elementów kolekcji i zdefiniowanych zmiennych musi być jednakowa.

enumerate

enumerate zwraca parę wartości, pierwszy element to indeks, drugi to obiekt z kolekcji.

```
lista = ['jeden', 'dwa', 'trzy']
print( list( enumerate( lista ) ) )
→ [(0, 'jeden'), (1, 'dwa'), (2, 'trzy')]
```

enumerate

Iteracja wraz z indeksem danego elementu

```
for indeks, wartosc in enumerate(lista):
   print( „elementem o indeksie {} jest
          ‘{}’”.format(indeks, wartosc) )
```

```
→ elementem o indeksie 0 jest ‘jeden’
→ elementem o indeksie 1 jest ‘dwa’
→ elementem o indeksie 2 jest ‘trzy’
```

Iteracja po elementach słownika

```
login = {„server”:”localhost”, „user”:”foo”,
„password”:”bar”}
```

- Iteracja po kluczach

```
for klucz in login:
    print( klucz )
```

→ 'server'
→ 'user'
→ 'password'

- Iteracja po kluczach i wartościach

```
for klucz, wartosc in login.items():
    print( "{}={}".format(klucz, wartosc) )
```

→ 'server=localhost'
→ 'user=foo'
→ 'password=bar'

continue

Słowo kluczowe **`continue`** powoduje przeskoczenie do kolejnej iteracji pętli. Można je wykorzystać w obu typach pętli.

```
for i in [1, 2, 3, 4, 5, 6]:
    if i % 2 == 1:
        continue  #przejście do kolejne iteracji
    print( i )
```

→ 2
→ 4
→ 6

break

Słowo kluczowe **`break`** powoduje natychmiastowe wyjście z pętli. Można je wykorzystać w obu typach pętli.

```
i = 0
while i < 5:
    print( i )
    if i == 2:
        break #Wyjście z pętli
    i += 1
→ 0
→ 1
→ 2
```

