Errors And Exception

Random Error just to annoy you.

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:

Type of Exception	Description
NameError	If an identifier is used before assignment
TypeError	If wrong type of parameter is sent to a function
ValueError	If function parameter has invalid value (Eg: log(-1))
ZeroDivisionError	If 0 is used as divisor
StopIteration	Raised by next(iter)
IndexError	If index is out of bound for a sequence
KeyError	If non-existent key is requested for set or dictionary
IOError	If I/O operation fails (eg: opening a file)
EOFError	If end of file is reached for console of file input
AttributeError	If an undefined attribute of an object is used

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

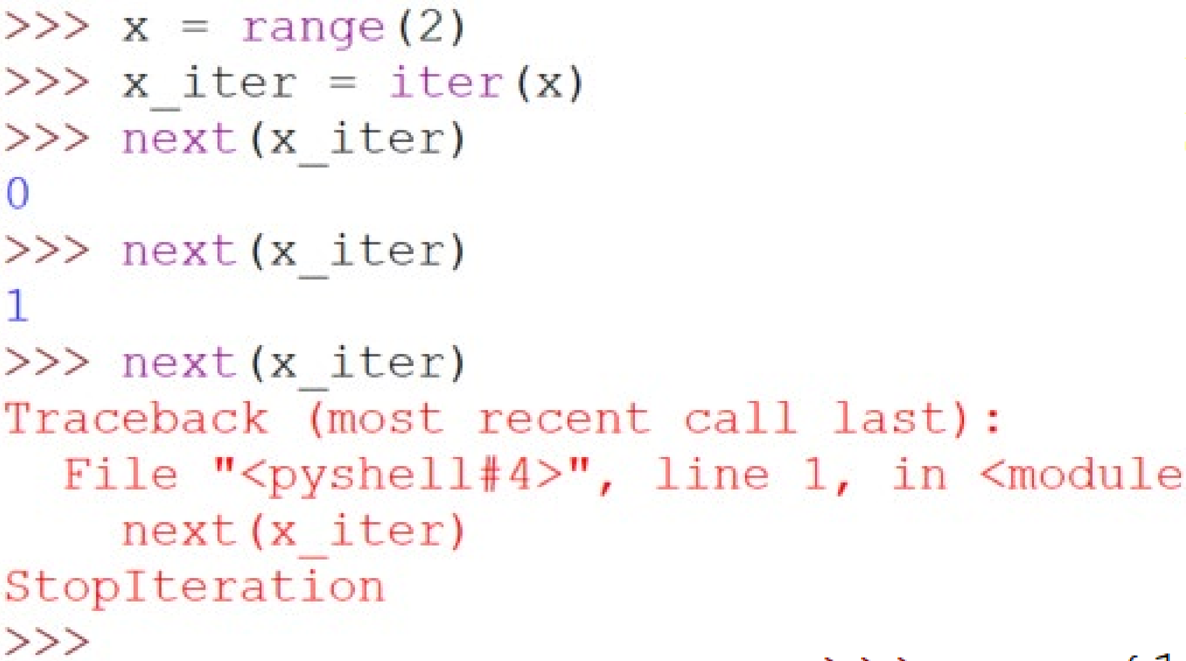
```
>>> int('one')
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    int('one')
ValueError: invalid literal for int() with base
10: 'one'
```

ZeroDivisionError

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

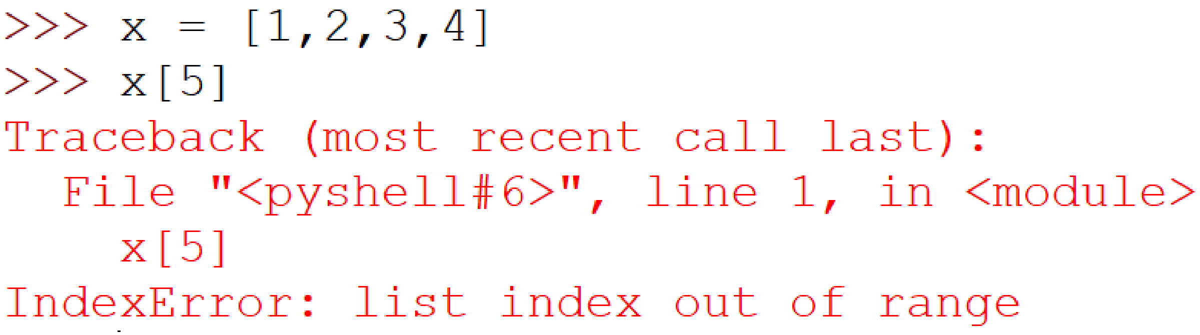
Other Common Errors

StopIteration Error



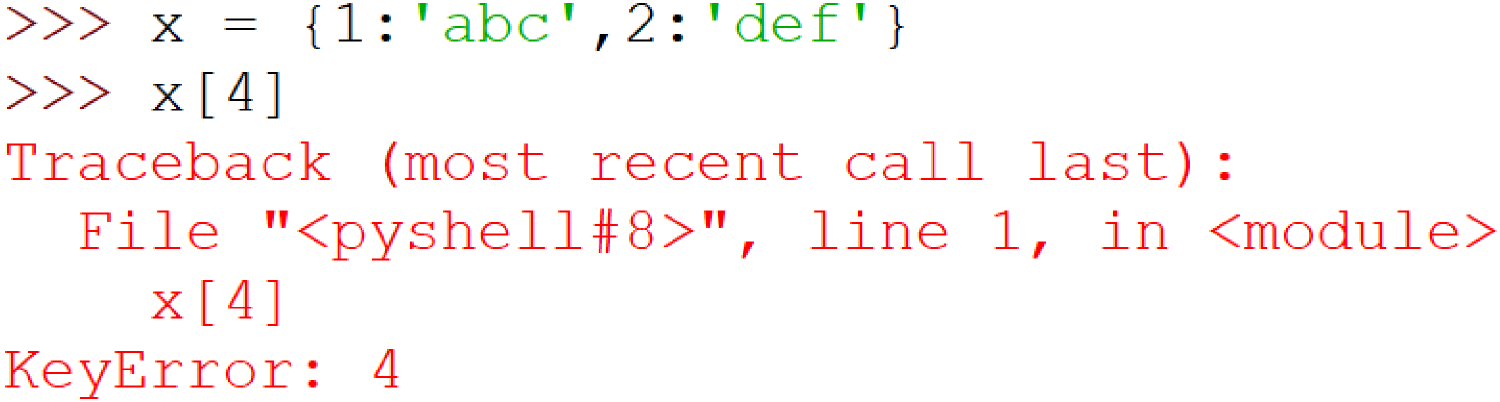
```
>>> x = range(2)
>>> x_iter = iter(x)
>>> next(x_iter)
0
>>> next(x_iter)
1
>>> next(x_iter)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    next(x_iter)
StopIteration
>>>
```

