Object-Oriented Programming (Python)

Exception Handling and Files

Dr. Bianca Schoen-Phelan
bianca.phelan@tudublin.ie
www.biancaphelan.ie
@BSPhelan
KE-3-005a

Objectives

- Discuss the need for exceptions
- Apply and use exceptions
- Write your own exceptions
- Interact with files

Why we need Exceptions

- In the old days we would rigorously check inputs for all possible values and try to avoid bad inputs/outputs
- Methods would have special return values to indicate an error had occurred
    - Example: return a negative value if a positive value could not be calculates
    - Different numbers indicate different errors
- Every code that is using this method would have to explicitly check for these error numbers
    - This is tedious so it rarely happened, so programs just crashed

What are Exceptions

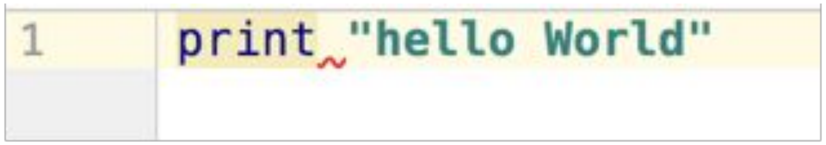
- Allow us to deal with errors only if they come up
- They handle recovery, so programs crash less
- Are used as part of your code’s flow control

How to Raise and Exception

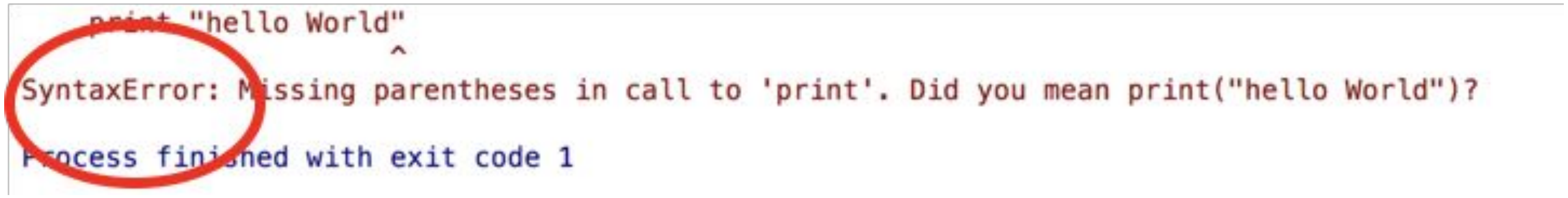
- It’s just an object
  - Like everything in Python!
- Many different classes are available to raise exceptions
- They all inherit from a built in class called BaseException

If an exception occurs, everything that was supposed to happen in your program, now doesn’t happen.

Common Exceptions



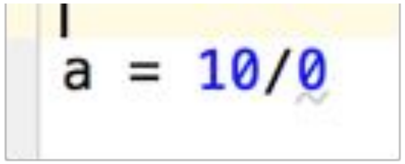
```
print "hello World"
```



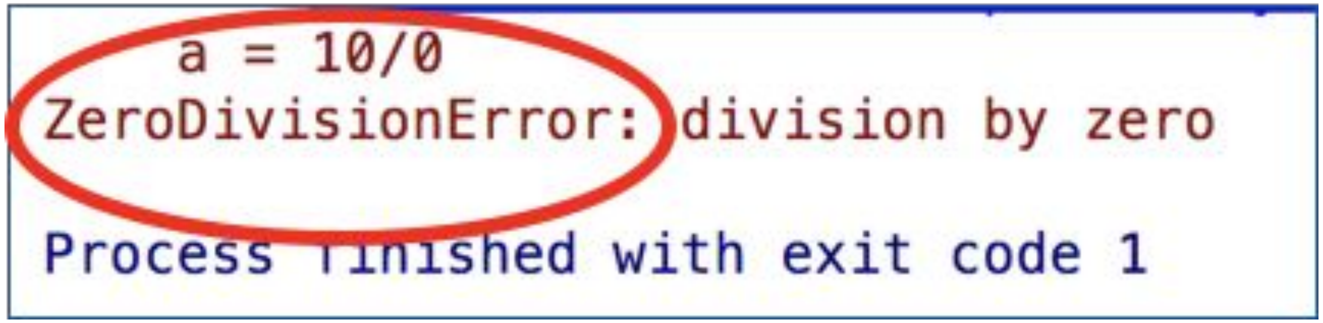
```
    print "hello World"
                      ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("hello World")?

Process finished with exit code 1
```

