

# Obsługa i przechwytywanie wyjątków

## Zagadnienia poruszane w ramach modułu

- Rodzaje wyjątków i ich zastosowanie
- Podstawy działania wyjątków
- Sposób przechwytywania wyjątków
- Implikacje i niespodzianki stosowania wyjątków
- Asercje

## Rodzaje wyjątków i ich zastosowanie (1 z 3)

- Wyjątki w programie Python mogą być stosowane do różnych celów:
  - Obsługi błędów – sytuacje braku zasobów
  - Niezwykłe i wyjątkowe przepływy sterowania – w bardzo uzasadnionych przypadkach
  - Obsługa przypadków specjalnych – przepływ uzależniony od specyficznego zestawu danych, np. ich końca lub błędu formatu danych
  - Powiadamiania o zdarzeniach – w przypadku obsługi zdarzeń asynchronicznych

## Rodzaje wyjątków i ich zastosowanie (2 z 3)

- Wyjątki w języku Python, połączone są w hierarchię obiektów wyprowadzoną od **BaseException**
- Własne obiekty wyjątków należy wyprowadzać z klasy **Exception** a nie z klasy **BaseException**
- Można wyprowadzić wyjątek z klas pochodnych
- Wszystkie wyjątki są zdefiniowane w module **exceptions**
- Modułu **exceptions** nie należy importować jawnie, jego obiekty są dostępne w głównej przestrzeni nazw

## Rodzaje wyjątków i ich zastosowanie (3 z 3)

- Instrukcje obsługi wyjątków, stosowane są w 2 różnych kontekstach:
  - **try, except, else:**
    - Jeśli wystąpił wyjątek w bloku **try**, Python uruchamia pierwsze pasujące instrukcje z bloków **except**. Sterowanie wychodzi poza blok **try** (chyba że **except** rzucił wyjątkiem)
    - Jeśli wyjątek rzucony w **try** nie pasuje do żadnej z klauzul **except**, propagowany jest o poziom wyżej.
    - Jeśli w bloku **try** nie powstał wyjątek, wywoływane są instrukcje z bloku **else**
  - **try, finally:**
    - Jeśli nie ma wyjątku w bloku **try**, uruchamiane są instrukcje z tegoż bloku i uruchamia instrukcje z bloku **finally**
    - Jeśli rzucono wyjątek w bloku **try**, Python uruchamia instrukcje z **finally** i propaguje wyjątek o poziom wyżej

## Przykłady prostej obsługi wyjątków

- Wyłapanie dowolnego wyjątku:
- Z nazwanym wyjątkiem:
- Z argumentami i komunikatem:

```

C:\Windows\system32\cmd.exe - python
>>> try:
...     print "Podaj liczbę:",
...     a = raw_input()
...     print "Kwadrat %d to %d" % (a, a**2)
... except:
...     print "wylapano wyjatek"
Podaj liczbę: 3wylapano wyjatek
>>> # wciśnięto Ctrl-C_

```

```

C:\Windows\system32\cmd.exe - python
>>> try:
...     a = raw_input()
... except KeyboardInterrupt as e:
...     print e.args, e.message
...
()__main__:4: DeprecationWarning: BaseExcepti
f Python 2.6
>>>

```

```

C:\Windows\system32\cmd.exe - python
>>> try:
...     a =raw_input()
... except Exception as a:
...     print a
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
>>> # i znów Ctrl-C, ale z nawanym wyjątkiem

```

## Hierarchia wyjątków (1 z 4)

- Od głównej klasy **BaseException** dziedziczą:
  - **SystemExit** – wywoływany w przypadku wykrycia błędu wewnętrznego interpretera
  - **KeyboardInterrupt** – wywoływany w trakcie wprowadzania sekwencji **Ctrl-C** lub **Delete**
  - **GeneratorExit** – wywoływana gdy generator wywołuje metodę **close()**
  - **Exception** – wszystkie własne klasy można wyprowadzać z tej klasy
    - **StopIteration** – klasa obecnie nieobsadzona przez uszczegółowione inne klasy
    - **StandardError** – klasa bazowa dla innych wyjątków
    - **Warning** – klasa bazowa dla ostrzeżeń

## Hierarchia wyjątków (2 z 4)

- Z klasy **Warning** wyprowadzono:  
**DeprecationWarning**, **PendingDeprecationWarning**, **RuntimeWarning**, **SyntaxWarning**, **UserWarning**, **FutureWarning**, **ImportWarning**, **UnicodeWarning**, **BytesWarning**
- Znaczenie tych klas zostało wyczerpująco opisane w podręczniku systemowym

## Hierarchia wyjątków (3 z 4)

- Z klasy **StandardError** wprowadzono:

