

Programowanie w Python

Wykład 4

dr Aneta Polewko-Klim

Wyższa Szkoła Ekonomiczna w Białymstoku

Dobre praktyki pisanie kodu - wytyczne PEP-8

- typ całkowity (*int*)
- typ zmiennoprzecinkowy (*float*)
- typ zespolony (*complex*)
- typ tekstowy (*str*)

spacje !

Uwaga zwróć uwagę że: zmienne są typowane dynamicznie

```
x = 20 #[zmienna][operator przypisania][wartość]
```

```
y = 20.0
```

```
z = 20j
```

```
print(type(x), type(y), type(z), sep=" ") # wypisz typy zmiennych, oddziel spacją  
<class 'int'> <class 'float'> <class 'complex'>
```

```
x = y
```

```
print(type(x))
```

```
<class 'float'>
```

Wcięcia w programie

- stosuj 4 spacje do wcięć w programie (Python 3 nie pozwala na mieszanie tab i spacji dla wcięć, wybrać jedną z dwóch i trzymać się jej!)
- dla większych fragmentów kodu stosuj układ kodu wertykalny

1.

```
value = square_of_numbers(num1, num2,  
                           num3, num4)
```

2.

```
def square_of_number(  
    num1, num2, num3,  
    num4):  
    return num1**2, num2**2, num3**2, num4**2
```

3.

```
value = square_of_numbers(  
    num1, num2,  
    num3, num4)
```

4.

```
list_of_people = [  
    "Rama",  
    "John",  
    "Shiva"  
]
```

5.

```
dict_of_people_ages = {  
    "ram": 25,  
    "john": 29,  
    "shiva": 26  
}
```

Długość linii kodu

- Optymalna długość linii kodu Pythona ma długość 79 znaków.

Zalety, przy utrzymaniu w/w długości:

- możliwe jest otwieranie plików obok siebie w celu porównania;
- wyświetlanie całych wyrażeń bez przewijania w poziomie, zwiększa czytelność i zrozumienie kodu.

- Komentarze powinny mieć długość linii równą 72 znakom.
- Dla właściwego podziału kodu używaj operatora +, ułatwia to zrozumienie kodu.

You should use...

```
total = (A +  
        B +  
        C)
```

You should avoid...

```
total = (A  
        + B  
        + C)
```

Puste linie

- W skryptach Pythona najwyższego poziomu funkcja oraz klasy są oddzielone dwoma pustymi liniami.
- Definicje metod wewnątrz klas powinny być oddzielone jedną pustą linią.

```
class SwapTestSuite(unittest.TestCase):
    """
    Swap Operation Test Case
    """
    def setUp(self):
        self.a = 1
        self.b = 2

    def test_swap_operations(self):
        instance = Swap(self.a, self.b)
        value1, value2 = instance.get_swap_values()
        self.assertEqual(self.a, value2)
        self.assertEqual(self.b, value1)

class OddOrEvenTestSuite(unittest.TestCase):
    """
    This is the Odd or Even Test case Suite
    """
    def setUp(self):
        self.value1 = 1
        self.value2 = 2
```

Białe znaki

- Unikaj białych znaków, gdy widzisz, że kod jest napisany tak, jak w poniższych przykładach:

You should use...

You should avoid...

```
dct['key'] = lst[index]
```

```
dct ['key'] = lst [index]
```

```
x = 1
```

```
x           = 1
```

```
y = 2
```

```
y           = 2
```

```
long_variable = 3
```

```
long_variable = 3
```

Białe znaki

- Unikaj białych znaków, gdy widzisz, że kod jest napisany tak, jak w poniższych przykładach:

You should use...

```
ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]  
ham[lower:upper], ham[lower:upper:], ham[lower::step]  
ham[lower+offset : upper+offset]  
ham[: upper_fn(x) : step_fn(x)], ham[:: step_fn(x)]  
ham[lower + offset : upper + offset]
```

You should avoid...

```
ham[lower + offset:upper + offset]  
ham[1: 9], ham[1 :9], ham[1:9 :3]  
ham[lower : : upper]  
ham[ : upper]
```

Białe znaki

- Unikaj białych znaków, gdy widzisz, że kod jest napisany tak, jak w poniższych przykładach:

You should use...	You should avoid...
<code>func(data, {pivot: 4})</code>	<code>func(data, { pivot: 4 })</code>
<code>indexes = (0,)</code>	<code>indexes = (0,)</code>
<code>if x == 4: print x, y; x, y = y, x</code>	<code>if x == 4 : print x , y ; x , y = y , x</code>
<code>spam(1)</code>	<code>spam (1)</code>

Białe znaki

You should use...	You should avoid...
<pre>i = i + 1 submitted += 1 x = x2 - 1 hypot2 = xx + yy c = (a+b) (a-b)</pre>	<pre>i=i+1 submitted +=1 x = x 2 - 1 hypot2 = x x + y y c = (a + b) (a - b)</pre>
<pre>def complex(real, imag=0.0): return magic(r=real, i=imag)</pre>	<pre>def complex(real, imag = 0.0): return magic(r = real, i = imag)</pre>

Kodowanie pliku źródłowego

- Jak wiesz komputer nie może przechowywać „liter”, „cyfr”, „obrazów” ani niczego innego, może przechowywać i pracować tylko z bitami o wartościach binarnych: tak/nie, prawda/fałsz, 1/0, itd.
- Fizycznie oznacza to że w komputerze w jakimś fragmencie obwodu płynie/niepłynie prąd elektryczny, obszar jest namagnesowany/nienamagnesowany itd.
- Zatem aby użyć bitów do reprezentowania czegokolwiek potrzebujemy zestawu reguł: musimy przekonwertować sekwencję bitów na coś w rodzaju liter, cyfr i obrazów, używając wybranego schematu kodowania np. ASCII, UTF-8 itp.:

ASCII: amerykański standardowy kod wymiany informacji (ASCII) jest najpowszechniejszym formatem plików tekstowych w komputerach i Internecie. W tego typu plikach każdy znak alfabetyczny, numeryczny lub specjalny jest reprezentowany przez 7-bitową liczbę dwójkową.

W Python 3: UTF-8 jest domyślnym kodowaniem źródłowym.

W Python 2: domyślnie jest ASCII.

Aby sobie z tym poradzić, możesz skorzystać z metod łańcuchowych:

`.encode ()` i `.decode ()`.

Importowanie modułów

You should use...	You should avoid...
<pre>from config import settings or import os import sys</pre>	<pre>import os, sys</pre>

Preferowane jest o ile to możliwe importowanie absolutne/całościowe modułu

Komentarze

Komentarze są używane do dokumentacji w kodzie w Pythonie, korzystamy z #, komentarz poprzedza wyjaśniane linie kodu.

Wariant1:

```
sel = 0 # initialize the counter --> # + spacja + Twój komentarz
```

Wariant2:

```
if Gram is None or Gram is False:
    Gram = None
    if copy_X:
        # force copy. setting the array to be fortran-ordered
        # speeds up the calculation of the (partial) Gram matrix
        # and allows to easily swap columns
        X = X.copy('F')
```

W dokumentacji na początku publicznych modułów, plików, klas i metod wstaw komentarz:

```
"""
    Algos module consists of all the basic algorithms and their implementation
"""
import os
```

Konwencja nazw

Identifier	Convention
Module or Package	lowercase e.g. pandas
Class	CapWords e.g. class OddOrEvenTestSuite():
Functions	lowercase eg. print()
Methods	lowercase e.g. *.upper
Type variables	CapWords e.g. String
Constants	UPPERCASE e.g. A = 10

Obsługa folderów

```
import os # tworzenie katalogów i podgląd ich zawartości itp
print(os.getcwd()) # podgląd bieżącej ścieżki, do folderu roboczego
C:/Users/aneta/Lab8.py

print(os.listdir('.')) # podgląd zawartości bieżącego folderu
['anaconda', 'astro.jpg', 'plik.doc']
print(os.listdir('c:\\Windows')) # zawartość folderu Windows na dysku c

os.chdir('d:\\Filmy') # zmiana bieżącego folderu na folder Filmy, dysk d
print(os.getcwd())
D:/Filmy

rozmiar_pliku_bajtach = os.path.getsize('E:\\DYDAKTYKA\\Programowanie')
print(rozmiar_pliku_bajtach)
32768
```

Zmiana nazwy folderu: metoda `os.rename(stara_nazwa, nowa_nazwa)`
`os.rename('D:\\DYDAKTYK\\Dokument3', 'D:\\DYDAKTYK\\Dokument4')`