This particular error happens a lot for programmers switching from Python2 to Python3, because print, although a valid instruction, is now a method!

Common Exceptions cont’d

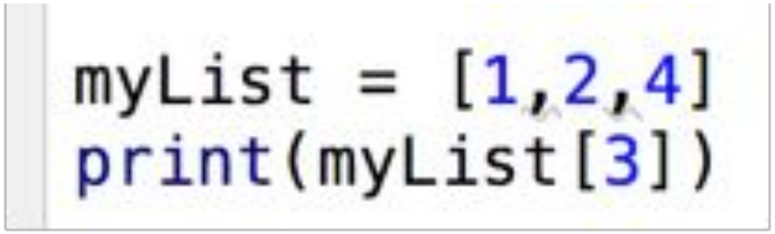


```
a = 10/0
```

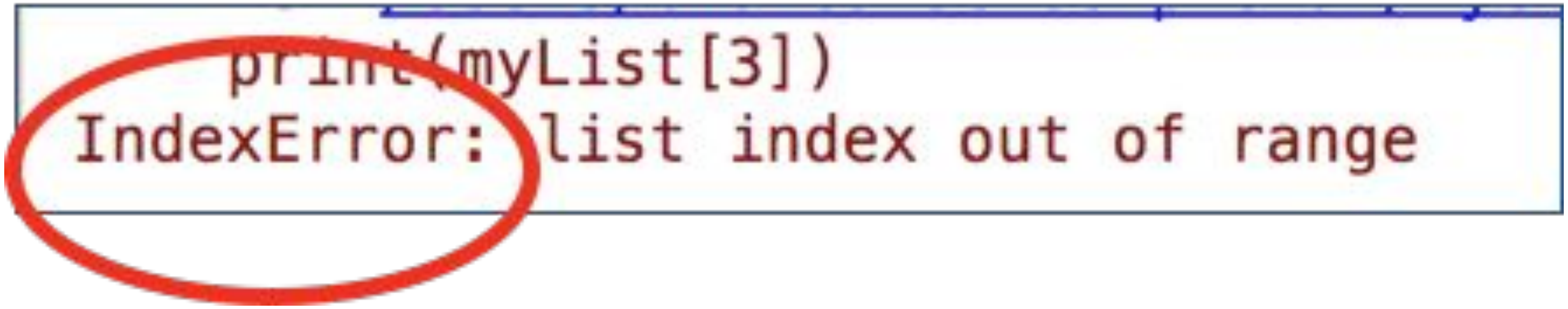


```
    a = 10/0
ZeroDivisionError: division by zero

Process finished with exit code 1
```

Common Exceptions cont'd

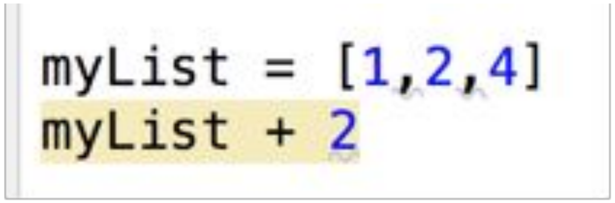


```
myList = [1,2,4]
print(myList[3])
```

