Errors And Exception

Types of Errors

- Until now error messages haven’t been more than mentioned, but you have probably seen some
- Two kinds of errors (in Python):
  1. Syntax errors
  2. Exceptions

Syntax Errors

```
>>> while True print('Hello world')
SyntaxError: invalid syntax
```

Exceptions

- Errors detected during execution are called exceptions
- Examples:
  - ZeroDivisonError,
  - NameError,
  - TypeError

ZeroDivisionError

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    10 * (1/0)
ZeroDivisionError: division by zero
```

NameError

```
>>> 4 + spam*3
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    4 + spam*3
NameError: name 'spam' is not defined
```

TypeError

```
>>> '2' + 2
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    '2' + 2
TypeError: Can't convert 'int' object to str
implicitly
```

ValueError

```
>>> int('one')
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    int('one')
ValueError: invalid literal for int() with base
10: 'one'
```

Handling Exceptions(Errors)

Handling Exceptions

- The simplest way to catch and handle exceptions is with a try-except block:

```
x, y = 5, 0
try:
    z = x/y
except ZeroDivisionError:
    print("divide by zero")
```

How it works

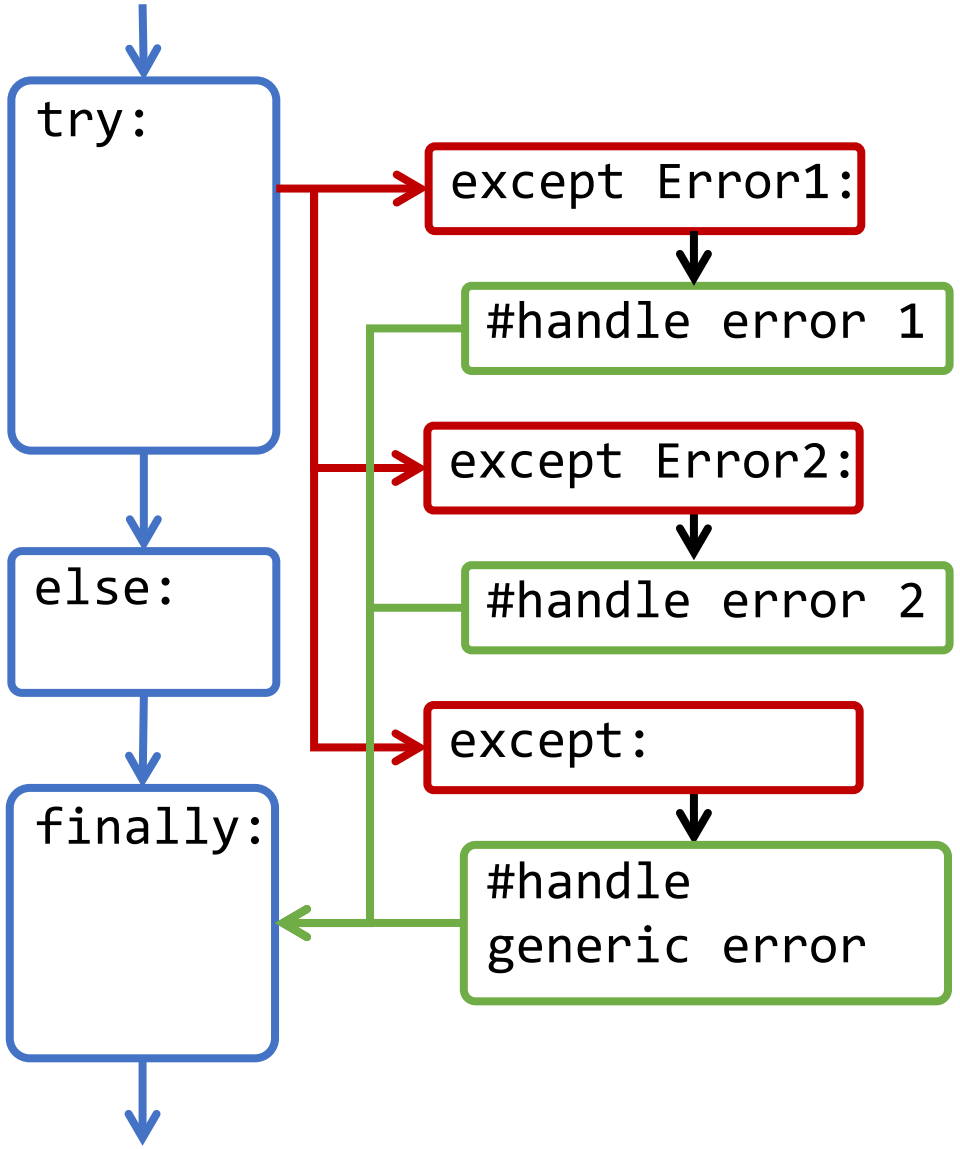
- The `try` clause is executed
- If an exception occurred, skip the rest of the `try` clause, to a matching `except` clause
- If no exception occurs, the `except` clause is skipped (go to the else clause, if it exists)
- The `finally` clause is always executed before leaving the `try` statement, whether an exception has occurred or not.

Try-Except

- A `try` clause may have more than 1 `except` clause, to specify handlers for different exception.
- At most one handler will be executed.
- Similar with `if-elif-else`
- `finally` will always be executed

Try-Except

