**Wyjątki i ich obsługa**

Założę się, że w trakcie tworzenia programów, nieraz spotkałeś się z podobnymi do poniższych błędami:

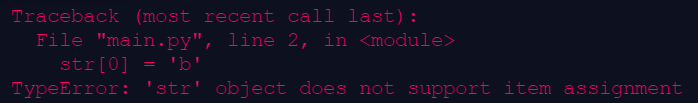
Przykład: *Wyjątek TypeError.*

**Kod:**

str = "Lorem ipsum."

str[0] = 'b'

**Efekt:**



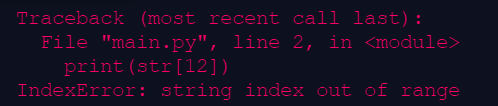
Przykład 2: *Wyjątek IndexError.*

**Kod:**

str = "Lorem ipsum."

print(str[12])

**Efekt:**



Przykład 3: *Wyjątek TypeError.*

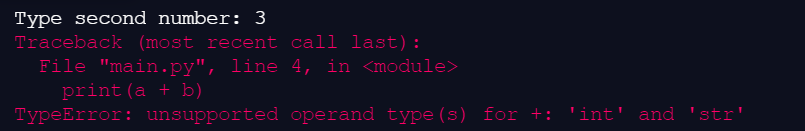
**Kod:**

a = 4

b = input("Type second number: ")

print(a + b)

**Efekt:**

****

Przyznaj szczerze, że na początku Twojej przygody z programowaniem, błędy te spędzały Ci sen z powiek. Uniemożliwiały one bowiem poprawne działanie programu i mówimy, że w ten sposób Nasza aplikacja się crashowała.

Błędy te nazywane są **wyjątkami** (ang. exceptions). Kiedy taki exception się pojawia (np. TypeError, IndexError), interpreter **zatrzymuje proces działania** programu.

Nasza aplikacja nie będzie poprawnie funkcjonowała, dopóki nie naprawimy miejsca w kodzie, który powoduje wyjątek lub, o czym za chwilę się dowiesz, nie zdefiniujemy, co ma się zadziać, gdy takowy wystąpi.

**Obsługiwanie wyjątków**

Muszę na pewno sprostować, co rozumiem przez obsługiwanie danego wyjątku. Chodzi o to, że czasami jesteśmy w pełni świadomi możliwości wystąpienia błędu w programie i wówczas chcielibyśmy powiedzieć interpreterowi, żeby nie crashował programu, ale np. **wyświetlał odpowiedni komunikat**.

Przyrównałbym to nawet do działania instrukcji warunkowej

if wystąpił\_wyjątek:  
 …

Definiowanie własnych zachowań, które mają nastąpić, gdy program rzuci wyjątkiem, nazywane jest właśnie **obsługiwaniem wyjątków** (ang. exceptions handling).

W Pythonie do obsługiwania wyjątków służy **wyrażenie try**.

Umieszcza się w nim kod sprawiający “potencjalnie zagrożenie” - czyli taki, który może rzucić wyjątkiem. Natomiast wyrażenie, w którym definiujemy, co ma się wykonać, gdy taki wyjątek nastąpi, określamy jako **except**.

Wyobraźmy sobie następujący program:

a = int(input("Type first number: "))

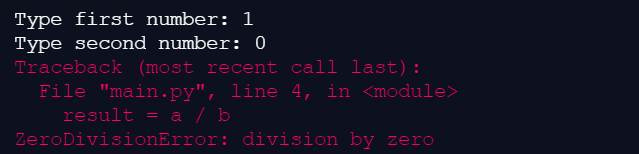
b = int(input("Type second number: "))

result = a / b

print(f"The result is equal to: {result}")

Użytkownik ma podać dwie liczby w postaci dzielnej i dzielnika. Program faktycznie będzie działał poprawnie i wyświetli Nam oczekiwane wyniki, dopóki **nie wprowadzimy liczby 0 do zmiennej b** (jak wtedy będzie mogła zostać wyliczona zmienna result, jeżeli nie możemy dzielić jakiejkolwiek wartości przez 0?).