Obsługa folderów: szukanie plików o określonych nazwach

Biblioteka do obsługi zmiennych typu string, metoda `fnmatch` sprawdza czy podana zmienna typu string zaczyna i/lub kończy się na określoną literę

```
import fnmatch
```

```
zdanie = 'Python to język programowania'
```

```
print(fnmatch.fnmatch(zdanie, 'P*a')) #czy zmienna zdanie zaczyna się  
na literę P i kończy na a
```

```
True
```

```
print(fnmatch.fnmatch(zdanie, '*m')) # czy zmienna zdanie kończy na m
```

```
False
```

```
print(fnmatch.fnmatch(zdanie, 'P*')) # czy zmienna zdanie zaczyna się  
na literę P
```

```
True
```

Obsługa folderów: operacje na ścieżkach

```
import os
# path.join() łączy ciąg katalog/plik w ścieżkę
print(os.path.join('C:\Users\aneta\','Labo1.py'))
>> C:\Users\aneta\Labo1.py
# path.realpath() wskazuje ścieżkę do pliku Labo1.py
print(os.path.realpath('Labo1.py'))
>> C:\Users\aneta\Labo1.py
# path.split() rozdziela ścieżkę i nazwę pliku (tworzy krotkę)
sciezka_plik = os.path.split(sciezka_do_pliku)
print(sciezka_plik)
>>('C:\\Users\\aneta', 'Labo1.py')
print(sciezka_plik[0])
>> C:\Users\aneta
print(sciezka_plik[1])
>> Labo1.py
```


Obsługa folderów: tworzenie i kasowanie

```
print('yes') if 'Nowy_Folder' in os.listdir() else print('no')
```

```
>> no
```

```
os.mkdir(my_folder) # stwórz katalog 'Nowy_Folder'
```

```
os.rmdir(my_folder) # usuń katalog 'Nowy_Folder'
```

Drzewo katalogów

```
os.makedirs("Pierwszy/Drugi/Trzeci") # utwórz "drzewo" katalogów
```

```
os.listdir("Pierwszy")
```

```
['Drugi']
```

```
os.listdir("Pierwszy/Drugi")
```

```
['Trzeci']
```

```
os.removedirs("level0/level1/level2") # usuń wszystkie katalogi
```

Zapis/odczyt plików

Opening and closing files in Python

- `open()` creates a file object, which would be utilized to call other support methods associated with it

Syntax: `file_object = open(file_name, access_mode)`

`access_mode` - determines the mode in which the file has to be opened

- `write()` write data to file
- `close()` closes the opened file. A closed file cannot be read or written any more

Opening and closing files in Python

<code>access_mode</code>	
<code>'r'</code>	Opens a file for reading only
<code>'w'</code>	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
<code>'a'</code>	Opens a file, appends data. If the file does not exist, creates a new file for writing.

Other value of `access_mode` parameter: [link](#)

Opening and Closing Files in Python

Example 1:

open a file for writing only, write data and close file

```
file1 = open('FirstFile.txt', 'w', encoding='utf-8')
```

```
text = 'I love python'
```

```
file1.write(text) # write data to file
```

```
file1.close() # close file
```

Opening and Closing Files in Python

Example 2:

open a file for writing only in binary format, write data and close file

```
file = open('FirstFile.txt', 'wb')  
text = 'I love python'  
file.write(text)  # write data to file  
file.close()     # close file
```

Opening and Closing Files in Python

Example 3:

open a file for appending, append data to a file as a new line
and close file

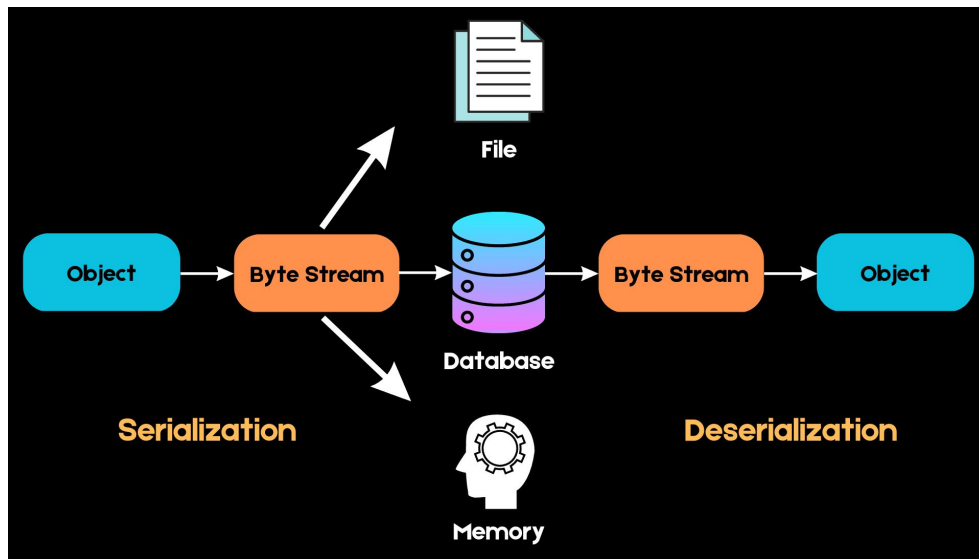
```
file = open('FirstFile.txt', 'a')  
text = 'I love python'  
file.write(text) # write data to file  
file.close()    # close file
```