IndexError



```
>>> x = [1,2,3,4]
>>> x[5]
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    x[5]
IndexError: list index out of range
```

KeyError



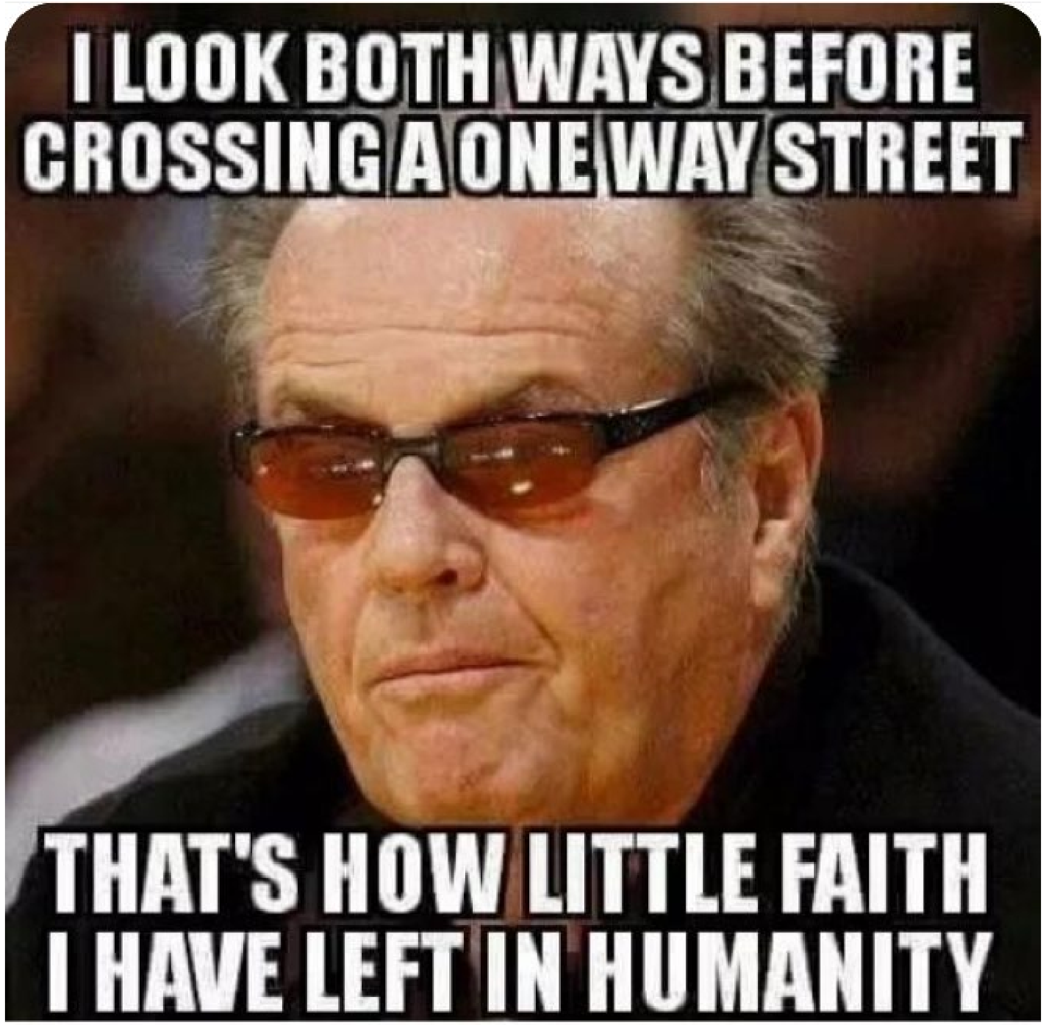
```
>>> x = {1:'abc',2:'def'}
>>> x[4]
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    x[4]
KeyError: 4
```

Handling Exceptions (Errors)

MOM, CAN I TAKE UP THE FLOORBOARDS IN MY ROOM AND MAKE A SECRET PASSAGEWAY?
OF COURSE NOT, CALVIN. DON'T BE RIDICULOUS.
WHY CAN'T I?
BECAUSE YOU'D COME RIGHT THROUGH THE KITCHEN CEILING. I SAID DON'T DO IT.
OKAY, OKAY...
HOW QUIETLY DO YOU THINK WE CAN NAIL THESE BACK IN?
© 1986 Universal Press Syndicate
WATTERSON
1-21