W takiej sytuacji napotkamy rzucony wyjątek.



Przykład: *Obsłużenie sytuacji, gdy użytkownik poda nieprawidłową liczbę.*

Przedstawiony wyżej problem możemy oczywiście rozwiązać, stosując choćby instrukcję warunkową i poprzez sprawdzanie, czy dzielnik jest różny od 0, np.

**Kod:**

a = int(input("Type first number: "))

b = int(input("Type second number: "))

if not b:

print("Can't divide by 0!")

else:

result = a / b

print(f"The result is equal to: {result}")

Jednak o wiele zgrabniejszym rozwiązaniem będzie użycie **bloku try** i **except** do zabezpieczenia się przed potencjalnym błędem.

**Kod:**

a = int(input("Type first number: "))

b = int(input("Type second number: "))

try:

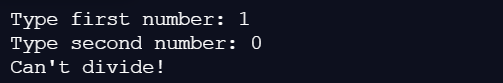
result = a / b

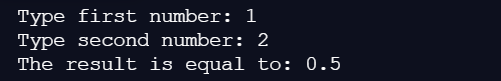
print(f"The result is equal to: {result}")

except:

print("Can't divide!")

**Wynik:**





Tak jak zostało wcześniej wspomniane, blok try służy do przechowywania kodu, **który może rzucić potencjalny wyjątek**.   
Dzięki temu w momencie, gdy dowolna linia kodu z try spowoduje rzucenie wyjątkiem, **flow programu przekierowywany jest bezpośrednio do bloku except** i wykonywane są odpowiednie czynności. Jeżeli natomiast nie pojawi się żaden z wyjątków, to **except jest pomijane**.

Przykład 2: *Dzielenie wartości liczbowej przez wszystkie elementy listy.*

values = ['a', 0, 2, 3.5]

for elem in values:

try:

print("The entry is", elem)

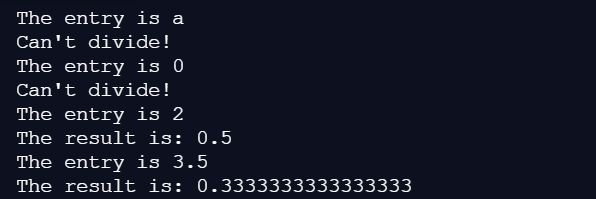
r = 1/elem

print(f"The result is: {r}")

except:

print("Can't divide!")

**Wynik:**



Błędy w powyższym przykładzie wynikają z kilku możliwości: dzielnikiem może być bowiem albo **typ str** albo **wartość 0**.

Niedociągnięcie w powyższym przykładzie wynika z tego, że zarówno dla wyjątku powstałego w wyniku dzielenia liczby przez literę oraz liczby przez 0 wyświetlamy ten sam komunikat (“Can’t divide”).

Aby personalizować komunikat i wyświetlać odpowiednią wiadomość w zależności od przypadku, np. **“Can’t divide by letter”** lub **“Can’t divide by 0”,** należy dodać do klauzuli except dodatkowy parametr:

except NazwaWyjątku:

…

**Przykład:**

try:

# zrób coś

except ValueError:

# obsłuż wyjątek ValueError

...

except TypeError, ZeroDivisionError:

# obsłuż kilka wyjątków

# TypeError i ZeroDivisionError

...

except:

# obsłuż wszystkie pozostałe wyjątki

…

**Poprawiony przykład:**

values = ['a', 0, 2, 3.5]

for elem in values:

try:

print("The entry is", elem)

r = 1/elem

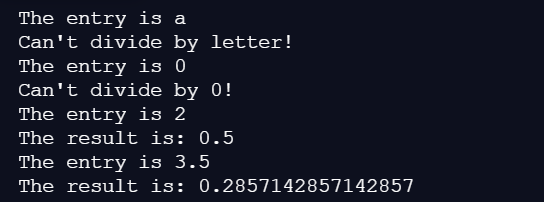
print(f"The result is: {r}")

except ZeroDivisionError:

print("Can't divide by 0!")

except TypeError:

print("Can't divide by letter!")