Serializing and de-serializing a Python object structure - pickle package

- W pythonie możemy wykonać serializację obiektów. Serializacja obiektu to proces konwersji obiektu na strumień bajtów, który można zapisać a później przywrócić (j.ang. pickling)
- Moduł pickle zawiera funkcje do serializacji obiektów
- Proces serializacji obiektów w Python wykorzystujemy do szybkiego zapisu i odczytu informacji.

Kroki procesu serializacji obiektu:

- otwórz plik w trybie zapisu binarnego
- wywołaj funkcję dump() z modułu pickle i zapisz obiekt do pliku
- zamknij plik



Write data using pickle

Example 1:

```
import pickle
filename = 'shopping_list.pkl'
products = ['potatoes', 'onions', 'tomatoes']
f1 = open(filename, 'wb') # with open('shopping_list.pkl', 'wb') as f1:
pickle.dump(products, f1)
f1.close()
```


Write data using pickle - multiple variables

Example 3:

```
import pickle
f1 = open('shopping_list.data' , 'wb')
product1 = 'potatoes'
product2 = 'onions'
pickle.dump((product1,product2) , f1)
f1.close()
```

Note: data will be save at the end of the program if you don't use .close()

```
f2 = open(filename, 'rb')
pickle.load(some_prod, f2)

print(some_prod[0])
>> 'potatoes'
```

Write data in binary format - funkcje pack() and unpack() z pakietu struct

Example1:

```
from struct import pack, unpack

f1 = open('plik.dat', 'wb')
numbers = pack('hhl', 50, 100, 150)
f1.write(numbers)
f1.close()
```

Example2:

```
from struct import pack, unpack

f2 = open('plik.dat', 'wb')
numbers = unpack('hhl', f2.read())
...
f1.close()
```

Struct module can be used in handling binary data stored in files, database or from network connections. Python values and C structs represented as Python bytes objects.

Syntax:

pack(format, var1, var2, ...)

?: boolean

h: short int

l: long int

i: int

f: float

q: long long int

Obsługa błędów i wyjątków

Typy błędów:

- błędy leksykalne
- błędy typowania
- błędy działania
- błędy semantyczne
- błędy składniowe
- błędy logiczne
- nieskończone obliczenia

np. niepoprawnie zapisane działanie
 $a+b/c$ powinno być $(a+b)/c$

np. niepoprawnie wpisana formuła
 $\text{delta} = b - 4*a*c$
powinno być $b*b-4*a*c$

np. wpisywanie niekończącej się pętli

TE NAJTRUDNIEJ WYKRYĆ BO PROGRAM NIE WYRZUCA BŁĘDÓW PODCZAS DZIAŁANIA.....TE ELIMINUJEMY TESTAMI OPROGRAMOWANIA

Skutki błędów oprogramowania:

Therac-25

Między 1985 a 1987 rokiem 6 osób uległo poparzeniu w wyniku naświetlań maszyną Therac-25. Trzy z nich zmarły w następstwie wypadku. W trakcie pierwszego wypadku, w wyniku którego pacjentka straciła pierś i czucie w ręce, okazało się, że automat zaaplikował ok. 100 razy większą dawkę promieniowania, niż wynikało ze zlecenia. Producent, firma AECL uznała jednak, że to niemożliwe, nie podjęto więc żadnych działań. Jeszcze w tym samym roku – w 1985 r. – inna maszyna uległa awarii, wyświetlając komunikat o błędzie i niepodjęciu naświetlania. Operator, przyzwyczajony do humorów urządzenia, wymusił wykonanie procedury. Maszyna pięciokrotnie podejmowała próbę wykonania naświetlenia, po czym zupełnie odmówiła posłuszeństwa. 3 miesiące później pacjent, który brał udział w zabiegu, zmarł w związku z powikłaniami napromieniowania.