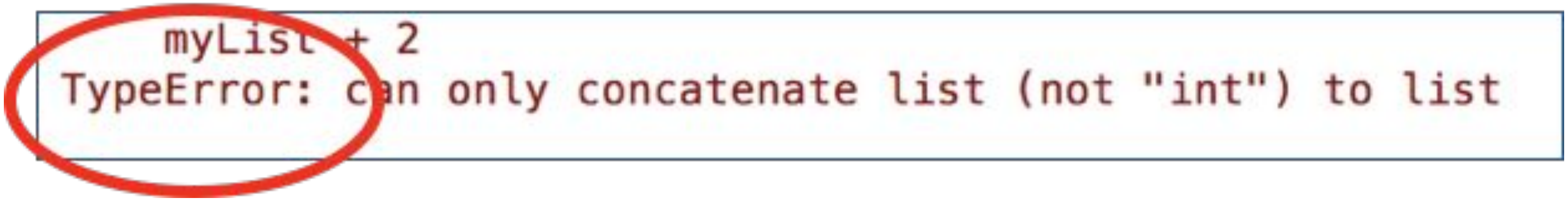


```
    print(myList[3])
IndexError: list index out of range
```

Common Exceptions cont'd

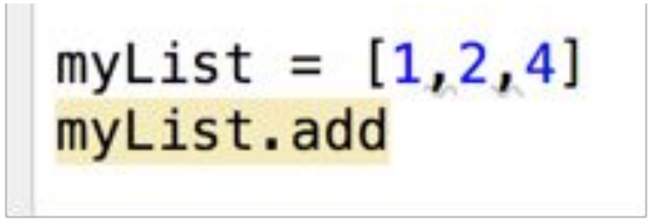


```
myList = [1,2,4]
myList + 2
```

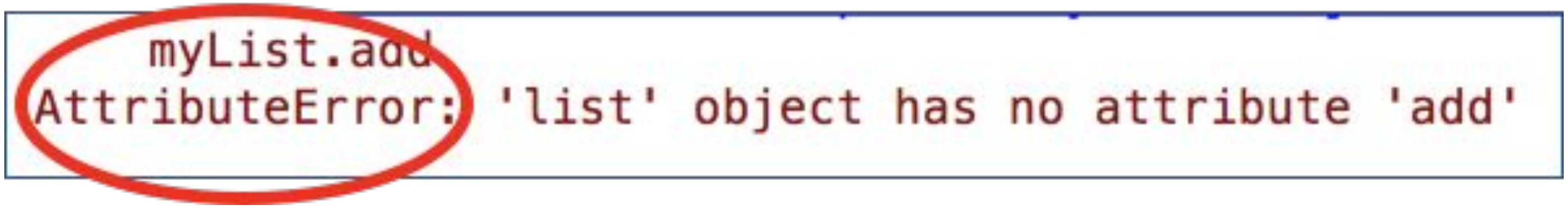


```
    myList + 2
TypeError: can only concatenate list (not "int") to list
```

