Faculty of Geodesy
and Cartography
WARSAW UNIVERSITY OF TECHNOLOGY

Informatyka Geodezyjna - wykłady/ćwiczenia, rok akad. 2021-2022

Wyk. 10: Python – błędy i wyjątki

Kinga Węzka
kinga.wezka@pw.edu.pl
Katedra Geodezji i Astronomii Geodezyjnej

Warsaw University
of Technology

1. Błędy i wyjątki w Pythonie

2. Obsługa błędów i wyjątków w Pythonie
   - Obsługa wyjątków – try/except
   - Obsługa wyjątków – try/exept/else
   - Obsługa wyjątków – try/exept/else/finally
   - Instrukcja raise – zgłaszanie wyjątku
     - Raise versus print
     - Instrukcja assert i AssertionError
     - Po co są wyjątki?

3. Podsumowanie
   - Instrukcje obsługujące błędy i wyjątki – podsumowanie
   - Lista wbudowanych błędów i wyjątków w Python 3.x
   - Hierarchia wbudowanych błędów i wyjątków w Python 3.x
   - Polecane strony internetowe

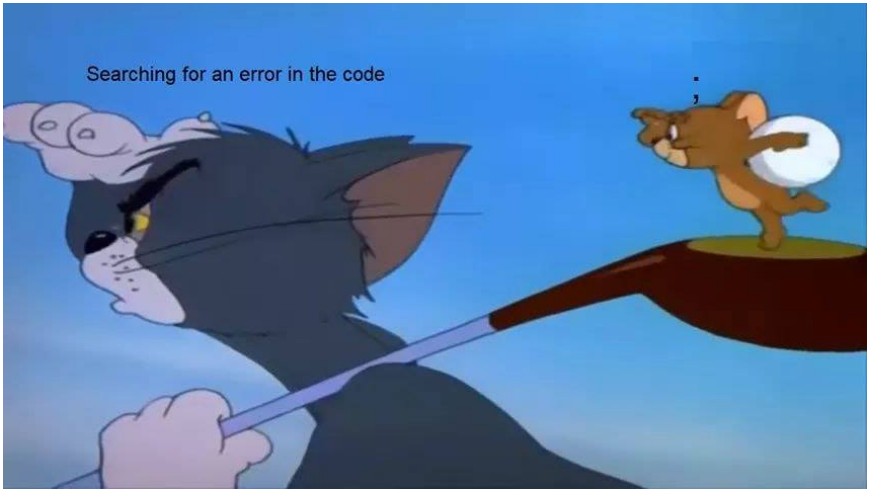


Figure: Indentyfikacja błedów

- **Błędy składni** [1] (ang. *Syntax errors*), zwane również *błędami parsingu*
  - Błędy *gramatyczne* i *ortograficzne* języka – program wykona się tylko bez tych błędów. Spowodowane są nieprawidłową składnią (np. indentacja) – to inaczej błędy **parsowania**[2], w efekcie otrzymujemy komunikat z numerem linii z błędem oraz jej treścią (strzałka pokazuje token przed którym wykryto błąd).
  - Błędy składni są sygnalizowane na etapie kompilacji kodu źródłowego.
  - Python nie sprawdza statycznie zgodności typów, niezgodności powodują błędy działania.
- **Błędy działania – Wyjatki** (ang. *Runtime errors – Exceptions*)
  - Pojawiają się na etapie wykonywania programu, **nawet gdy wyrażenie jest składniowo poprawne, może spowodować błąd podczas wykonania go** (np. brak pliku do otwarcia).
  - Tego typu błędy są nazywane **Wyjątkami** (ang. *Exceptions*) i istnieje specjalny mechanizm ich obsługi (**try/except**), **prawidłowo obsłużone nie kończa działania programu**.
- **Błędy semantyczne** (ang. *Semantic errors*)
  - Program działa i nie zgłasza błędów, ale wynik jego działania jest niepoprawny – to błąd semantyczny. Programista miał co innego na myśli niż faktycznie wykonuje program.

[1] Więcej na ten temat w (Lutz, 2011, rozdział 33, p. 851) oraz (Downey, 2016, p.36)

[2] Analizator składniowy lub parser – program dokonujący analizy składniowej danych wejściowych w celu określenia ich struktury gramatycznej w związku z określoną gramatyką formalną.