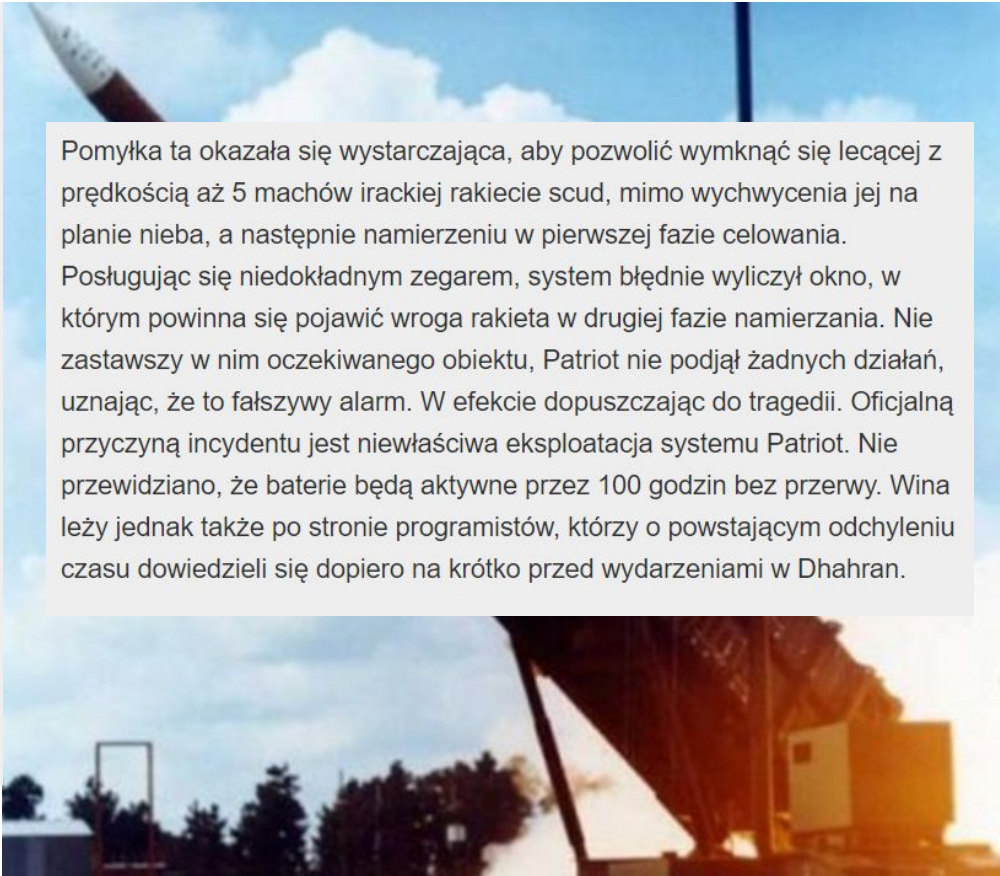
AECL bardzo długo wypierało się winy, uznając, że nie istnieje możliwość, aby Therac-25 mylił dawki albo wykonał naświetlania mimo przeczącego temu komunikatowi. Mimo to poparzeniom uległo jeszcze kilka osób, a sprawa trafiła do sądu. W toku postępowania, rzecznik AECL przyznał, że wykonano „małą liczbę” testów urządzenia nim trafiło na rynek. Jak się okazało, wart ponad 1 mln dolarów automat został wyposażony w napisane w assemblerze oprogramowanie, stworzone przez jedną osobę. Wypadki powodowały dwa drobne przeoczenia programisty. Ogółem jednak, zabrakło jednego, jak się okazało, niezmiernie istotnego wiersza kodu. Raptem kilkudziesięciu znaków. Z drugiej strony, błąd z dużym prawdopodobieństwem zostałby wychwycony przed wprowadzeniem produktu na rynek, gdyby nie zabrakło rzetelniejszej procedury testowej.