Common Exceptions cont’d

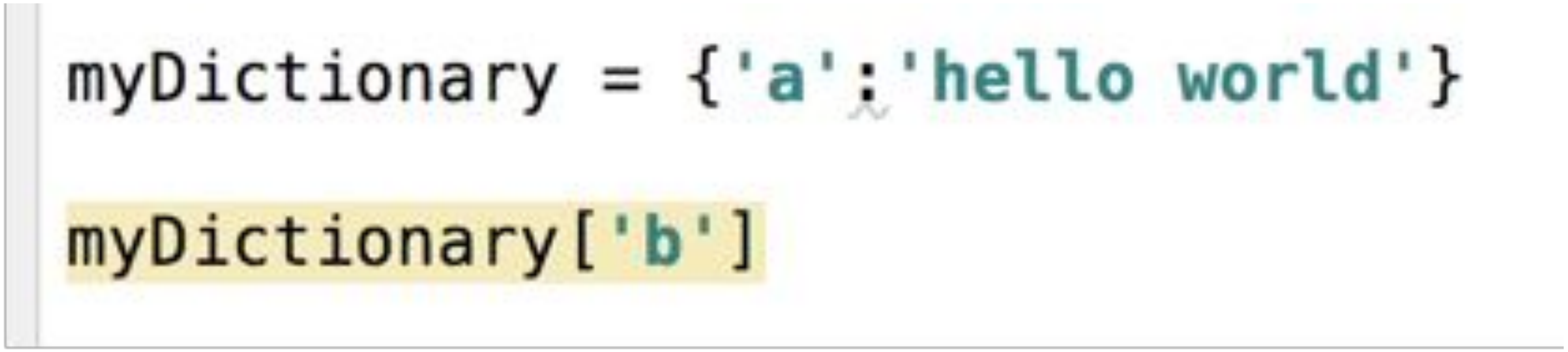


```
myList = [1,2,4]
myList.add
```

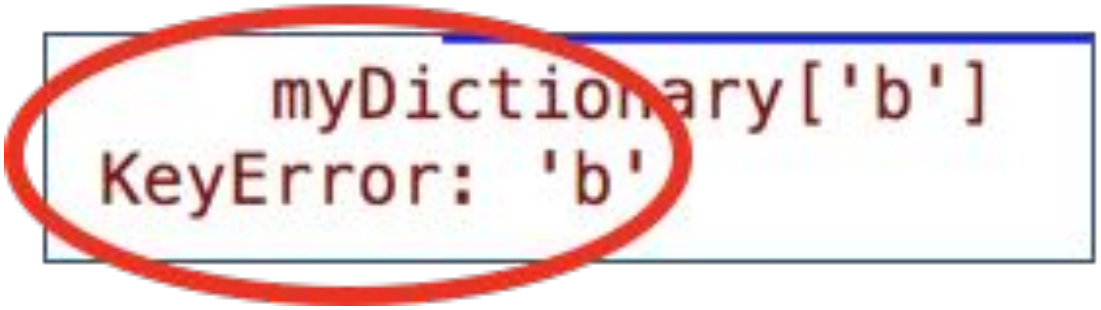


```
    myList.add
AttributeError: 'list' object has no attribute 'add'
```

Common Exceptions cont'd

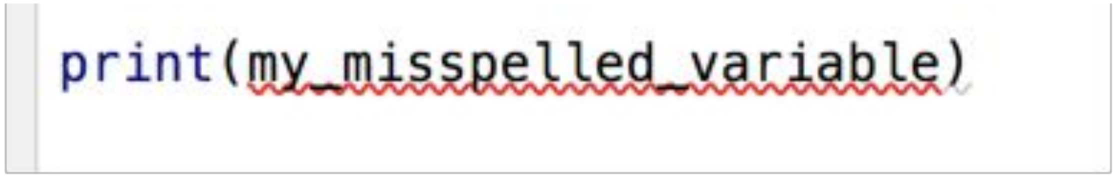


```
myDictionary = {'a':'hello world'}

myDictionary['b']
```

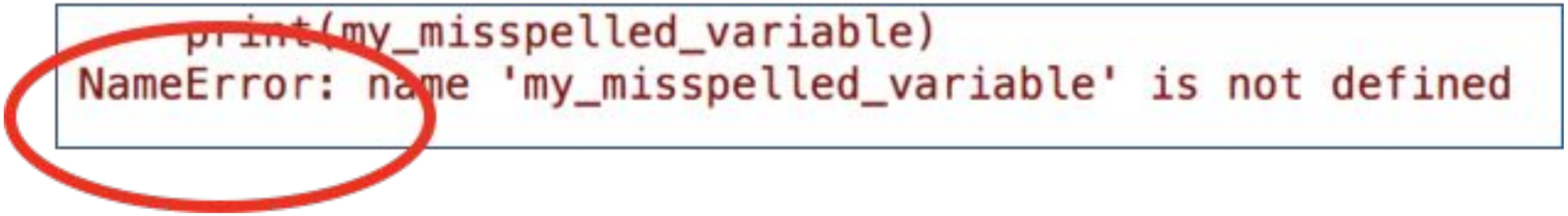


```
    myDictionary['b']
KeyError: 'b'
```