- Błędy składniowe występują, gdy analizator składni wykryje niepoprawną instrukcję.

```
>>> print( 1 / 0 ))
  File "<stdin>", line 1
  print( 1 / 0 ))
                ^
SyntaxError: invalid syntax
```

- Strzałka wskazuje, gdzie parser napotkał **błąd składniowy**. W tym przykładzie było o jeden nawias za dużo. Po usunięciu i ponownym uruchomieniu otrzymamy:

```
>>> print( 1 / 0)
 Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

- Tym razem wystąpił **wyjątek**. Ten typ błędu występuje, ilekroć poprawny składniowo kod Pythona powoduje błąd. Ostatni wiersz komunikatu definiuje typ wyjątku. Python oferuje wbudowane wyjątki, a także możliwość tworzenia samodzielnie zdefiniowanych wyjątków.

- **W Pythonie komunikaty o błędach obejmują: zgłoszony wyjątek wraz z śladem stosu** (ang. *stack trace*) – lista wszystkich wierszy oraz funkcji aktywnych w momencie, kiedy nastąpił wyjątek.
- Program, w którym zostanie wywołany wyjątek jest przerwany, chyba że wyjątek zostanie **obsłużony** – działanie **obsługi wyjątków**. Rodzaj wyjątku sugeruje rodzaj błędu.

```
1 adict = {}                          # pusty słownik
2 print(adict["klucz1"])
```

```
1 File "/home/kinga/.config/spyder-py3/temp.py", line 22, in <module>
2     print(adict["klucz1"])
3 KeyError: 'klucz1'
```

- **Wbudowane w Pythonie rodzaje wyjątków (ang. *build-in exeptions*) tworzą hierarchię wyjątków**, więcej na ten temat:
  https://pl.python.org/docs/lib/module-exceptions.html
  https://docs.python.org/3/library/exceptions.html .

Python oferuje następujące funkcjonalności do obsługi wszelkich błędów:

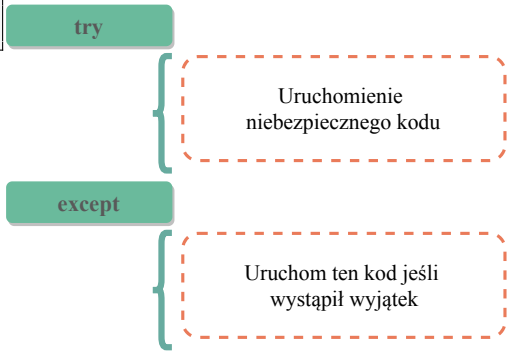
- Obsługa wyjątków (ang. *Exception Handling*) – **try/except/else/finally** – reakcja na błędy i wyjątki ich obsługa.
- Rzucanie wyjątków (zgłaszanie wyjątków) – **raise** – wznoszenie.
- Asercje (ang. *Assertions*) – **assert** – weryfikacja kodu na podstawie warunku logicznego

Wyjątki i błędy są przetwarzane przez instrukcje::

- **try/except/else/finally** – przechwytywanie wyjątków i działania oczyszczające.
- **assert** – warunkowe wywołanie wyjątku w kodzie, weryfikacja kodu.
- **raise** – jawne wywołanie wyjątku w kodzie.

```
try:
    niebezpieczny_kod
except RodzajWyjątku:
    wykonaj_alternatywę
```

- W pierwszej kolejności wykonana zostanie **klauzula try** (część kodu pomiędzy `try` i `except`) czyli `niebezpieczny_kod`.
- Jeśli nie wystąpi żaden błąd, to blok `except` zostanie pominięty i parser przejdzie dalej.
- Jeśli wystąpił błąd podczas wykonywania **klauzuli try**, wykonany zostanie blok `except`:
  - Jeśli **wyjątek jest nie zgodny** z `RodzajWyjątku` – wyjątek zostanie zgłoszony, program przerwany.
  - Jeśli **wyjątek jest zgodny** z `RodzajWyjątku` – wykonany zostanie blok `except` i program przejdzie dalej, chyba że znów trafi na wyjątek.



Program wymuszający wczytanie liczby całkowitej, Podanie przez użytkownika wartości tekstowej spowoduje błąd.

- Jeśli po instrukcji `except` nie ma żadnego wyjątku to przechwytywane są wszystkie.

