

Obsługa błędów

Mamy 3 rodzaje błędów:

1. Błędy składniowe – powstają gdy piszemy program niezgodnie z gramatyką języka np. błędy w definicjach funkcji, niezamknięte nawiasy czy cudzysłów bez pary.
2. Błędy czasu wykonania – powodują przerwanie lub niewłaściwe działanie. Mamy możliwość ich przechwycenia i wymuszenia odpowiedniej reakcji np. użytkownik podaje litery a miał wpisywać liczby.
3. Błędy logiczne – błędy w algorytmie lub programie. Nie wykrywalne przez interpreter ale możliwe przez człowieka po analizie programu

Obsługę błędów realizujemy przez specjalny blok instrukcji.

Składnia:

try:

 instrukcje

except nazwa_bledu_1:

 awaryjne_instrukcje_1

...

[except nazwa_bledu_n:

 awaryjne_instrukcje_n]

[else:

 blok_bez_bledu]

Przykład

Podajemy dwie liczby do dzielenia, chcemy wyłapać dzielenie przez 0

```
print("Proszę podać pierwszą liczbę")
```

```
licz1 = input()
```

```
print("Proszę podać drugą liczbę")
```

```
licz2 = input()
```

```
try:
```

```
    wynik = int(licz1) / int(licz2)
```

```
    print("Wynik= " + str(wynik))
```

```
except ZeroDivisionError: #nazwa błędu dzielenia przez zero
```

```
    print("Tylko Chuck Norris może dzielić przez zero!")
```

Python Comprehension

Jest to mechanizm służący do generowania kolekcji (lista, słownik, zbiór) na podstawie jednowierszowej definicji. Równoważne definicje zawsze można podać za pomocą pętli. Czasami zaś wystarczy przepisać na język Python definicję matematyczną zbioru.

Możliwa składnia

#Zamiast pisać w pętli

lista = []

for element in zakres:

 if pewien_warunek_na(element):

 lista.append(„Cos sie dzieje z:” + element)

#możemy zapisać w jednej linijce

lista = [„Cos sie dzieje z:” + element for element in zakres if pewien_warunek_na(element)]

Przykład pierwszy

$A = \{x^2: x \in \langle 0, 9 \rangle\}$

$B = \{1, 3, 9, 27, \dots, 3^5\}$

$C = \{x: x \in A \text{ i } x \text{ jest liczbą nieparzystą}\}$

W pythonie zapiszemy to:

#wersja z pętlą

a = []

for x *in* range(10):

a.append(x**2)

print(*a*)

b = []

for y *in* range(6):

b.append(3**y)

print(*b*)

c = []

for z *in* *a*:

if z % 2 == 1:

c.append(z)

```
print(c)

#wersja z python comprehension
a = [x**2 for x in range(10)]
b = [3**i for i in range(6)]
c = [x for x in a if x % 2 == 1]

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

Przykład drugi

Chcemy uzyskać liczby parzyste z podanego zakresu

```
#wersja z pętlą
liczby = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
lista = []

for i in liczby:
    if i % 2 == 0:
        lista.append(i)

print("Liczby parzyste uzyskane z wykorzystaniem pętli")
print(lista)

print()
```

```
#wersja z python comprehension
lista2 = [i for i in liczby if i % 2 == 0]
print(lista2)
```

Przykład trzeci zagnieżdżenia

```
#wersja z zagnieżdżonymi pętlami
lista = []

for i in [1, 2, 3]:
    for j in [4, 5, 6]:
        lista.append((i,j))
```

```
print(lista)
```

#wersja z python comprehension

```
lista2 = [(i,j) for i in [1, 2, 3] for j in [4, 5, 6]]
```

```
print(lista2)
```

Przykład czwarty związany ze zamianą klucza z wartością w słowniku

#wersja z pętlą

```
skroty = {"PZU": "Państwowy zakład ubezpieczeń",  
          "ZUS": "Zakład ubezpieczeń społecznych",  
          "PKO": "Państwowa kasa oszczędności"}
```