Common Exceptions cont'd



```
print(my_misspelled_variable)
```



```
      print(my_misspelled_variable)
NameError: name 'my_misspelled_variable' is not defined
```

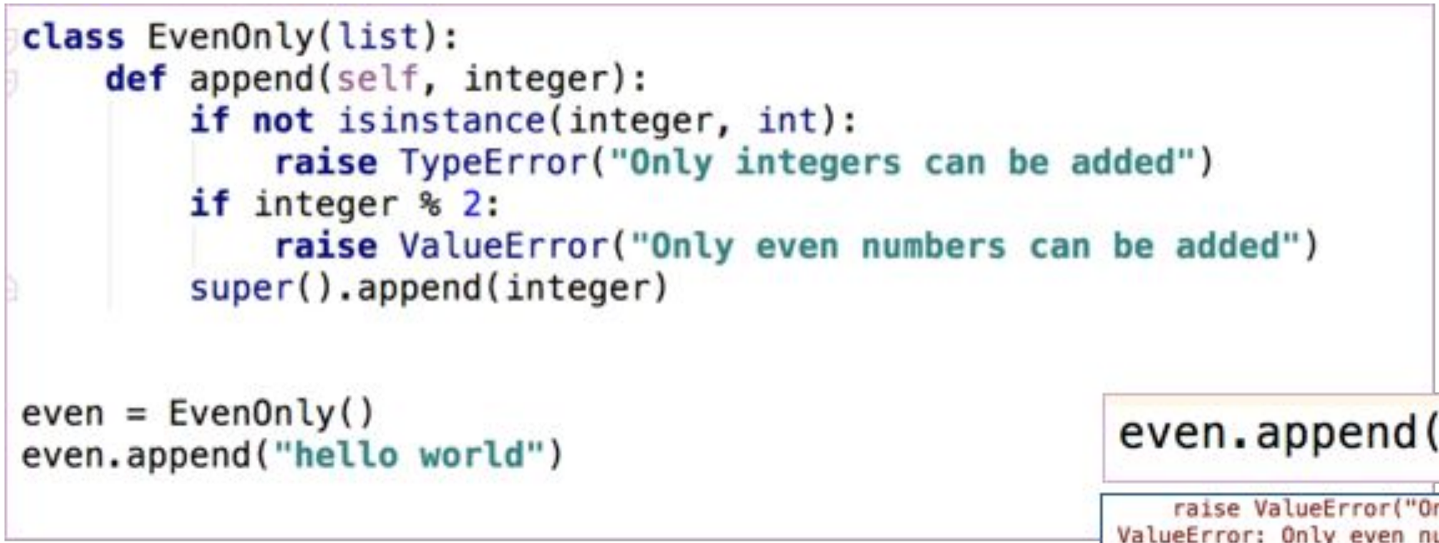
Common Exceptions Summary

- Not always ‘wrong’ input as such, sometimes intentional, for example zero division error
- All exceptions end on Error
- Error also uses Exception’s base class to inherit from

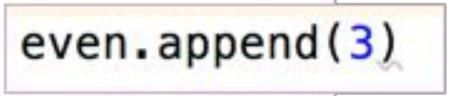
You can use the same mechanisms Python uses to check for inputs.

In Python we use the words error and exceptions mostly interchangeably.