pass

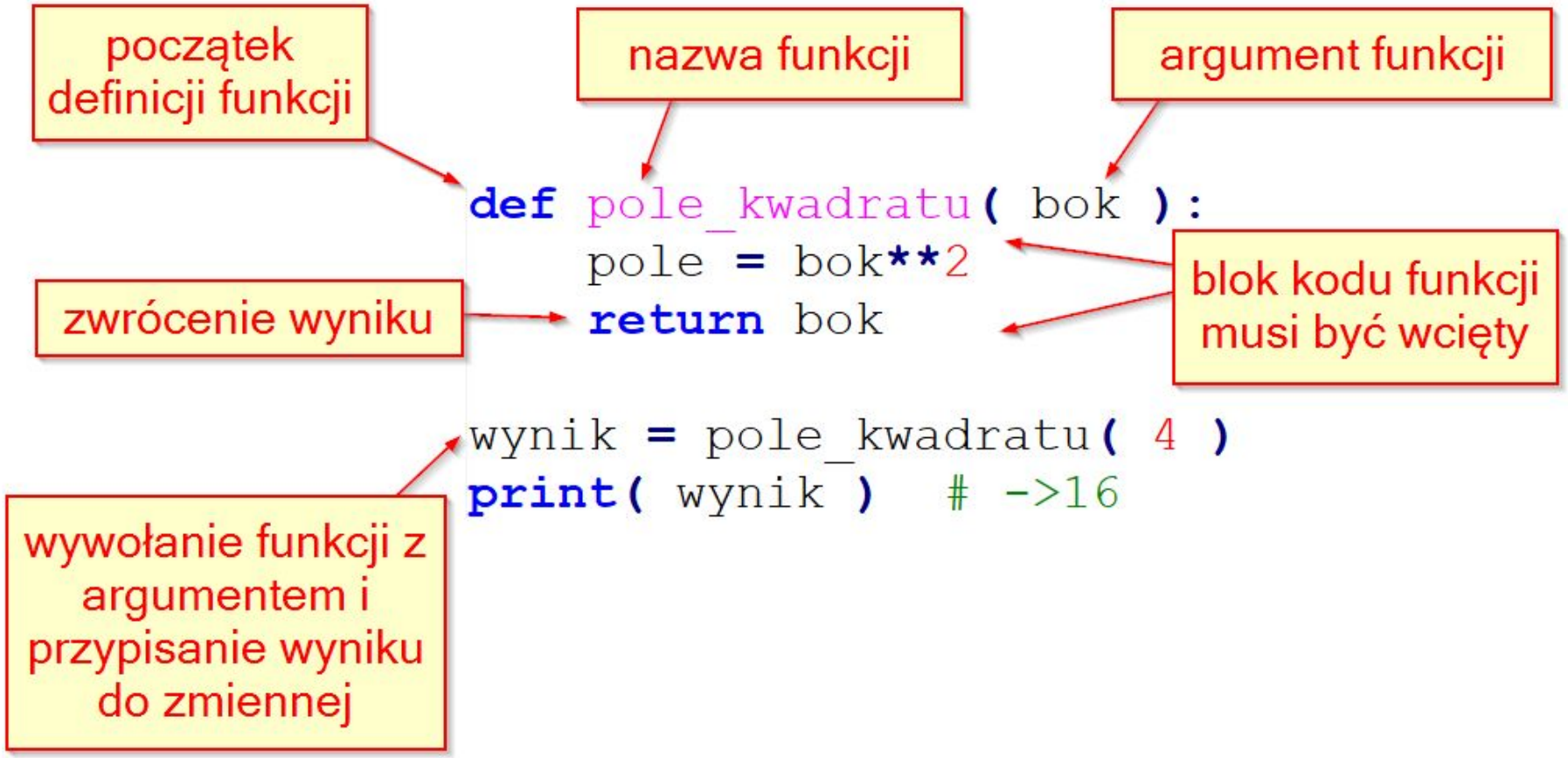
pass to tzw. *instrukcja pusta*, która nie wykonuje żadnego działania. Obiekt jest wykorzystywany jako wypełniacz linii kodu jeśli składnia wymaga podania instrukcji, ale nie jest potrzebne wykonanie żadnej czynności. Obiekt ten może być podany w dowolnym miejscu w kodzie, jednak najczęściej stosowany jest w miejscach z wcięciem, gdzie w nowej linijce musi być wcięcie, ale nie jest wymagana żadne działanie.

```
x = random.randint( 0, 10 )
if x==0:
   pass  #Dla 0 nie podejmujemy żadnych działań
elif a%2==0:
   print( x, 'liczba jest parzysta' )
else:
   print( x, 'liczba jest nieparzysta' )
```

Funkcja - wydzielony fragment kodu wykonujący operacje. Jest to jeden z podstawowych elementów programowania. Dzięki funkcjom możliwe jest podzielenie aplikacji na logiczne części, wykorzystanie tego samego kodu w wielu miejscach, zwiększenie czytelności kodu oraz łatwiejsze testowanie i usuwanie błędów.

Funkcja może przyjąć argumenty wejściowe oraz może zwracać wynik.

Budowa funkcji



return

`return` jest opcjonalne, służy zwróceniu wyniku lub wcześniejszemu przerwaniu działania funkcji. Jeśli funkcja nie ma klauzuli **`return`** to zostanie zwrócone **`None`**. Użycie samego **`return`** (bez podania wartości wyjściowej) również zwróci **`None`**.
Funkcja może posiadać wiele instrukcji **`return`**.

return

- **Funkcja nie zwraca żadnego wyniku (brak *return*)**

```
def drukuj_liste():
    print( [1, 2, 3, 4] )

drukuj_liste()
→ [1, 2, 3, 4]
```

- **Funkcja zwraca wynik, można go przypisać do zmiennej i użyć w dalszym kodzie**

```
def generuj_liste():
    return [1, 2, 3, 4]

l = generuj_liste()
print( l )
→ [1, 2, 3, 4]
```

Argumenty funkcji

Funkcja może przyjmować jeden lub wiele argumentów, które rozdzielamy przecinkami:

```
def nazwa_funkcji(argument1, argument2, …):
    <kod>
```

```
def pole_kwadratu(bok):
    return bok**2

pole = pole_kwadratu(4)
print( pole )
→ 16
```