```
while True:
    try:
        k = int(input('Podaj liczbę całkowitą: '))
        break
    except:
        print('To nie jest liczba całkowita. Spróbuj ponownie...')
```

```
Podaj liczbę całkowitą: tekst
To nie jest liczba całkowita. Spróbuj ponownie...
Podaj liczbę całkowitą: 5
```

Program wymuszający wczytanie liczby całkowitej, Podanie przez użytkownika wartości tekstowej spowoduje błąd.

- W tym przypadku **wyjątek jest zgodny** z `RodzajWyjątku` (`ValueError`) – wykonany zostanie blok `except` i program przejdzie dalej, chyba że znów trafi na wyjątek.

```
while True:
    try:
        k = int(input('Podaj liczbę całkowitą: '))
        break
    except ValueError:
        print('To nie jest liczba całkowita. Spróbuj ponownie...')
```

```
Podaj liczbę całkowitą: tekst
To nie jest liczba całkowita. Spróbuj ponownie...
Podaj liczbę całkowitą: 5
```

Program wymuszający wczytanie liczby całkowitej, Podanie przez użytkownika wartości tekstowej spowoduje błąd.

- W tym przypadku **wyjątek jest nie zgodny** z `RodzajWyjątku` (`RuntimeError`) – wyjątek zostanie zgłoszony, a program przerwany.

```
1 while True:
2     try:
3         k = int(input('Podaj liczbę całkowitą: '))
4         break
5     except RuntimeError:
6         print('To nie jest liczba całkowita. Spróbuj ponownie...')
```

```
1 Podaj liczbę całkowitą: s
2 ValueError      Traceback (most recent call last)
3 ----> 3        k = int(input('Podaj liczbę całkowitą: '))
4 ValueError: invalid literal for int() with base 10: 's'
```

- Po except można wymienić więcej niż jeden wyjątek.

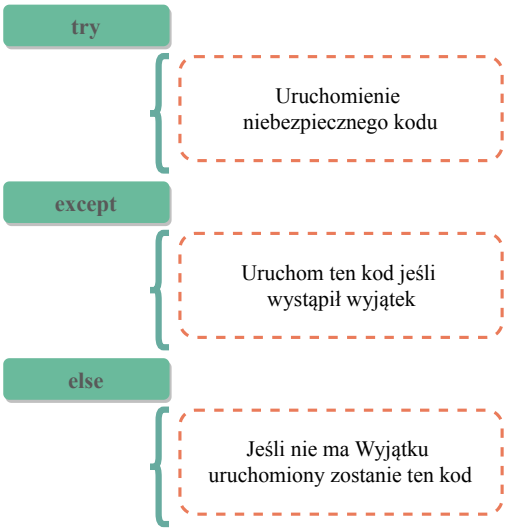
```
1 kontener, pozycja = {'a': 1, 'b': 2, 'c': 6}
2 try:
3     print(kontener[pozycja])
4 except (RuntimeError, IndexError, KeyError):
5     print('Nieprawidłowo wybrana pozycja.')
```

- Instrukcje except można wymieniać kolejno po sobie.

```
1 kontener, pozycja = {'a': 1, 'b': 2}
2 try:
3     print(kontener[pozycja])
4 except IndexError:
5     print('Pozycja poza sekwencją.')
6 except KeyError:
7     print('Błędny klucz.')
```

W Pythonie za pomocą instrukcji `else` można polecić programowi wykonanie określonego bloku kodu **tylko w przypadku braku wyjątków**

- W pierwszej kolejności wykonana zostanie **klauzula try** (część kodu pomiędzy `try` i `except`) czyli `niebezpieczny_kod`.
- Jeśli nie wystąpi żaden błąd, to blok `except` zostanie pominięty i parser przejdzie dalej.
- Jeśli wystąpił błąd w **klauzuli try**, wykonany jest blok `except`: **wyjątek jest nie zgodny** z `RodzajWyjątku` – zgłoszenie wyjątku i przerwanie programu, **wyjątek jest zgodny** z `RodzajWyjątku` – wykonanie `except` (dalej).
- Blok `else` wykona się tylko wtedy, gdy nie ma wyjątku (błędu) w bloku `try`

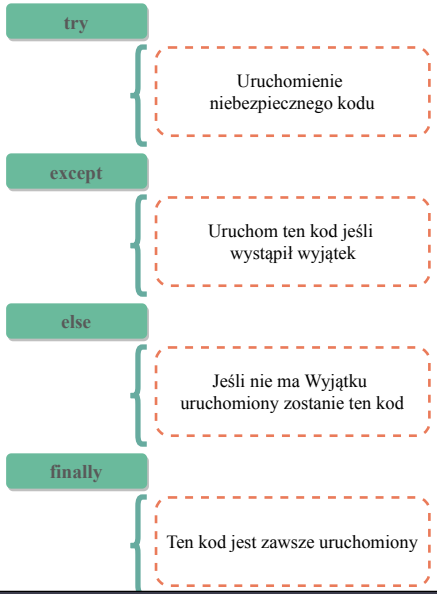


- Po wszystkich blokach `except` można opcjonalnie wstawić blok `else`. Blok ten wykona się tylko wtedy, gdy w bloku `try` nie pojawił się wyjątek