Handling Exceptions

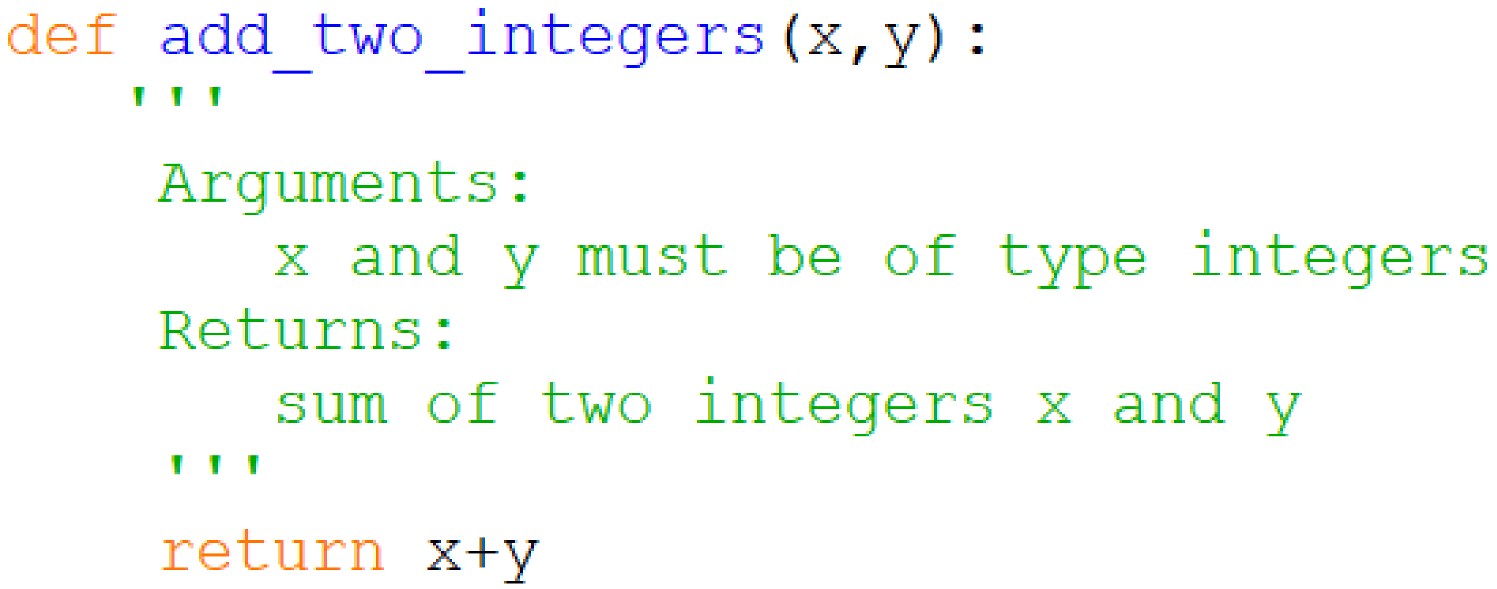
- Two Approaches
  - Using Guard Clauses
  - Using Try-Except-Else constructs


I LOOK BOTH WAYS BEFORE CROSSING A ONE WAY STREET
THAT'S HOW LITTLE FAITH I HAVE LEFT IN HUMANITY

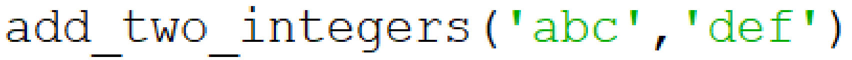
Guard Clauses

Guard is a Boolean expression that must evaluate to *True* if the program execution is to continue in the branch in question

Raising Exceptions



```
def add_two_integers(x,y):
   '''
    Arguments:
       x and y must be of type integers
    Returns:
       sum of two integers x and y
    '''
    return x+y
```



```
add_two_integers('abc','def')
```

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
```

Raising Exceptions

```
def add_two_integers_1(x,y):
    '''
   Arguments:
       x and y must be of type integers
   Returns:
       sum of two integers (or floats) x and y
    '''
    if not isinstance(x,int):
        raise TypeError('First argument must be of type integer')
    if not isinstance(y,int):
        raise TypeError('Second argument must be of type integer')
    return x+y


z = add_two_integers_1('abc','def')
```

raise terminates the function and shows the message

```
    raise TypeError('First argument must be of type integer')
TypeError: First argument must be of type integer
```

Raising Exceptions

checking for multiple types

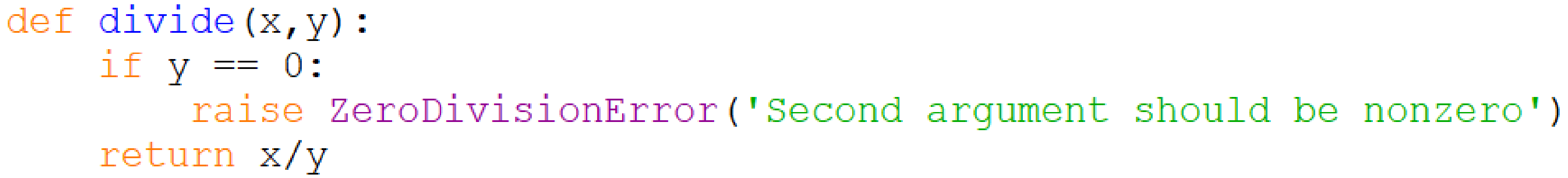
```
def add_two_numbers(x,y):
    '''
   Arguments:
       x and y must be of either type integer or float
   Returns:
       sum of two integers (or floats) x and y
    '''
    if not isinstance(x,(int,float)):
        raise TypeError('Only numerics are allowed for first argument')
    if not isinstance(y,(int,float)):
        raise TypeError('Only numerics are allowed for second argument')
    return x+y
```

```
add_two_numbers(2.0,1)
```

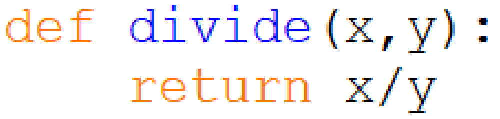
```
add_two_numbers('abc','def')
```

Guard Clauses: Use with Caution

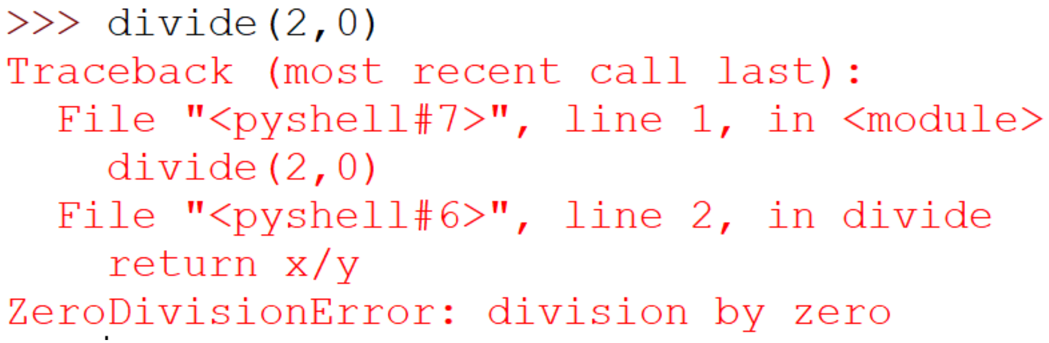
- Python can raise exceptions without explicit guard clauses
- Checking for a specific exception may consume resources (eg: time)
  - Especially if it is done within a loop with several iterations