**UWAGA:**

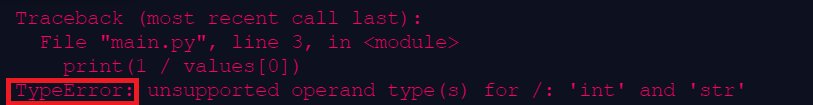
Jeżeli nie wiesz, jaka jest nazwa wyjątku rzucanego w danym przypadku, po prostu uruchom program i zobacz, jakim błędem kończy się wykonanie.

**Przykład:**

values = ['a', 0, 2, 3.5]

print(1 / values[0])

**Wynik:**



**Ręczne rzucanie wyjątkiem**

Czasami ręczne rzucanie wyjątkami w ściśle określonych momentach może poprawić czytelność i czystość kodu. Przeanalizujmy następującą sytuację.

Przykład: *Stwórz funkcję obliczającą iloczyn liczb dodatnich. Zabezpiecz się przed próbą wykonania operacji na liczbach niedodatnich.*

Chcemy stworzyć funkcję, która obliczać będzie iloczyn przesłanych argumentów. Warunkiem do przeprowadzenia operacji jest jednak fakt, że **każda przesłana liczba musi być dodatnia**.   
W przeciwnym razie mamy poinformować użytkownika o nieprawidłowych danych i zwrócić jakąś wartość wyniku (np. -1).

**Kod:**

def mul\_positive\_nums(\*args):

result = 1

for num in args:

if num <= 0:

print("Each number shoud be positive!")

return -1

else:

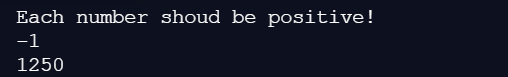
result \*= num

return result

print(mul\_positive\_nums(1, 2, 10, 20, -1))

print(mul\_positive\_nums(5, 5, 5, 10))

**Wynik:**



Można to jednak zapisać o wiele ładniej:

def mul\_positive\_nums(\*args):

result = 1

for num in args:

if num <= 0:

**raise ValueError("Only positive numbers!")**

else:

result \*= num

return result

try:

print(mul\_positive\_nums(1, 2, 10, 20, -1))

except ValueError as e:

print(e)

**Wynik:**



Zwróć uwagę na pogrubioną linię kodu. Aby rzucić własnym wyjątkiem, wprowadziliśmy linię **raise ValueError(“Only positive numbers!”)**.

W momencie gdy warunek if value <= 0 faktycznie będzie spełniony i ta linia zostanie wykonana, działanie programu przekierowywane jest do wyrażenia except.

Zwróć również uwagę, że wraz z rzuceniem wyjątku ValueError, określiliśmy jego komunikat (“Only positive numbers!”). Równie poprawne byłoby rzucenie wyjątku bez wiadomości, czyli raise ValueError.   
Niemożliwe jednak wtedy byłoby wykonanie polecenia print(e) i wyświetlenie odpowiedniego komunikatu wraz z wyjątkiem w bloku except.

**Try, except, else i finally…**

W pewnych sytuacjach może zajść potrzeba uruchomienia skrawka kodu wtedy, gdy instrukcje wewnątrz try przebiegły bez błędu.   
Wówczas z pomocą przychodzi **słowo kluczowe else**.

W przypadku natomiast, gdybyśmy chcieli zawsze uruchamiać pewną część kodu, niezależnie od tego, czy został rzucony wyjątek, czy też nie, użyjemy **finally**.

Schemat może mieć się następująco:



Przykład: *Wykorzystanie bloku else.*

try:

age=int(input('Enter your age: '))

except:

print ('You have entered an invalid value.')

else:

if age <= 21:

print('You are not allowed to enter, you are too young.')

else:

print('Welcome, you are old enough.')