Skutki błędów oprogramowania:

Patriot

Podczas Wojny w Zatoce Perskiej w 1991 roku iracka rakietą scud trafiła w amerykańskie baraki w Dhahran, zabijając 28 osób i raniąc ponad 100. Doszło do tragedii, mimo że bezpieczeństwa bazy pilnowało aż 6 baterii systemu przeciwrakietowego Patriot. Wyjaśniając kulisy wypadku należy podkreślić, że Patriot stworzono w latach 70., w celu zwalczania radzieckich rakiet, poruszających się przeciętnie z prędkością około 2 machów. Dodatkowo, w założeniach, system miał być wysoce mobilny. Czas ciągłej pracy przewidywano więc na nie więcej niż 8 godzin. Po tym okresie miał on być dezaktywowany, przewożony i uruchomiany ponownie w nowym miejscu.

Jak się okazało, w związku z długim procesem aktywowania się systemu (60-90 sekund), amerykańskie baterie w Iraku działały bez przerwy nawet ponad 100 godzin. Nie byłoby to problemem, gdyby nie fakt, że system wykorzystywał w procesie namierzania pocisków wroga własny czas pracy w sekundach. Niestety, z pewnych względów, dokładność obliczeń zmiennoprzecinkowych, wykonywana w tym celu, była daleka od ideału. W efekcie, 1 sekunda wyliczana przez baterię nie była równa rzeczywistej sekundzie. Po 100 godzinach pracy, system mylił się już o 0,34 sekundy.



Pomyłka ta okazała się wystarczająca, aby pozwolić wymknąć się lecącej z prędkością aż 5 machów irackiej rakiecie scud, mimo wychwycenia jej na planie nieba, a następnie namierzeniu w pierwszej fazie celowania. Posługując się niedokładnym zegarem, system błędnie wyliczył okno, w którym powinna się pojawić wroga rakietą w drugiej fazie namierzania. Nie zastawszy w nim oczekiwanego obiektu, Patriot nie podjął żadnych działań, uznając, że to fałszywy alarm. W efekcie dopuszczając do tragedii. Oficjalną przyczyną incydentu jest niewłaściwa eksploatacja systemu Patriot. Nie przewidziano, że baterie będą aktywne przez 100 godzin bez przerwy. Wina leży jednak także po stronie programistów, którzy o powstającym odchyleniu czasu dowiedzieli się dopiero na krótko przed wydarzeniami w Dhahran.

3 podstawowe zasady zapobiegania błędom:

1. **Pisz czytelny kod:**

- nazwy zmiennych powiązane z ich znaczeniem;
- komentuj kod źródłowy, który nie jest dla Ciebie zrozumiały od razu;
- twórz dokumentację w trakcie pisania programu
- opisuj przyjęte przez Ciebie założenia (w komentarzach i/lub dokumentacji)

2. **Code review**

przełącz kod innemu programiście do przeglądu przed włączeniem kodu do systemu kontroli wersji i przekazaniem go do testowania

3. **Wykorzystaj wbudowany debugger**

Skontroluj kod pod nadzorem debuggera, programu komputerowego służącego do dynamicznej analizy innych programów, w celu odnalezienia i identyfikacji zawartych w nich błędów

Obsługa wyjątków: `SyntaxError`

- błędy leksykalne (pojedyncza jednostka leksykalna, której nie przewiduje definicja języka)

```
x = 10
```

```
x++ # w python nie istnieje operator ++
```

```
File "C:/Users/aneta/Obsluga bledow.py", line 2
```

```
x++ # w python nie istnieje operator ++
```

```
SyntaxError: invalid syntax
```

-

- błędy składniowe (jednostki leksykalne poprawne ale niewłaściwie zestawione)

```
for i in range(0,10)    # brakuje :
```

```
print(i)
```

```
File "C:/Users/aneta/Obsluga bledow.py", line 1
```

```
for i in range(0,10)
```

```
^
```

```
SyntaxError: invalid syntax
```

Obsługa wyjątków: **TypeError**

- **błędy typowania** (wyrażenie nieadekwatne do typu)

```
x = 10
```

```
x[0] = 20
```

Traceback (most recent call last):

File "C:/Users/aneta/Obsluga bledow.py", line 2, in <module>

x[0] = 20

TypeError: 'int' object does not support item assignment

- **błędy działania** (*runtime error*)

```
print(10/0)
```

Traceback (most recent call last):

File "C:/Users/aneta/Obsluga bledow.py", line 1, in <module>

print(10/0)