```
def divide(x,y):
    if y == 0:
        raise ZeroDivisionError('Second argument should be nonzero')
    return x/y
```

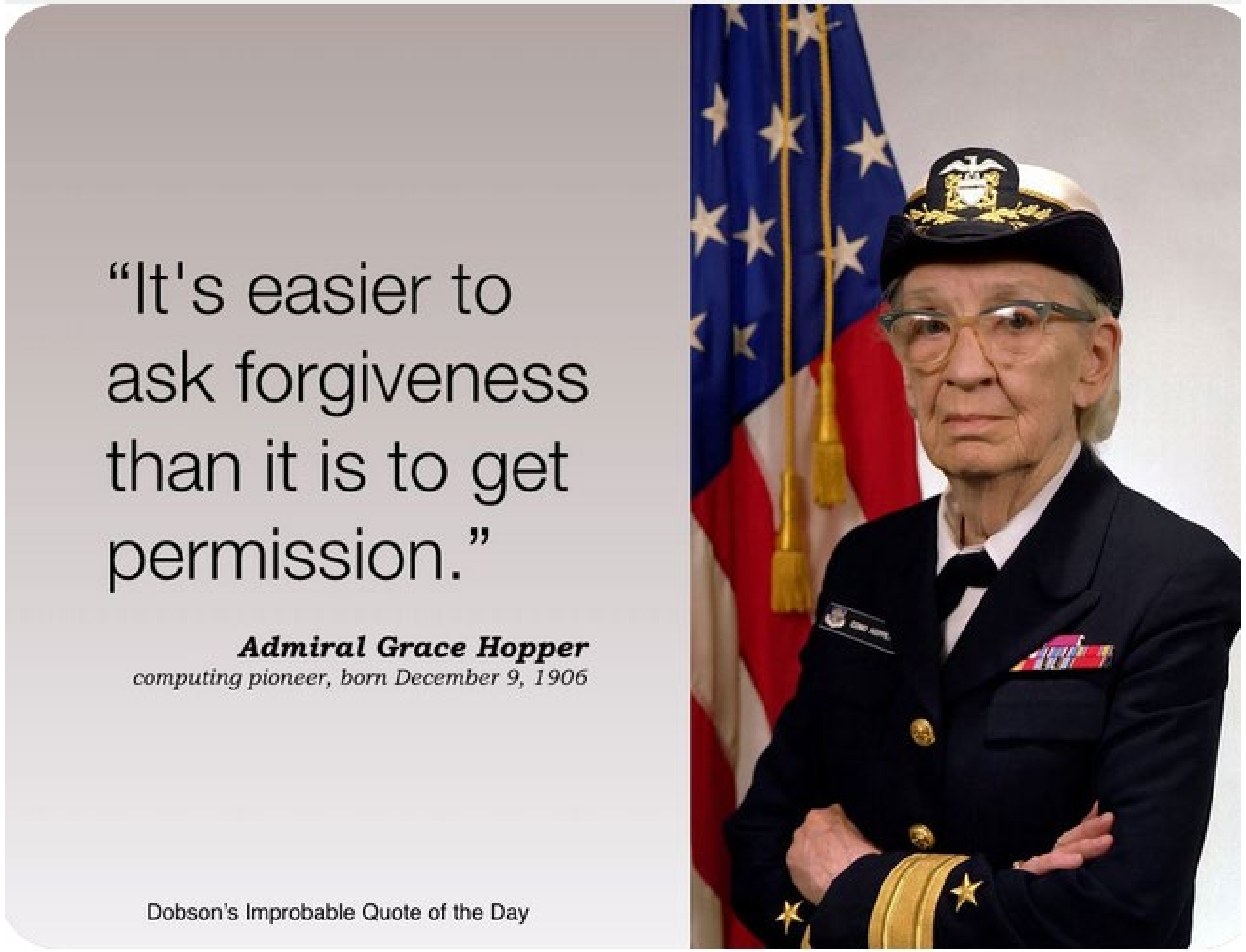