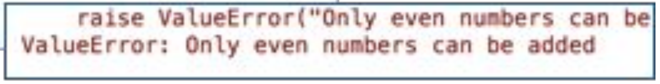
Use in your own code



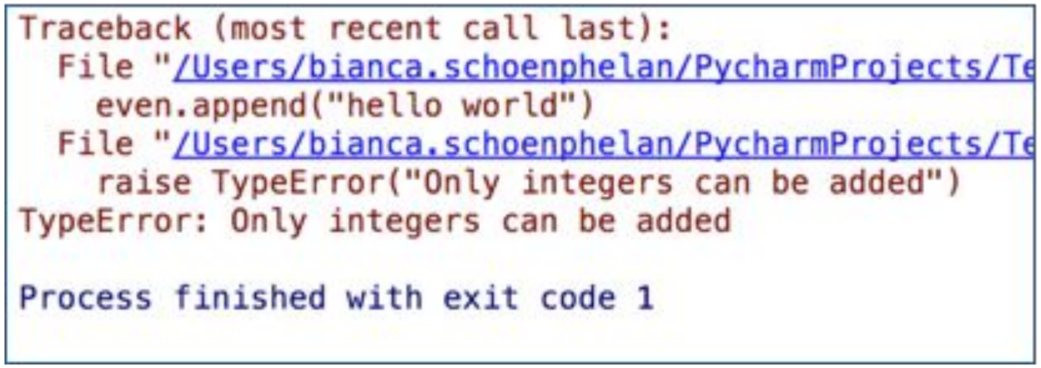
```python
class EvenOnly(list):
    def append(self, integer):
        if not isinstance(integer, int):
            raise TypeError("Only integers can be added")
        if integer % 2:
            raise ValueError("Only even numbers can be added")
        super().append(integer)


even = EvenOnly()
even.append("hello world")
```



```python
even.append(3)
```



```
    raise ValueError("Only even numbers can be
ValueError: Only even numbers can be added
```



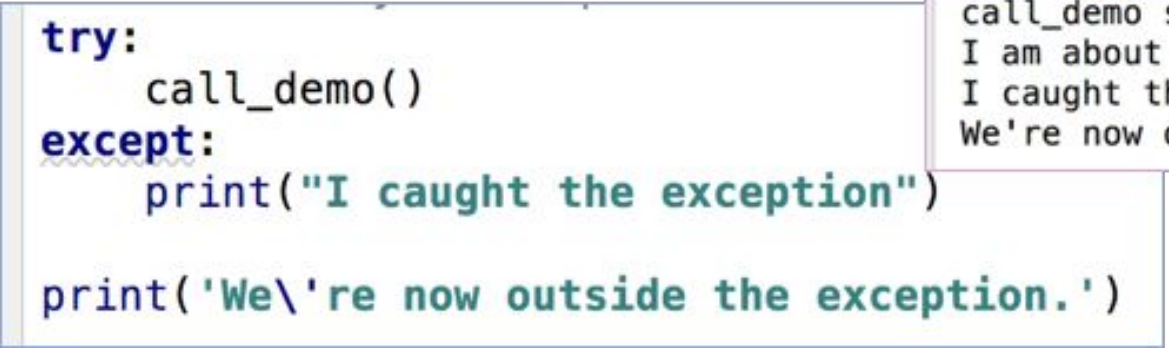
```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/PycharmProjects/Te
    even.append("hello world")
  File "/Users/bianca.schoenphelan/PycharmProjects/Te
    raise TypeError("Only integers can be added")
TypeError: Only integers can be added

Process finished with exit code 1
```

Exception Handling

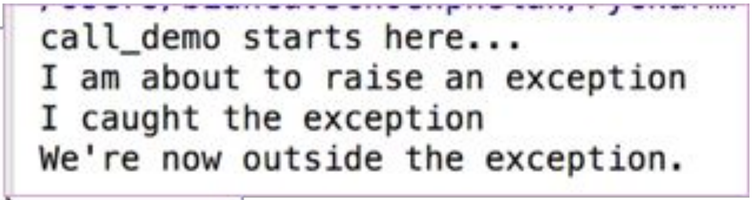
- When an exception occurs the program appears to stop immediately
- None of the code after the exception is executed
- Unless the exception is dealt with, the program will terminate with an error message
- Even if your method calls another method that raises an exception, nothing beyond the 2nd method call will be executed once the exception has been raised.