```
1 filename = raw_input("Podaj nazwę pliku: ")
2 try:
3     afile = open(filename, "r")
4 except Exception: # blok (w razie wyjątku)
5     print("Nie ma pliku o nazwie", filename)
6 else:  # blok opcjonalny (nie było wyjątku)
7     print("Plik został otwarty")
```

Blok w instrukcji `finally` wykonywany jest zawsze nawet jeśli wystąpi wyjątek w `try` lub `else`

- W pierwszej kolejności wykonana zostanie **klauzula try** (część kodu pomiędzy `try` i `except`) czyli `niebezpieczny_kod`.
- Jeśli nie wystąpi żaden błąd, to blok `except` zostanie pominięty i parser przejdzie dalej.
- Jeśli wystąpił błąd w **klauzuli try**, wykonany jest blok `except`: **wyjątek jest nie zgodny** z `RodzajWyjątku` – zgłoszenie wyjątku i przerwanie programu, **wyjątek jest zgodny** z `RodzajWyjątku` – wykonanie `except` (dalej).
- Blok `else` wykona się tylko wtedy, gdy nie ma wyjątku (błędu) w bloku `try`.
- Blok `finally` wykonywany jest zawsze.



- Opcjonalny blok `finally` wstawia się po wszystkich `except` lub po `else` (gdy występuje). Blok `finally` zostaje wykonany przed opuszczeniem bloku try, bez względu na to co w bloku `try` się działo. Blok `finally` służy do wykonania akcji "czyszczących".

```
1 filename = raw_input("Podaj nazwę pliku: ")
2 try:
3     afile = open(filename, "r")
4 except Exception: #  blok (w razie wyjątku)
5     print("Nie ma pliku o nazwie", filename)
6 else:  #  blok opcjonalny (nie było wyjątku)
7     print("Plik został otwarty")
8 finally:
9     afile.close()
```

- Instrukcja `raise` służy do jawnego wywoływania wyjątków (wywołanie wyjątku nazywamy również **zgłaszaniem wyjątku** lub **rzucanie wyjątku**)

- Składnia instrukcji jest następująca: instrukcja składa się ze słowa `raise`, po którym opcjonalnie następuje klasa zgłaszanego wyjątku lub jej instancja (nazwa wyjątku).

```
raise <instancja>     # Zgłoszenie instancji klasy (obiektu klasy)
raise <klasa()>       # Utworzenie i zgłoszenie instancji klasy
raise                 # Ponowne zgłoszenie ostatniego wyjątku
raise Exception('dodatkowa informacja') # przykład
```

- Na przykład:

```
raise IndexError   # Klasa (utworzenie instancji)
raise IndexError() # Instancja (utworzona w instrukcji)
raise ExceptionError('dodatkowa informacja')
```

- Przykład zgłaszania wyjątku ze stowarzyszoną informacją:

```
1 try:
2     1 / 0
3 except ZeroDivisionError:
4     raise ZeroDivisionError('Próbowałeś dzielić przez zero!')
```

```
1 File "/home/kinga/.config/spyder-py3/temp.py", line 27, in <module>
2 ZeroDivisionError: Próbowałeś dzielić przez zero!
```

- `raise` podane bez parametrów w bloku `except` wywołuje ostatni obsługiwany wyjątek:

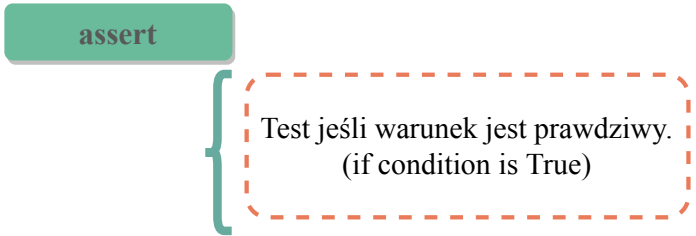
```
1 a, b = 1, 0
2 try:
3     print(a / b)
4 except:
5     print('Nie wiem co się stało :-(')
6     raise
```