```
def divide(x,y):
    return x/y
```



```
>>> divide(2,0)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    divide(2,0)
  File "<pyshell#6>", line 2, in divide
    return x/y
ZeroDivisionError: division by zero
```

Handling Exceptions



“It's easier to ask forgiveness than it is to get permission.”

Admiral Grace Hopper
computing pioneer, born December 9, 1906

Dobson’s Improbable Quote of the Day

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

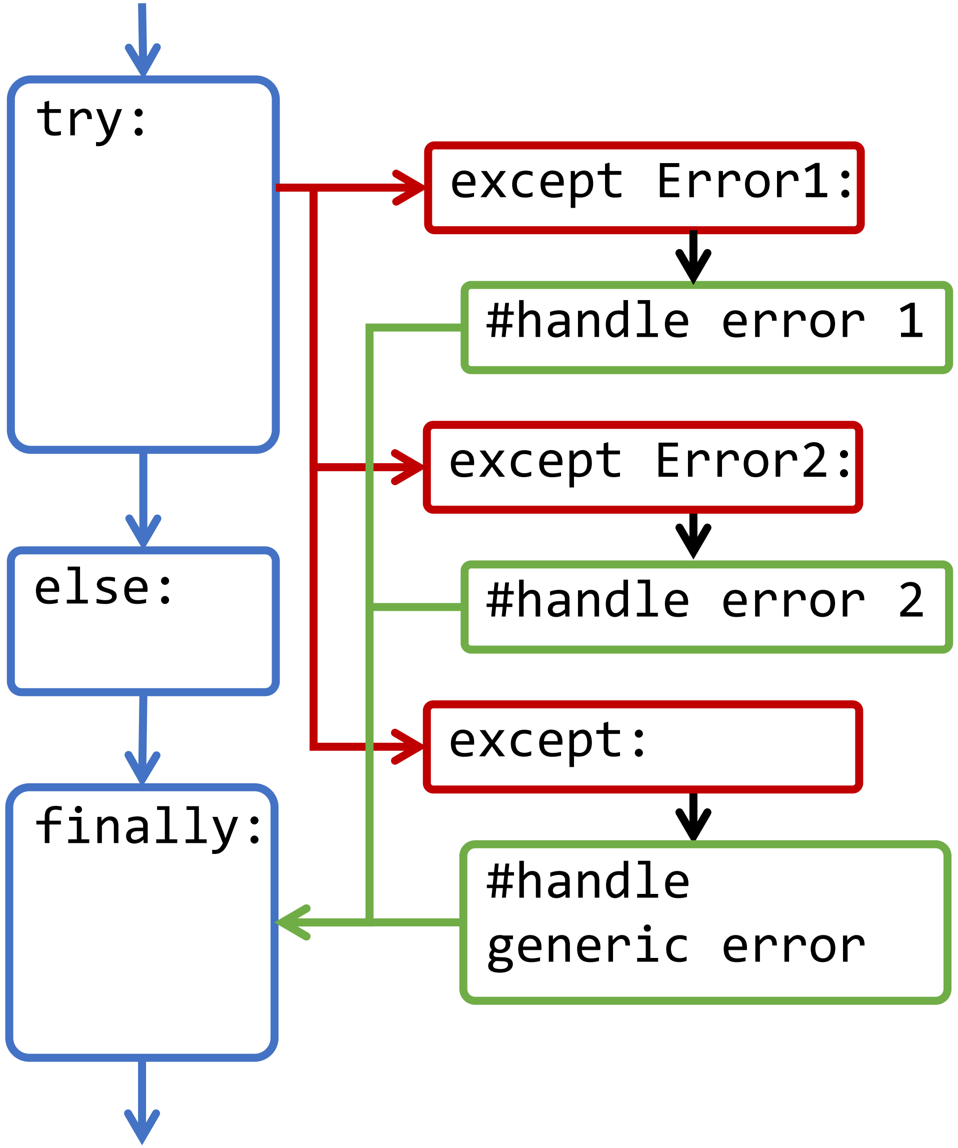
- The `try` clause is executed
- If an exception occurred, skip the rest of the `try` clause, to a matching `except` clause
- If no exception occurs, the `except` clause is skipped (go to the else clause, if it exists)
- The `finally` clause is always executed before leaving the `try` statement, whether 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 to `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

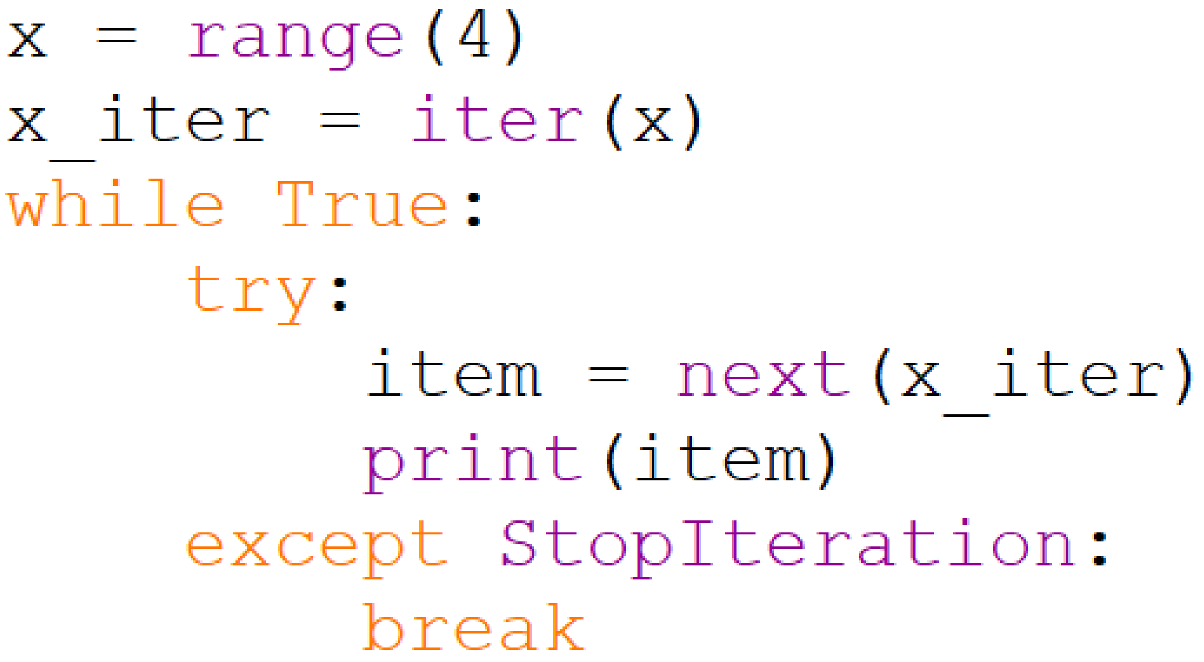
```
def divide_test(x, y):
    try:
      result = x / y
    except ZeroDivisionError:
      print("division by zero!")
    else:
      print("result is", result)
    finally:
      print("executing finally clause")
```

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")
```