Exceptions

- What would happen in case of an exception?



```
try:
    call_demo()
except:
    print("I caught the exception")

print('We\'re now outside the exception.')
```

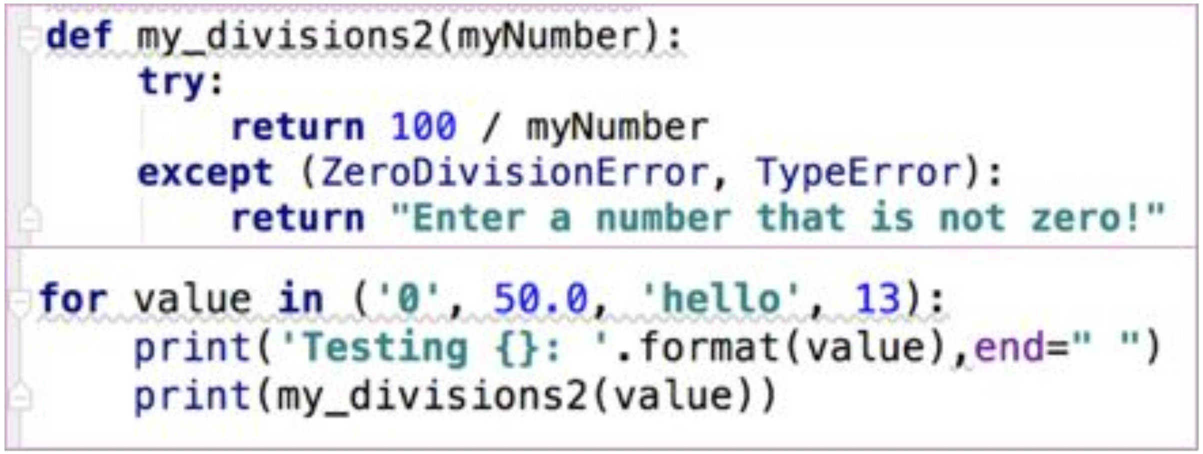


```
call_demo starts here...
I am about to raise an exception
I caught the exception
We're now outside the exception.
```

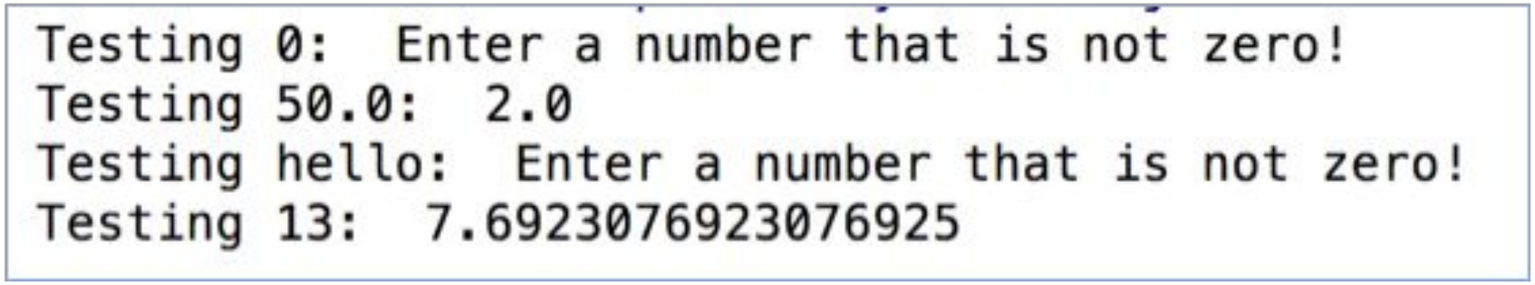
- Recover and continue!!