```
try:
    # statements
except Error1:
    # handle error 1
except Error2:
    # handle error 2
except: # wildcard
    # handle generic error
else:
    # no error raised
finally:
    # always executed
```



Try-Except Example

```
def divide_test(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
```

Try-Except Blocks

```
>>> divide_test(2, 1)
result is 2.0
executing finally clause

>>> divide_test(2, 0)
division by zero!
executing finally clause

>>> divide_test("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

```
def divide_test(x, y):
    try:
      result = x / y
    except ZeroDivisionError:
      print("division by zero!")
    else:
      print("result is", result)
    finally:
      print("executing finally
            clause")
```

Raising Exceptions

- The `raise` statement allows the programmer to force a specific exception to occur:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

Exception Types

- Built-in Exceptions: https://docs.python.org/3/library/exceptions.html
- User-defined Exceptions

Example: How to use exceptions

- Remember our tic-tac-toe?
- We would like the user to input the number from 1 to 9
  - We assume that the user is good to enter it obediently
- But not the real life situation in life
  - There is always mistake or naughty users

```
1|2|3
-----
4|5|6
-----
7|8|9

Player X move:what
Traceback (most recen
  File "/Volumes/Goog
Arrays/TTT.py", line
    tttGamePlay()
  File "/Volumes/Goog
Arrays/TTT.py", line
    pos = int(input(f
ValueError: invalid l
```

How to make sure your user input is a number?

- Original code:

```
pos = int(input(f'Player {piece[player]} move:')) - 1
```

- You can do a lot of checking, e.g.

```
userinput = input(f'Player {piece[player]} move:')
if userinput.isnumeric():
    #play as normal
else:
    #error and input again
```

- However, it requires:
  - You can consider ALL wrong situations
  - And you can check them all out with codes

Example:

```
while True:
    try:
        pos = int(input("Input:"))
        break
    except:
        print("Wrong")
```

- If the user input an integer
  - Nothing wrong
  - break, exit the while loop
- Otherwise, go to “except:”
  - Hence, will not break the while loop

User-defined Exceptions I

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)
```

User-defined Exceptions II

```
try:
    raise MyError(2*2)
except MyError as e:
    print('Exception value:', e.value)
Exception value: 4

raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

Assertion

- For example, in tic-tac-toe, you also assume the position is from 1 to 9
- For a lot of situations, you ”assume” certain conditions in your code, e.g.
  - A sorting function will only take sequences as input
  - A function checking prime number will only take in integers
  - In a certain part of your code, you expect some index i will not exceed a certain range
- In Python, you can simply add an assertion
  - If the statement following in the assertion is False, then EXCEPTIONS!

Example

- Assert that the pos must be within range

```
while True:
    try:
        pos = int(input("Input:"))
        assert 0 < pos < 10
        break
    except:
        print("Wrong")
```

Example

- In order to catch the particular exception of the assertion, we can

```
while True:
    try:
        pos = int(input("Input:"))
        assert 0 < pos < 10
        break
    except AssertionError:
        print("Your number is not in the range")
    except:
        print("Wrong")
```

Why use Exceptions?

- In the good old days of C, many procedures returned special ints for special conditions, i.e. -1

Why use Exceptions?

- But Exceptions are better because:
  - More natural
  - More easily extensible
  - Nested Exceptions for flexibility