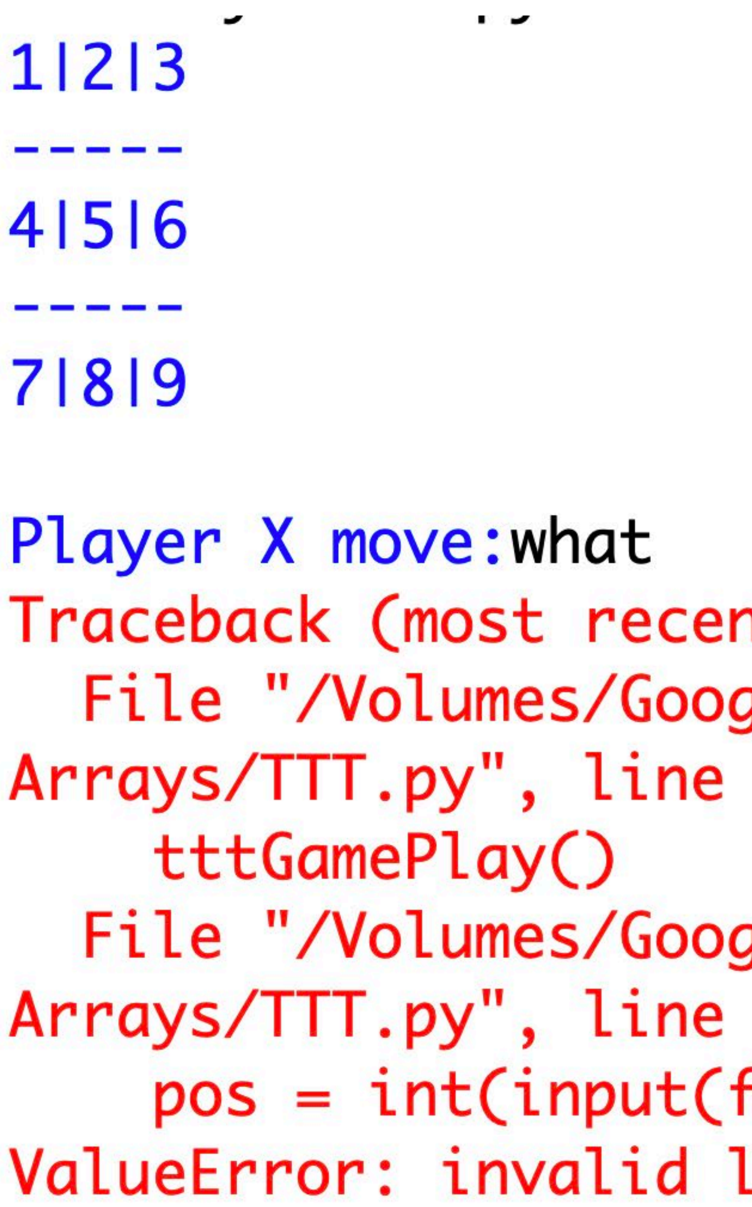
Simulation of *for* loop using *while* Loop



```
x = range(4)
x_iter = iter(x)
while True:
    try:
        item = next(x_iter)
        print(item)
    except StopIteration:
        break
```

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:

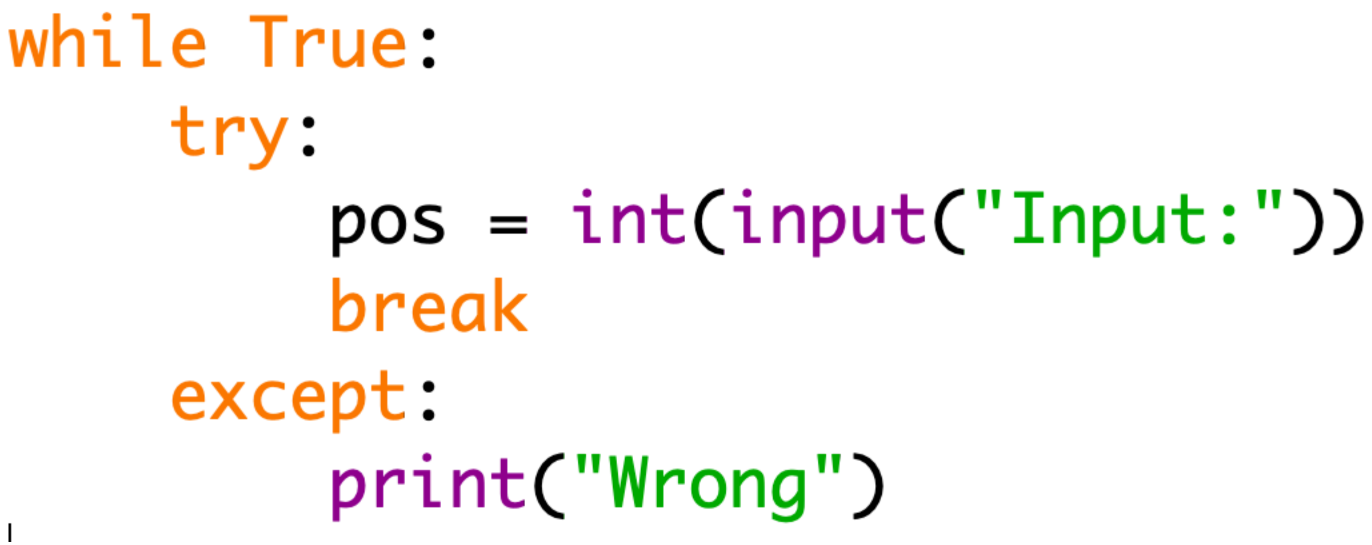
```
pos = int(input(f'Player {piece[player]} move:')) - 1
```

- 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:

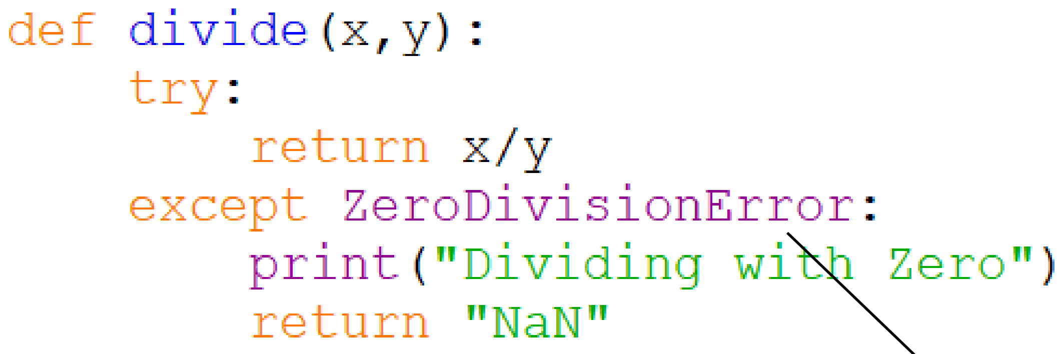


```python
while True:
    try:
        pos = int(input("Input:"))
        break
    except:
        print("Wrong")
```

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

Try-Except: Checking for Single Exception

If exceptions occur rarely, Try-Except is better than guard clauses



```python
def divide(x,y):
    try:
        return x/y
    except ZeroDivisionError:
        print("Dividing with Zero")
        return "NaN"
```