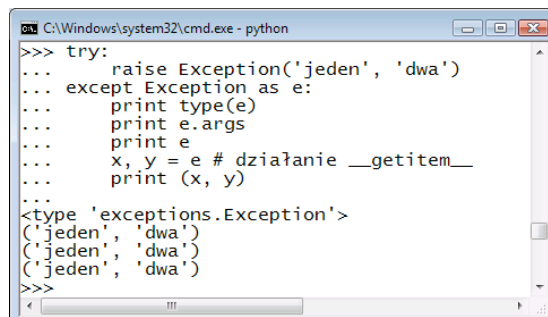
- **StandardError**

- BufferError
- ArithmeticError
  - FloatingPointError
  - OverflowError
  - ZeroDivisionError
- AssertionError
- AttributeError
- EnvironmentError
- EOFError
- ImportError
- LookupError:
  - IndexError, KeyError

- MemoryError
- NameError
- NameError
  - UnboundLocalError
- ReferenceError
- RuntimeError
  - NotImplementedError
- SyntaxError
  - IndentationError
    - TabError
- SystemError
- TypeError
- ValueError
  - UnicodeError:
    - UnicodeDecodeError
    - UnicodeEncodeError
    - UnicodeTranslateError

## Podstawy działania wyjątków

- Rzucić wyjątek można z pomocą **raise()**
- Rzucany wyjątek można wzbogacić o argumenty wywołania
- Argumenty te po przechwyceniu można wykorzystać do raportowania błędu

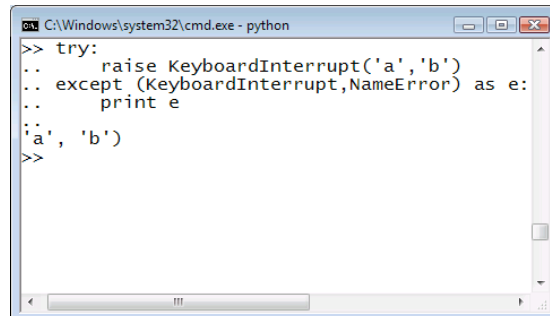


```

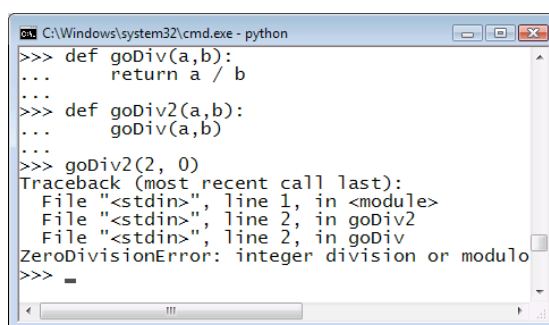
C:\Windows\system32\cmd.exe - python
>>> try:
...     raise Exception('jeden', 'dwa')
... except Exception as e:
...     print type(e)
...     print e.args
...     print e
...     x, y = e # działanie __getitem__
...     print (x, y)
...
<type 'exceptions.Exception'>
('jeden', 'dwa')
('jeden', 'dwa')
('jeden', 'dwa')
>>>
  
```

## Sposób przechwytywania wyjątków

- Przechwytywane wyjątki można podawać także jako listę wartości
- Rzucany wyjątek zawiera także ślad stosu wywołania

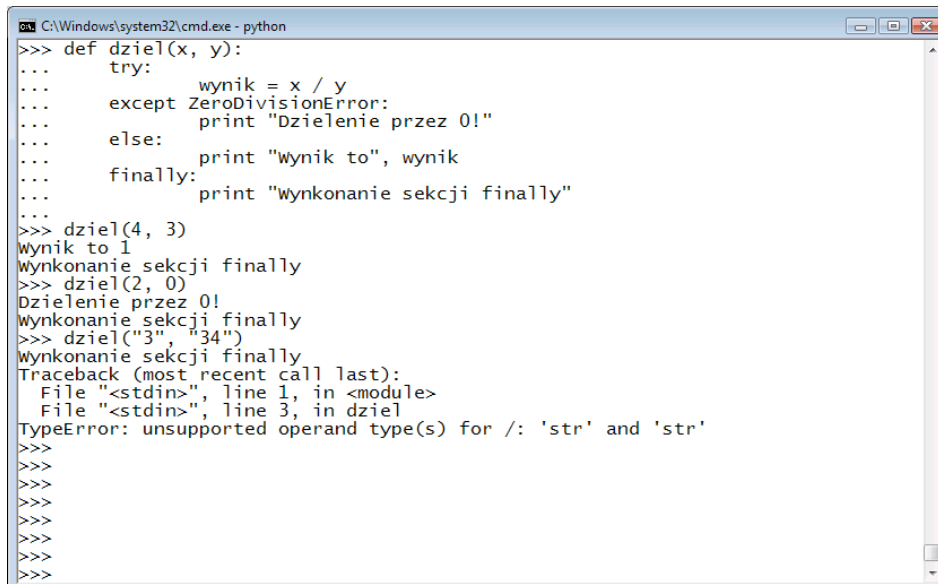


```
C:\Windows\system32\cmd.exe - python
>> try:
..     raise KeyboardInterrupt('a','b')
.. except (KeyboardInterrupt,NameError) as e:
..     print e
'a', 'b'
>>
```



```
C:\Windows\system32\cmd.exe - python
>>> def goDiv(a,b):
...     return a / b
>>> def goDiv2(a,b):
...     goDiv(a,b)
...
>>> goDiv2(2, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in goDiv2
  File "<stdin>", line 2, in goDiv
ZeroDivisionError: integer division or modulo
>>> _
```