```
1 if size < 0:
2     raise ValueError('liczba musi być dodatnia')
```

```
1 if size < 0:
2     print('liczba musi być dodatnia')
```

- Zgłoszenie błędu za pomocą `raise` powoduje **zatrzymanie całego programu** w tym momencie (chyba że zostanie przechwycony i obsłużony wyjątek).
- Podczas gdy komunikat z funkcji `print` po prostu zapisuje coś na standardowe wyjście (`stdout`) – dane wyjściowe mogą być przesyłane do innego narzędzia (aniżeli wiersz poleceń) lub aplikacja może być uruchamiana nie z wiersza poleceń, wtedy komunikat `print` nie będzie nigdy widoczny.

- **Asercja (zapewnienie)** to takie wyrażenie które potwierdza (lub sprawdza prawdziwość) warunku w kodzie – wyrażenie boolowskie, które potwierdza wartość logiczną warunku.
- Instrukcja `assert` jest w pewnym sensie przypadkiem specjalnym na cele debugowania. Jest to przede wszystkim po prostu składniowy skrót dla często wykorzystywanego wzorca z instrukcją `raise`, który można sobie wyobrazić jako warunkową instrukcję `raise`.



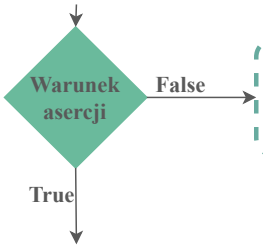
- Innymi słowy, jeśli test okaże się fałszem, Python zgłasza wyjątek; element danych (jeśli jest podany) służy jako argument konstruktora wyjątku. Tak jak wszystkie wyjątki, zgłoszony `AssertionError` zakończy działanie programu, jeśli nie przechwycimy go za pomocą instrukcji `try`; w tym drugim przypadku element danych zostanie wyświetlony jako część komunikatu o błędzie (Lutz, 2011).

- Podstawowa składnia instrukcji rzuca wyjątkiem `AssertionError`, gdy wartość zwracana przez wyrażenie nie jest przez wartością bolonowską `True`:

```
1 assert warunek
```

- Wywołanie z komunikatem, wtedy wyjątek `AssertionError` ma dodatkowo komunikat:

```
1 assert warunek, "komunikat"
```



- Podczas sprawdzania typów/sprawdzania poprawnych danych wejściowych.
- Do sprawdzania argumentów funkcji
- Sprawdzanie wyjścia funkcji.
- Jako debugger do zatrzymania w przypadku wystąpienia błędu.
- W kodzie testowym (Testowanie kodu).
- W wykrywaniu nadużycia interfejsu przez innego programistę.

```
1 def div(p,q):
2     assert q!=0, "Nie można dzielić przez zero\nSpróbuj ponownie"
3     return p/q
4 div(2,0)
```

```
1 AssertionError: Nie można dzielić przez zero
2 Spróbuj ponownie
```

```
1 lista = ["a","b","c"]
2 assert "x" in lista
```

```
1 AssertionError
```

```
1 lista = ["a","b","c"]
2 assert "x" in lista, "x is not in the list"
```

```
1 AssertionError: x is not in the list
```

assert jest podobny w działaniu do raise