ZeroDivisionError: division by zero

Obsługa wyjątków: hierarchia

BaseException

```
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
        |   +-- FloatingPointError
        |   +-- OverflowError
        |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
        |   +-- ModuleNotFoundError
    +-- LookupError
        |   +-- IndexError
        |   +-- KeyError
    +-- MemoryError
    +-- NameError
        |   +-- UnboundLocalError
    +-- OSError
        |   +-- BlockingIOError
        |   +-- ChildProcessError
        |   +-- ConnectionError
        |       |   +-- BrokenPipeError
        |       |   +-- ConnectionAbortedError
        |       |   +-- ConnectionRefusedError
        |       |   +-- ConnectionResetError
        |   +-- FileExistsError
        |   +-- FileNotFoundError
```

cd.:

```
+-- FileExistsError
+-- FileNotFoundError
+-- InterruptedError
+-- IsADirectoryError
+-- NotADirectoryError
+-- PermissionError
+-- ProcessLookupError
+-- TimeoutError
+-- ReferenceError
+-- RuntimeError
    +-- NotImplementedError
    +-- RecursionError
+-- SyntaxError
    +-- IndentationError
    +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
    +-- UnicodeError
        +-- UnicodeDecodeError
        +-- UnicodeEncodeError
        +-- UnicodeTranslateError
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
```

Typ wyjątku	Opis
IOError	Zgłaszany w przypadku wystąpienia błędu operacji wejścia-wyjścia, takiego jak próba otwarcia nieistniejącego pliku w trybie odczytu.
IndexError	Zgłaszany przy indeksowaniu sekwencji, gdy numer indeksu wskazuje na nieistniejący element.
KeyError	Zgłaszany, gdy nie zostanie znaleziony klucz słownika.
NameError	Zgłaszany, gdy nie zostanie znaleziona nazwa (na przykład zmiennej lub funkcji).
SyntaxError	Zgłaszany, gdy zostanie wykryty błąd składni.
TypeError	Zgłaszany, gdy wbudowana operacja lub funkcja zostanie zastosowana do obiektu nieodpowiedniego typu.
ValueError	Zgłaszany, gdy wbudowana operacja lub funkcja otrzyma argument, który ma właściwy typ, ale nieodpowiednią wartość.
ZeroDivisionError	Zgłaszany, gdy drugi argument operacji dzielenia lub modulo jest równy zero.

Obsługa wyjątków:

```
1 while True:
2     x = int(input("Podaj liczbę różną od zera x =: "))
3     if x == 0:
4         print("Nie wpisałeś liczby")
5         continue
6     else:
7         print('Wartość 10/x = ', 10 / x)
8         break
```

ZeroDivisionError

Podaj liczbę różną od zera x =: a

Traceback (most recent call last):

File "C:/Users/aneta/Obsługa bledow.py", line 2, in <module>

x = int(input("Podaj liczbę różną od zera x =: "))

ValueError: invalid literal for int() with base 10: 'a'

Obsługa wyjątków

“złap błąd”
użyj klauzuli: **try**

SCHEMAT “polowania” na potencjalne BŁĘDY

try:

doSomething() # wykonaj właściwy kod programu

except SomeException:

print("Something went wrong") #wyświetl komunikat

```
1 while True:
2     try:
3         x = int(input("Podaj liczbę różną od zera x =: "))
4         print('Wartość 10/x = ', 10/x)
5         break
6     except ValueError: # jeśli błąd wartości
7         print("Nie wpisałeś liczby")
8
```

Podaj liczbę różną od zera x =: 5
Wartość 10/x = 2.0

Podaj liczbę różną od zera x =: a
Wpisz literę alfabetu
Podaj liczbę różną od zera x =:

Obsługa wyjątków:

```
1  while True:
2      try:
3          x = int(input("Podaj liczbę różną od zera: "))
4          if x == 0:
5              print("Wpisałeś liczbę zero.")
6              continue
7          else:
8              print('Wartość 10/x = ', 10 / x)
9              break
10         break
11     except ValueError:
12         print("To nie jest cyfra")
```

Czy to wystarczy? Czy to efektywne ?