Argumenty z wartością domyślną

Możliwe jest zdefiniowanie domyślnej wartości atrybutu. Przyjmie on podaną wartość, jeśli użytkownik nie poda tego argumentu. Argumenty tego typu zawsze muszą być podane po argumentach bez wartości domyślnej.

```
def pierwiastek(podstawa, wykladnik=2):
    return podstawa ** (1./wykladnik)

print( pierwiastek( 4 ) )       #wykladnik=2
→ 2
print( pierwiastek( 27, 3 ) ) #wykladnik=3
→ 3
```

Argumenty z wartością domyślną

```
def funkcja(a, b, c=30, d=40):
   print( a, b, c, d )

funkcja( 1, 2, 3, 4 )
→ 1, 2, 3, 4

funkcja( 1, 2, 3 )
→ 1, 2, 3, 40

funkcja( 1, 2 )
→ 1, 2, 30, 40

funkcja( 1, 2, d=4 )
→ 1, 2, 30, 4

funkcja( b=2, c=3, a=1, d=4 )
→ 1, 2, 3, 4
```

Klasy

Klasy służą do tworzenia własnych struktur danych, które mogą reprezentować dowolny obiekt.

```
#Definicja klasy
class Klasa:
   pass
#Instancja klasy
k = Klasa()
```

Klasa może posiadać przypisane do niej atrybuty oraz metody (funkcje). Metody zawsze jako pierwszy parametr przyjmują instancję danej klasy, zwyczajowo przyjmuje on nazwę `self`.

```
class Klasa:

   atrybut = 1

   def metoda(self, argument):
      print( argument )

k = Klasa()              #Tworzenie instancji
print( k.atrybut )       #Wywołanie atrybutu
k.metoda( 10 )           #Wywołanie metody
```

self

`self` jest zawsze pierwszym argumentem metod klasy. Jest on automatycznie wstawiany przez Pythona, więc przy wywoływaniu metody pomijamy go.

Obiekt `self` jest referencją do instancji danej klasy tzn. że z jego poziomu mamy dostęp do atrybutów i metod zdefiniowanych w danej klasie.

Korzystając z tego obiektu w metodach można zmieniać wartości lub dodawać nowe atrybuty.

self

```
class Klasa:
    atrybut = 1
    def metoda(self, argument):
        self.atrybut = argument

k = Klasa()
print( k.atrybut )      #zwróci 1
k.metoda( 2 )
print( k.atrybut )      #zwróci 2
```

Metoda __init__

Specjalna metoda `__init__` wywoływana jest podczas tworzenia nowej instancji klasy.

```
class Klasa:
    def __init__(self, wartosc):
        self.atrybut = wartosc

k = Klasa( 5 )          #tworzenie instancji klasy
print( k.atrybut )      #wartość została zapamiętana
w atrybucie
```

Dziedziczenie

Dziedziczenie (ang. *inheritance*) to jeden z mechanizmów programowania obiektowego dotyczący klas. W Pythonie istnieje możliwość dziedziczenia elementów (atrybutów i metod) jednej klasy (tzw. *klasa bazowa*) przez inne klasy (*klasy potomne, pochodzne*). Tworzony w ten sposób kod tworzy tzw. *hierarchię klas*.

Mechanizm ten pozwala znacząco uprościć i przyspieszyć pisanie aplikacji dzięki wykorzystaniu istniejącego już kodu.

Dziedziczenie odbywa się poprzez podanie nazwy klasy bazowej w nawiasie po nazwie definiowanej klasy.

```
class KlasaBazowa:
   atrybut1 = 1
   def metoda1(self):
      print( self.atrybut1 )

class KlasaPotomna( KlasaBazowa ):
   atrybut2 = 'tekst'
   def metoda2( self, argument ):
      print( argument )
```

Klasa bazowa nie posiada atrybutów i metod klas potomnych (dziedziczenie działa w jedną stronę), więc próba wywołania atrybutu klasy potomnej kończy się błędem.

```
dir( KlasaPotomna )
→ [..., 'atrybut1', 'metoda1']

kb = KlasaBazowa()
print( kb.atrybut1 )
→ 1
kb.metoda1()
→ 1
print( kb.atrybut2 )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'KlasaBazowa' object has no attribute
'atrybut2'
```