We can check multiple exceptions

Try-Except: Multiple Exceptions

Check each exception and provide specific message

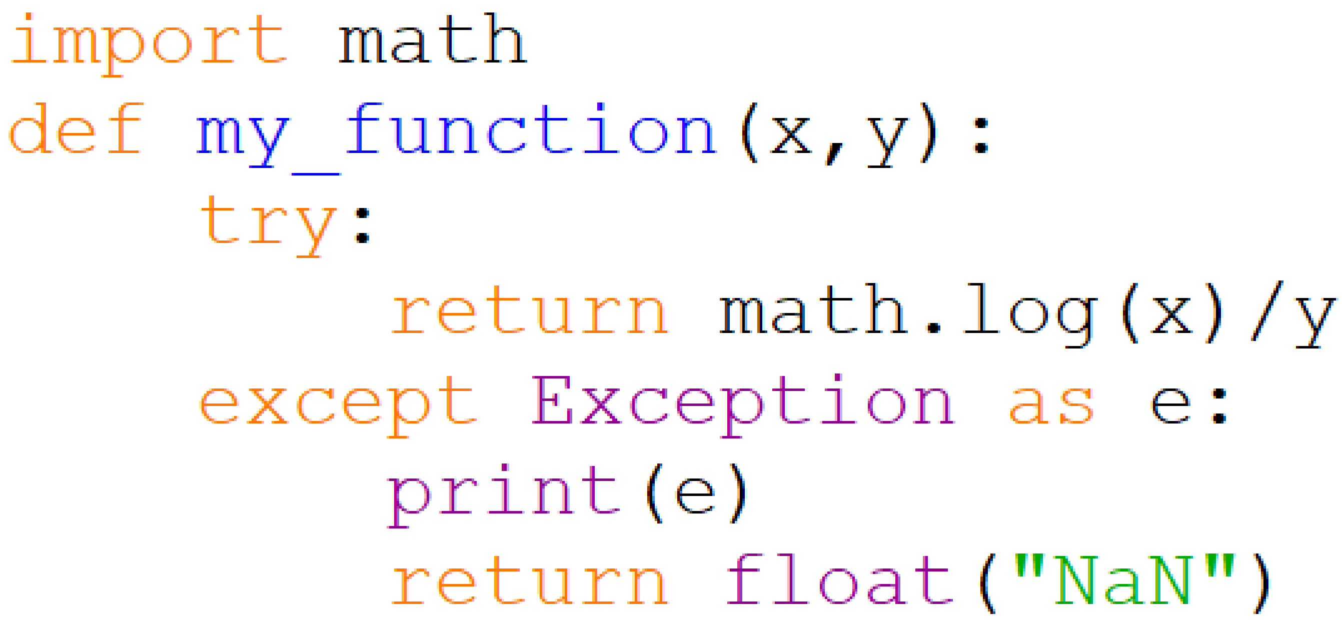
```
import math
def my_function(x,y):
    try:
        return math.log(x)/y
    except ValueError:
        print('First argument must be nonzero positive; returning nan')
        return float("NaN")
    except ZeroDivisionError:
        print('Second argument must be nonzero; returning nan')
        return float("NaN")
```

Try-Except: Multiple Exceptions

Check for multiple exceptions simultaneously

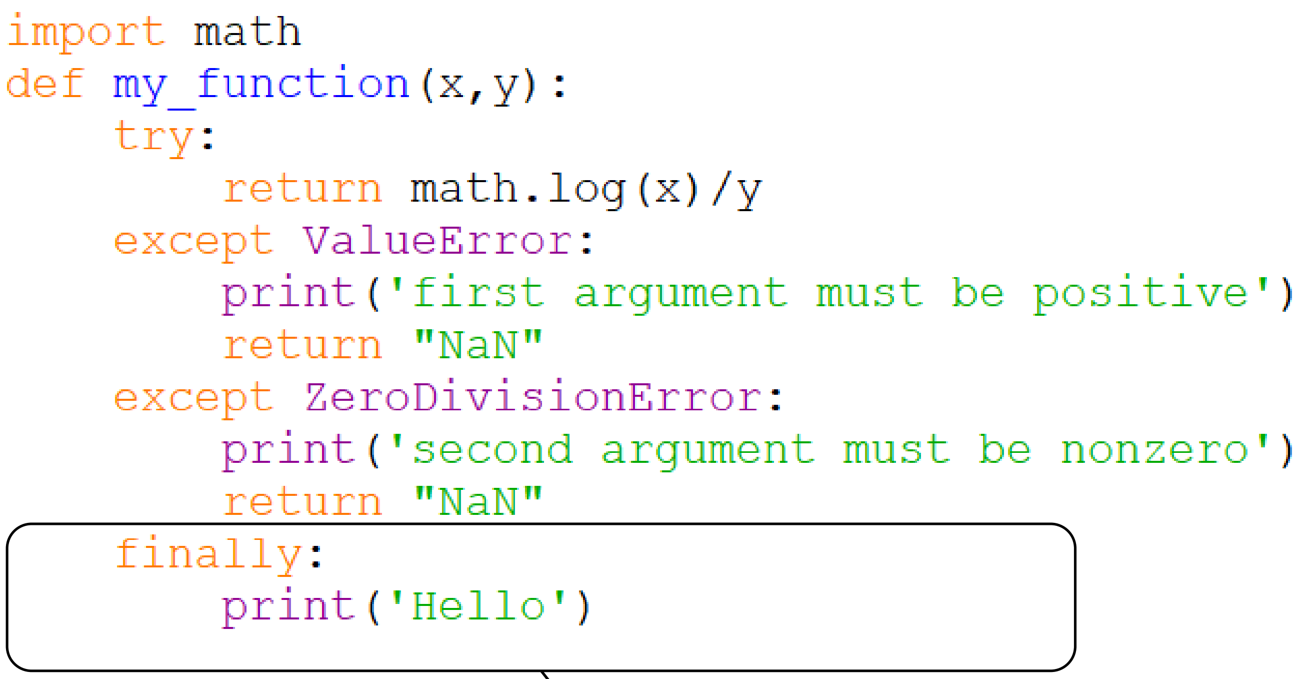
```
import math
def my_function_1(x,y):
    try:
        return math.log(x)/y
    except (ZeroDivisionError,ValueError):
        print('First argument must be nonzero positve')
        print('Second argument must be nonzero')
        print('Returning nan')
        return float("NaN")
```