```python
def foo1(param):
    assert param, "fail"
```

```python
def foo2(param):
    if not param:
        raise AssertionError("fail")
```

Do najważniejszych zadań występowania wyjątków należą:

- **Obsługa błędów** Python zgłasza wyjątki za każdym razem, kiedy w czasie wykonywania programu znajduje w nim błąd. Możemy przechwytywać i odpowiadać na błędy w kodzie lub ignorować zgłaszane wyjątki. Jeśli błąd jest ignorowany, do gry wkracza domyślna obsługa wyjątków Pythona – zatrzymuje ona program i wyświetla komunikat o błędzie. Jeśli takie zachowanie nam nie odpowiada, musimy zapisać w kodzie instrukcję `try`, która przechwyci wyjątek i pozwoli go obsłużyć. Po wykryciu błędu Python przeskoczy do programu obsługi `try`, a program wznowi wykonywanie po `try`.
- **Powiadomienia o zdarzeniach** Wyjątki można również wykorzystać do sygnalizowania poprawnych warunków bez konieczności przekazywania flag wyników w programie lub jawnego ich sprawdzania. Procedura wyszukiwania może na przykład zgłosić wyjątek dla niepowodzenia, zamiast zwracać liczbowy kod wyniku (i mieć nadzieję, że kod nigdy nie będzie miał poprawnego wyniku).

- **Obsługa przypadków specjalnych** Czasami jakiś warunek może występować tak rzadko, że trudno jest uzasadnić przekształcanie kodu w taki sposób, by go obsługiwał. Często możemy wyeliminować kod specjalnych przypadków, obsługując je zamiast tego w programach obsługi wyjątków na wyższych poziomach.
- **Działania końcowe** Jak zobaczymy, instrukcja `try`/ `finally` pozwala nam zagwarantować, że wymagane operacje czasu zakończenia zostaną wykonane bez względu na obecność lub nieobecność wyjątków w programach.
- **Niezwykły przebieg sterowania** I wreszcie, ponieważ wyjątki są rodzajem operacji „goto" wysokiego poziomu, możemy ich użyć jako podstawy do implementacji egzotycznego przebiegu programu. Choć na przykład Python oficjalnie nie obsługuje nawracania (ang. *backtracking*), można je w tym języku programowania zaimplementować za pomocą wyjątków, a także niewielkiej ilości logiki obsługującej służącej do rozwinięcia przypisania. W Pythonie nie ma instrukcji „goto" (na całe szczęście!), jednak wyjątki mogą czasami pełnić te same role.

- `raise` – pozwala na jawne zgłoszenie wyjątku w dowolnym momencie.
- `assert` – pozwala zweryfikować, czy określony warunek jest spełniony i zgłosić wyjątek, jeśli nie jest spełniony.
- `try` – w klauzuli `try` wszystkie instrukcje są wykonywane do momentu napotkania wyjątku.
- `except` – służy do wychwytywania i obsługi wyjątków, które występują w klauzuli `try`.
- `else` – pozwala oprogramować sekcje, które powinny być uruchamiane tylko wtedy, gdy nie występują wyjątki w klauzuli `try`.
- `finally` – umożliwia wykonanie sekcji kodu, które powinny być zawsze uruchamiane, z wcześniejszymi wyjątkami lub bez nich.

Pełaną lista oraz wyjaśnienie wyjątków i błędów można znaleźć:

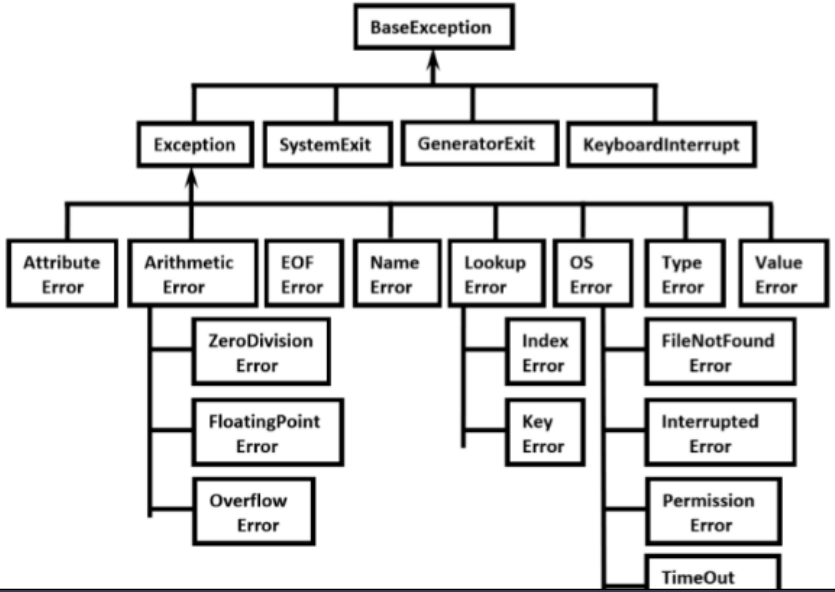
- https://data-flair.training/blogs/python-exception/
- https://www.tutorialspoint.com/python3/python_exceptions.htm

- `SyntaxError` – wywoływany, gdy występuje błąd w składni Pythona.
- `IndentationError` – wywoływany, gdy wcięcie nie jest poprawnie określone.
- `IndexError` – wywoływany, gdy indeks nie znajduje się w sekwencji.
- `KeyError` – wywoływany, gdy określony klucz nie został znaleziony w słowniku.
- `NameError` – wywoływany, gdy identyfikator (zmienna) nie zostanie znaleziony w lokalnej lub globalnej przestrzeni nazw.