## Wszystkie sekcje obsługi wyjątków



```
C:\Windows\system32\cmd.exe - python
>>> def dziel(x, y):
...     try:
...         wynik = x / y
...     except ZeroDivisionError:
...         print "Dzielenie przez 0!"
...     else:
...         print "Wynik to", wynik
...     finally:
...         print "Wykonanie sekcji finally"
...
>>> dziel(4, 3)
Wynik to 1
Wykonanie sekcji finally
>>> dziel(2, 0)
Dzielenie przez 0!
Wykonanie sekcji finally
>>> dziel("3", "34")
Wykonanie sekcji finally
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in dziel
TypeError: unsupported operand type(s) for /: 'str' and 'str'
>>>
>>>
>>>
>>>
>>>
```

## Implikacje i niespodzianki stosowania wyjątków

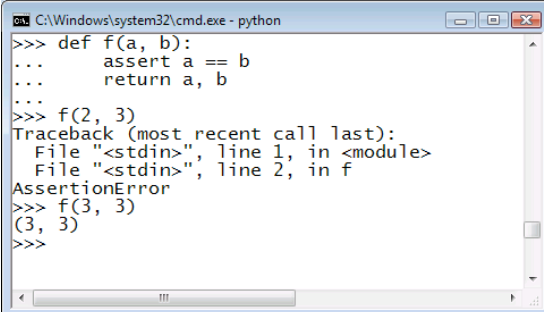
- Wyjątki są dopasowywane na podstawie relacji identyczności (**is**)
- Podawać należy zawsze wyjątki które chcesz wyłapać **explicite**

## Asercje

- Asercja to sprawdzenie warunku oraz bezwarunkowe zakończenie programu w przypadku jego niespełnienia
- Kod asercji jest wyłączany z kodu programu w przypadku uruchomienia kompilacji do kodu \*.pyc z optymalizacją (przełącznik **-O** interpretera)
- Asercja jest wbudowana w kod gdy zmienna **\_\_debug\_\_** ma wartość **True**
- Zmiennej tej nie można nadpisać, jej wartość jest uzależniona od przełącznika **-O** interpretera

## Przykład użycia asercji

- Asercja rzuca wyjątek **AssertionError** który można przechwycić
- Asercje nie są sposobem na obsługę regularnych błędów aplikacji. Mają jedynie pomagać na etapie jej tworzenia



```
C:\Windows\system32\cmd.exe - python
>>> def f(a, b):
...     assert a == b
...     return a, b
...
>>> f(2, 3)
>>> f(3, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
AssertionError
>>>
```

## Quiz

- Jak nazywa się instrukcja wywołująca wyjątek?
- Jakie dane przechowywane są w atrybucie **message** obiektu wyjątku?
- Co oznacza pojęcie śladu wyjątku?
- Czy jest prawdą stwierdzenie: blok **else** wyjątku jest uruchamiany zawsze gdy wyjątek jest wychwycony?
- Z jakiej klasy wyprowadzono błędy interpretera?
- Z jakiej (najbardziej ogólnej) klasy można wyprowadzić swój wyjątek?
- Jaka jest różnica w działaniu asercji i wyjątku?



# Ćwiczenie

- Wykonaj ćwiczenie 7.

# Podsumowanie

## Rodzaje wyjątków i ich zastosowanie

- Wyjątki tworzą hierarchię klas które zastosowane są w całym interpreterze
- Wyjątki należy stosować z umiarem
- Wyjątek jest przeznaczony do wychwycenia **istotnych odchyłeń działania aplikacji** które najczęściej nie zależą od normalnego przepływu sterowania
- Nie należy poprzez wyjątki obsługiwać przepływu sterowania w algorytmach

## Podstawy działania wyjątków

- Blok **try** chroni kod przed rzucanym wyjątkiem
- Blok **except** rozpoczyna blok kodu obsługi wyjątku
- Może być wiele sekcji **except** lub każda z sekcji może wyłapywać listę wyjątków podaną w postaci listy
- Blok **else** wykonywany jest gdy wyjątek nie jest rzucony
- Blok **finally** jest blokiem finalizacji wyjątku

## Implikacje i niespodzianki stosowania wyjątków

- Pamiętaj że wyjątki dopasowywane są na podstawie relacji identyczności
- Podawaj dokładnie listę wyjątków które chcesz wyłączyć

## Asercje

- Asercje są warunkami sprawdzanymi w trakcie wykonywania kodu
- Są usuwane gdy wykonujemy kod z opcją **-O** interpretera
- Nie powinny być częścią implementowanego algorytmu