Dziedziczenie

Klasa potomna posiada wszystkie atrybuty i metody określone w jej definicji (atrybut2, metoda2) jak i w klasie bazowej (atrybut1, metoda1).

```
dir( KlasaPotomna )
→ [...,'atrybut1','atrybut2','metoda1','metoda2']

kp = KlasaPotomna()
print( kp.atrybut1 )
→ 1
print( kp.atrybut2 )
→ 'tekst'
kp.metoda1()
→ 1
kp.metoda2( 2 )
→ 2
```

Do obsługi plików służy funkcja open, która przyjmuje ścieżkę do pliku oraz tryb w jakim ma on być otworzony. Drugi argument jest opcjonalny, jeśli go nie podano plik zostanie otworzony w trybie tylko do odczytu. Funkcja zwraca obiekt, dzięki któremu można odczytywać lub zapisywać informacje z pliku.

```
plik = open( 'C:/plik.txt' )
```

Tryby otwierania plików

Tryby w jakich można otworzyć plik:

- **`r`** – otwarcie w trybie tylko do odczytu, domyślny - jeśli nie podano drugiego argumentu to plik zostanie otworzony w tym trybie,
- **`w`** – otworzenie z możliwością zapisywania, jeśli plik istnieje to zostanie nadpisany,
- **`a`** – otworzenie z zapisywaniem danych na końcu pliku, jeśli plik istnieje to będzie on modyfikowany, jeśli nie istnieje to zostanie utworzony,
- **`b`** – tryb binarny, stosowany łącznie w innymi trybami np. `'rb'`, `'wb'`.

Zamykanie pliku

Bardzo ważne jest poprawne zamknięcie pliku po zakończeniu operacji. Należy dodatkowo zabezpieczyć się przed wystąpieniem błędów.

```
try:
   plik = open( 'plik.txt' )
   print( plik )
finally:
   plik.close()
```

Od Python 2.5 możliwe jest zastosowanie krótszej formy:

```
with open( 'plik.txt' ) as plik:
   print( plik )
```

Po wyjściu z bloku kodu **with** plik jest automatycznie zamykany. Nastąpi to również jeśli wystąpi dowolny błąd w trakcie operowania na otwartym pliku.

Odczyt plików

Metody do odczytu danych z plików tekstowych:

- **`read()`** - odczyta całego pliku
- **`readline()`** - odczyt pojedynczej linii tekstu, do znaku końca linii (**`\n`**)
- **`readlines()`** - odczyt całego pliku, zwraca listę, w której każda linia jest osobnym elementem

Operacje na plikach
odczyt

- Odczyt całego pliku jako pojedynczy łańcuch znaków

```
with open('plik.txt') as plik:
    tekst = plik.read()
```

- Odczyt pierwszego znaku w pliku

```
with open('plik.txt') as plik:
    znak = plik.read(1)
```

- Odczyt pojedynczych wierszy

```
with open('plik.txt') as plik:
    linia1 = plik.readline()    #odczyt wiersza 1
    linia2 = plik.readline()    #odczyt wiersza 2
```

- Odczyt całego pliku w formie listy

```
with open('plik.txt') as plik:
    lista = f.readlines()
```

Metody do odczytu danych z plików tekstowych:

- **`write`**`( str )` - zapis łańcucha znaków do pliku,
- **`writelines`**`( list )` - zapis listy zawierającej łańcuchy znaków, znak nowej linii nie jest automatycznie dodawany, więc aby każdy element stanowił osobny wiersz należy dopisać na ich końcu znak nowej linii `\n`.

Operacje na plikach
zapis

```
with open(plik.txt', 'w') as plik:
    #zapisanie pierwszej linii
    plik.write('linia do zapisu\n')
    #zapisanie drugiej linii
    plik.write('druga linia')
    #zapisanie w pliku listy z tekstami
    plik.writelines( ['jeden', 'dwa'] )
```

Jeśli plik, który otwieramy istnieje to zostanie on nadpisany. Aby dopisać dane do istniejącego pliku należy otworzyć go w trybie **`'a'`** (append).

Więcej informacji

http://www.python.org/ - oficjalna strona Pythona

https://wiki.python.org - oficjalne kompendium wiedzy

http://pl.python.org/ - oficjalna strona polskiej społeczności (m.in. kursy, podręczniki)

http://www.python.rk.edu.pl - polska strona z wieloma poradnikami

https://pl.wikibooks.org/wiki/Zanurkuj_w_Pythonie - darmowy podręcznik