- `AttributeError` – w przypadku złego odwołania lub przypisania atrybutu funkcji.
- `TypeError` – wywoływany, gdy próbuje się wykonać operację lub funkcję, która jest niepoprawna dla określonego typu danych.

- `ImportError` – wywoływane, gdy nie powiedzie się instrukcja importu.
- `StopIteration` – gdy metoda `next()` iteratora nie wskazuje na żaden obiekt.
- `SystemExit` – wywołany przez funkcję `sys.exit()`.
- `Exception` – klasa podstawowa dla wszystkich wyjątków
- `StandardError` – klasa podstawowa dla wszystkich wbudowanych wyjątków oprócz `StopIteration` i `SystemExit`.
- `ArithmeticError` – klasa podstawowa dla wszystkich błędów występujących w obliczeniach numerycznych.
- `OverflowError` – gdy obliczenia przekraczają maksymalny limit dla typu numerycznego.
- `FloatingPointError` – gdy obliczenie dla liczb zmiennoprzecinkowych są niepoprawne.
- `ZeroDivisonError` – wywoływany, gdy liczba jest dzielona przez zero
- `AssertionError` – wywoływane w przypadku niepowodzenia instrukcji `assert`.
- `EOFError` – gdy nie ma danych wejściowych z funkcji `input()` (koniec pliku)

- `KeyboardInterrupt` – wywoływany, gdy użytkownik przerywa wykonywanie programu, zwykle przez naciśnięcie `Ctrl + c`.
- `LookupError` – klasa podstawowa dla wszystkich błędów wyszukiwania.
- `RuntimeError` – wywoływany, gdy wygenerowany błąd nie należy do żadnej kategorii.
- `UnboundLocalError` – wywoływany podczas próby dostępu do zmiennej lokalnej w funkcji lub metodzie, ale nie przypisano do niej żadnej wartości.
- `EnvironmentError` – klasa podstawowa dla wszystkich wyjątków występujących poza środowiskiem Python.
- `IOError` – gdy operacja inputoutput kończy się niepowodzeniem, na przykład instrukcja `print` lub funkcja `open()` podczas próby otwarcia nieistniejącego pliku.
- `OSError` – wywoływany z powodu błędów związanych z systemem operacyjnym.
- `SystemError` – wywoływany, gdy interpreter znajdzie problem wewnętrzny, ale gdy napotka ten błąd, interpreter Pythona nie kończy działania.

- `SystemExit` – wywoływany, gdy interpreter Pythona jest zamykany za pomocą funkcji sys.exit (). Jeśli nie zostanie obsłużony w kodzie, spowoduje to wyjście interpretera.
- `ValueError` – wywoływane, gdy wbudowana funkcja dla typu danych ma poprawny typ argumentów, ale argumenty mają określone niepoprawne wartości.
- `NotImplementedError` – wywoływany, gdy abstrakcyjna metoda, która musi zostać zaimplementowana w odziedziczonej klasie, nie jest tak naprawdę implementowana.


BaseException
Exception
SystemExit
GeneratorExit
KeyboardInterrupt
Attribute Error
Arithmetic Error
EOF Error
Name Error
Lookup Error
OS Error
Type Error
Value Error
ZeroDivision Error
FloatingPoint Error
Overflow Error
Index Error
Key Error
FileNotFound Error
Interrupted Error
Permission Error
TimeOut

- Jak stosować wyjątki: https://rk.edu.pl/pl/wyjatki-5/
- hierarchia i obsługa wyjątków: https://docs.python.org/3/library/exceptions.html
- wyjątki w Pythonie: https://www.tutorialspoint.com/python/python_exceptions.htm
- Tutorial, przykłady wyjatków: https://data-flair.training/blogs/python-assert/
- Tutorial, obsługa wyjatków: https://data-flair.training/blogs/python-exception-handling/
- Tutorial: https://realpython.com/python-exceptions/

A. Downey. *Myśl w jezyku Python*. 2016.

M. Lutz. *Python. Wprowadzenie*. Helion, 2011.

Dziękuje za uwagę

Kinga Węzka kinga.wezka@pw.edu.pl