**UWAGA:**

Jeżeli uważasz, że powyższy przykład możemy również zapisać tak:

try:

age=int(input('Enter your age: '))

if age <= 21:

print('You are not allowed to enter, you are too young.')

else:

print('Welcome, you are old enough.')

except:

print ('You have entered an invalid value.')

To w pełni się z Tobą zgadzam, jednak zauważ, że zapis z użyciem else znacznie poprawia czytelność.

Blok finally, wykorzystywany jest najczęściej, gdy chcesz “posprzątać” po pewnych operacjach. Najlepszym przykładem będzie otwarcie **pliku w bloku try**, wykonanie pewnych operacji i **jego automatyczne zamknięcie w finally**.

Przykład: *Sprzątanie po pliku w bloku finally.*

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

try:

file.write("Testing.")

print("Writing to file.")

except IOError:

print("Could not write to file.")

else:

print("Write successful.")

finally:

file.close()

print("File closed.")

**Tworzenie własnych rodzajów wyjątków**

W momencie, gdy chcesz spersonalizować rodzaj rzucanego wyjątku, możesz utworzyć własną klasę, której obiekty **będą właśnie reprezentowały pojedyncze exceptiony**.

Jedyny warunek, który musi zostać spełniony przy tworzeniu własnych klas - wyjątków - **muszą one dziedziczyć po rodzicu Exception**.

>>> class CustomError(Exception):

... pass

...

>>> raise CustomError

Traceback (most recent call last):

...

\_\_main\_\_.CustomError

Tworzenie własnych wyjątków ma prosty cel. Dostarczają one łatwiejsze nawigowanie po programie.

Pamiętasz przykład z dzieleniem liczby przez typ str i pojawiającym się wtedy **wyjątkiem TypeError**? Aby usprawnić czytanie kodu, moglibyśmy utworzyć własny wyjątek, np. **StringAsDividerError** i ręcznie go rzucać podczas próby dokonania nieprawidłowego dzielenia.

Ponadto tworzone przez Nas wyjątki, mogą być przez Nas ulepszane, dzięki możliwości dodania do nich niektórych metod specjalnych, np. **\_\_str\_\_().**

Przykład z wykorzystaniem tejże funkcjonalności poniżej.

Przykład: *Sprawdzaj, czy liczba znajduje się w określonym przedziale.*

class ValueTooSmallError(Exception):

def \_\_str\_\_(self):

return "Typed value is too small!"

class ValueTooBigError(Exception):

def \_\_str\_\_(self):

return "Typed value is too big!"

def get\_num\_in\_range(start: int, end: int):

num = int(input(f"Type a number in range from {start} to {end}: "))

if num < start:

raise ValueTooSmallError

elif num > end:

raise ValueTooBigError

else:

return num

try:

print(get\_num\_in\_range(1, 5))

except (ValueTooSmallError, ValueTooBigError) as e:

print(e)

**Wynik:**







**Wyjaśnienie:**

Powyższy przykład jest prosty, ale bardzo dobrze odzwierciedla praktyczność tworzenia własnych klas - wyjątków. Powiem Ci więcej, Nasz program można by ulepszyć, wprowadzając następujące zmiany!

class ValueTooSmallError(Exception):

def \_\_init\_\_(self, value):

self.value = value

def \_\_str\_\_(self):

return f"Typed value - {self.value} - is too small!"

class ValueTooBigError(Exception):

def \_\_init\_\_(self, value):

self.value = value

def \_\_str\_\_(self):

return f"Typed value - {self.value} - is too big!"

I w momencie rzucania wyjątków, moglibyśmy zapisać, np. raise ValueTooBigError(num). W wyniku takich działań, otrzymalibyśmy jeszcze bardziej spersonalizowany komunikat wyjątku:



A to wszystko za sprawą utworzenia pola **self.value** w klasie **ValueTooBigError**. Klasa reprezentująca wyjątek może zachowywać się jak normalny obiekt, przechowywać własne pola i metoda. W momencie potrzeby manipulowania wyjątkami, może się okazać to bardzo przydatne.