Obsługa wyjątków:

TRZYMAJ SIĘ ZASAD:

1. ŁAP WSZYSTKIE MOŻLIWE BŁĘDY RÓWNOCZEŚNIE
2. UŻYWAJ WBUDOWANYCH KOMUNIKATÓW

```
try:
    1/0
except ZeroDivisionError as e:
    print("ZeroDivisionError Exception:", e)
except TypeError as e:
    print("TypeError exception:", e)
```

ZeroDivisionError Exception: division by zero

e - komunikat wbudowany

```
try:
    1/'a'
except ZeroDivisionError as e:
    print("ZeroDivisionError Exception:", e)
except TypeError as e:
    print("TypeError exception:", e)
```

TypeError exception: unsupported operand type(s) for /: 'int' and 'str'

Obsługa wyjątków:

TRZYMAJ SIĘ ZASAD:

1. ŁAP WSZYSTKIE MOŻLIWE BŁĘDY RÓWNOCZEŚNIE
2. UŻYWAJ WBUDOWANYCH KOMUNIKATÓW

```
1 def fun_error():
2     try:
3         x = int("A")
4     except ValueError as e:
5         print("Exception in function f:", e)
6         raise
7
8
9 try:
10     fun_error()
11 except ValueError as e:
12     print("Exception:", e)
13
14 print("It works :) ")
```

raise - instrukcja powoduje ponowne wygenerowanie ostatniego wyjątku, który był aktywny w bieżącym zasięgu. Jeśli w bieżącym zasięgu nie był aktywny żaden wyjątek, generowany jest wyjątek sygnalizujący ten błąd.

Klauzula **raise** służy do samodzielnego wywołania błędu **jeżeli zajdzie jakaś sytuacja w kodzie, która zająć nie powinna a o której chcemy poinformować użytkownika**

Zgłoszenie wyjątku: raise

Jako programista możesz zgłosić wyjątek na podstawie warunków. Użyj „raise”, aby zgłosić wyjątek w Pythonie, jeśli wystąpi określony warunek.

```
1 x=5
2 if x>10:
3     print(x)
4 else:
5     raise Exception("Negative numbers are not allowed")
```

else

Wykład_program x

C:\Users\aneta\PycharmProjects\python2022\venv\Scripts\python.exe C:/Users/aneta/PycharmPro

Traceback (most recent call last):

File "C:/Users/aneta/PycharmProjects/python2022/Wyklad_program.py", line 5, in <module>

raise Exception("Negative numbers are not allowed")

Exception: Negative numbers are not allowed

Process finished with exit code 1

Obsługa wyjątków:

klauzula: **else**

w przeciwnym przypadku
wykonaj;

można po niej umieścić kod, który
zostanie wykonany, jeżeli nie
zostanie zgłoszony wyjątek

Ale co się dzieje w tym przypadku?

```
a = (division_by2('a', 0))  
print(a)
```

```
1  def division_by2(x, y):  
2      try:  
3          result = x / y  
4          return result  
5      except ZeroDivisionError:  
6          print("ZeroDivisionError")  
7      else:  
8          result = 'Nan'  
9          return result  
10  
11  a = (division_by2(10, 0))  
12  print(a)  
13  
14  ZeroDivisionError  
15  None
```

Traceback (most recent call last):

File "C:/Users/aneta/Obsługa błędów.py", line 12, in <module>

a = (division_by2('a', 0))

File "C:/Users/aneta/Obsługa błędów.py", line 3, in division_by2

result = x / y

TypeError: unsupported operand type(s) for /: 'str' and 'int'

Obsługa wyjątków:

klauzula: **finally**
niezależnie od typu każdego
błędu podstaw

```
def division_by2(x, y):  
    try:  
        result = x / y  
        return result  
    finally:  
        result = 1  
        return result
```

```
a = (division_by2('a', 0))  
print(a)
```




1

Wyjątek FileNotFoundError

```
try:  
    with open('file.txt') as file:  
        read_data = file.read()  
except FileNotFoundError as e:  
    print(e)
```

analogia



```
file = open('file.txt', 'w')
```

Podsumowanie

Zapamiętaj:

- Używaj słowa kluczowego **raise**, aby w dowolnym momencie zgłosić wyjątek (informowanie użytkownika o sytuacji w kodzie, która zajść nie powinna).
- Używaj słowa kluczowego **assert** aby sprawdzić słowa kluczowe pod kątem określonego warunku. Jeśli warunek jest fałszywy, zgłasza wyjątek.
- Kod wewnątrz bloku **try** jest wykonywany, dopóki nie napotka wyjątku
- Kod wewnątrz bloku **else** jest wykonywany tylko wtedy, gdy nie ma wyjątku.
- Kod wewnątrz bloku **finally** wykonywany zawsze niezależnie od wyjątku