Checking for all exceptions



```
import math
def my_function(x,y):
    try:
        return math.log(x)/y
    except Exception as e:
        print(e)
        return float("NaN")
```

Try-Except-Finally



```
import math
def my_function(x,y):
    try:
        return math.log(x)/y
    except ValueError:
        print('first argument must be positive')
        return "NaN"
    except ZeroDivisionError:
        print('second argument must be nonzero')
        return "NaN"
    finally:
        print('Hello')
```

This block is always executed

Exception Types

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

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

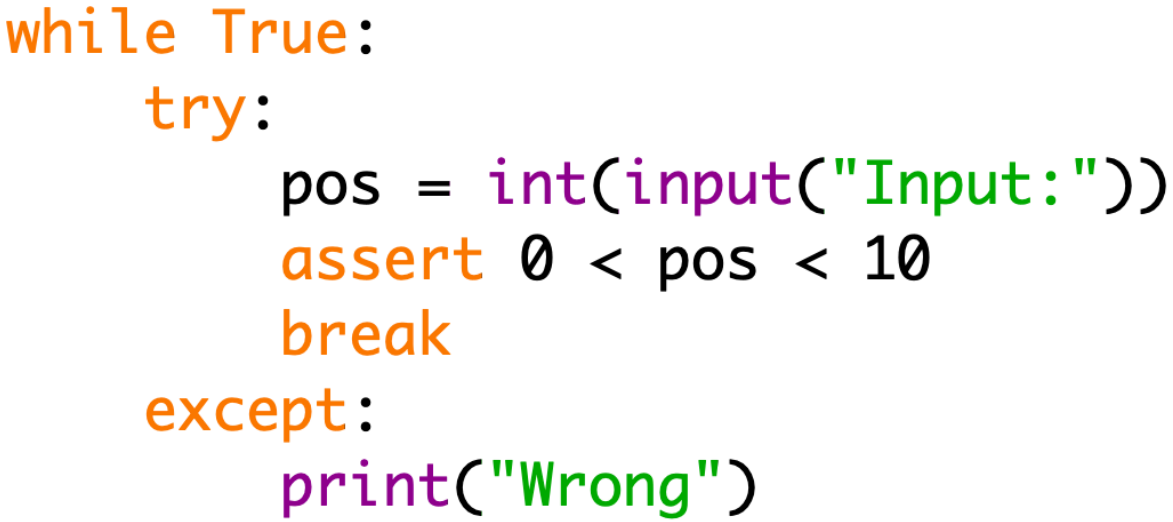
```
try:
    raise MyError(2*2)
except MyError as e:
    print('Exception value:', e.value)
Exception value: 4

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!
    - Raises an AsserionError

Example

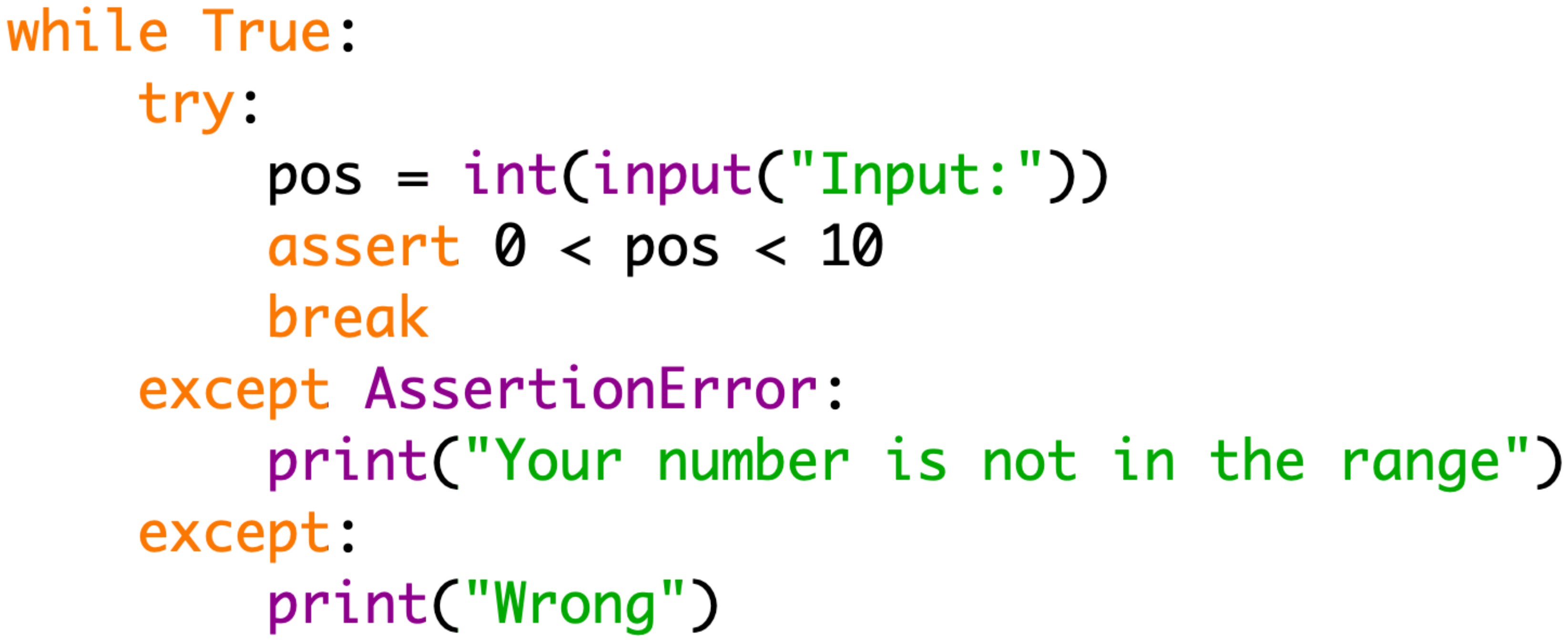
- Assert that the pos must be within range



```
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