```
odwrocone = {}
```

```
for key,value in skroty.items():
```

```
    odwrocone[value] = key
```

```
print(odwrocone)
```

#wersja z python comprehension

```
odwrocone2 = {value: key for key, value in skroty.items()}
```

```
print(odwrocone2)
```

Funkcje

W pythonie możemy definiować własne funkcje, które będziemy traktować jak podprogramy lub jak funkcje w matlabie.

Składnia

```
def nazwa_funkcji(arg_pozycyjny, arg_domyslny=wartosc, *arg_4, **arg_5):  
    instrukcje  
    return wartość
```

Definicja instrukcji to instrukcja która tworzy obiekt. Funkcje możemy wywoływać z argumentami lub bez ale zawsze musimy używać nawiasów (nawet jak nie ma argumentów). Funkcja może zwracać jedną lub wiele wartości, które będą zwrócone jako krotka

Przykład pierwszy

Chcemy zdefiniować funkcję, która będzie obliczać pierwiastki równania kwadratowego

```
def row_kwadratowe(a,b,c):
```

```
    delta = b**2 - 4 * a * c
```

```
    if delta < 0:
```

```
        print("brak pierwiastków")
```

```
        return -1
```

```
    elif delta == 0:
```

```
        print("jedne pierwiastek")
```

```
        x = (-b) / (2 * a)
```

```
        return x
```

```
    else:
```

```
        print("dwa pierwiastki")
```

```
        x1 = (-b - math.sqrt(delta)) / (2 * a)
```

```
        x2 = (-b + math.sqrt(delta)) / (2 * a)
```

```
        return x1,x2
```

```
print(row_kwadratowe(6,1,3))
```

```
print(row_kwadratowe(1,2,1))
```

```
print(row_kwadratowe(1,4,1))
```

Przykład drugi

Definiujemy funkcję z wartościami domyślnymi

```
import math
```

```
def dlugosc_odcinka(x1 = 0, y1 = 0, x2 = 0, y2 = 0):
```

```
    return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
```

```
#wywołujemy funkcje dla wartości domyślnych
```

```
print(dlugosc_odcinka())
```

#wywołujemy funkcje dla własnych podanych wartości

#są to argumenty pozycyjne czyli ważna jest kolejność podania wartości

```
print(dlugosc_odcinka(1,2,3,4))
```

#wywołujemy funkcje podając mieszane wartości

#dwie pierwsze interpretowane są jako x1 i y1 jak podano w definicji funkcji

```
print(dlugosc_odcinka(2, 2, y2=2, x2=1))
```

#wywołujemy funkcje podając wartości nie w kolejności

```
print(dlugosc_odcinka(y2=5, x1=2, y1=2, x2=6))
```

#wywołujemy funkcje podając tylko dwa argumenty a reszta domyślne

```
print(dlugosc_odcinka(x2=5, y2=5))
```

Przykład trzeci

Symbol * oznacza dowolną ilość argumentów przechowywanych w krotce

```
def ciag(* liczby):
```

```
    # jeżeli nie ma argumentów to
```

```
    if len(liczby) == 0:
```

```
        return 0
```

```
    else:
```

```
        suma = 0
```

```
        #sumujemy elementy ciągu
```

```
        for i in liczby:
```

```
            suma += i
```

```
        #zwracamy wartość sumy
```

```
        return suma
```

#wywołanie gdy nie ma argumentów

print(ciaag())

#podajemy argumenty

print(ciaag(1, 2, 3.5, 4, 5, 6, 7, 8))

Przykład czwarty

****** dwie gwiazdki oznaczają że możemy użyć dowolną ilość argumentów z kluczem

*def to_lubie(** rzeczy):*

for cos in rzeczy:

print("To jest ")

print(cos)

print(" co lubie ")

print(rzeczy[cos])

to_lubie(slodycze="czekolada", rozrywka=['gry', 'filmy'])