Handling different Scenarios

We can catch several different exceptions and handle them with the same code.



```
def my_divisions2(myNumber):
    try:
        return 100 / myNumber
    except (ZeroDivisionError, TypeError):
        return "Enter a number that is not zero!"

for value in ('0', 50.0, 'hello', 13):
    print('Testing {}: '.format(value),end=" ")
    print(my_divisions2(value))
```



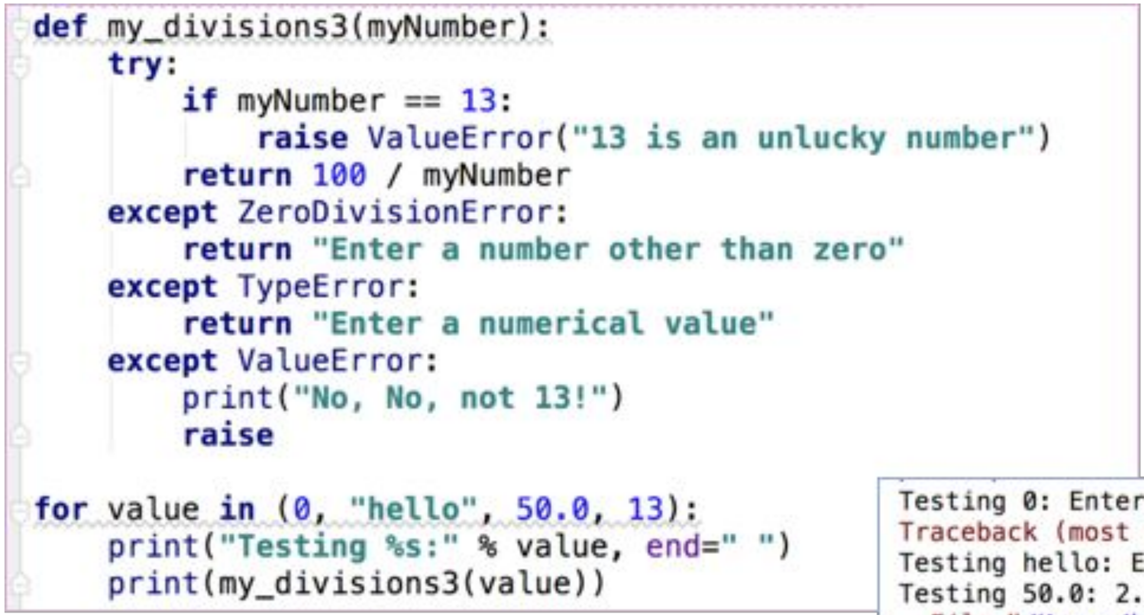
```
Testing 0:  Enter a number that is not zero!
Testing 50.0:  2.0
Testing hello:  Enter a number that is not zero!
Testing 13:  7.6923076923076925
```

[1]

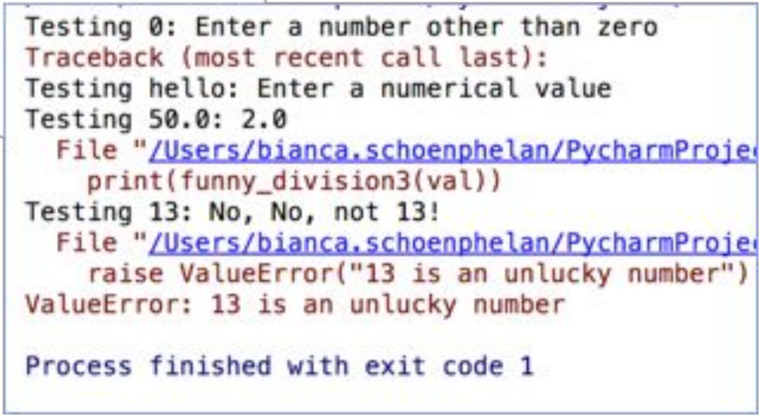
[1]

Handling different Scenarios cont’d

- We can stack exceptions
- Only first one will be executed that fits, even if more fit
- Remember all inherit from Exception



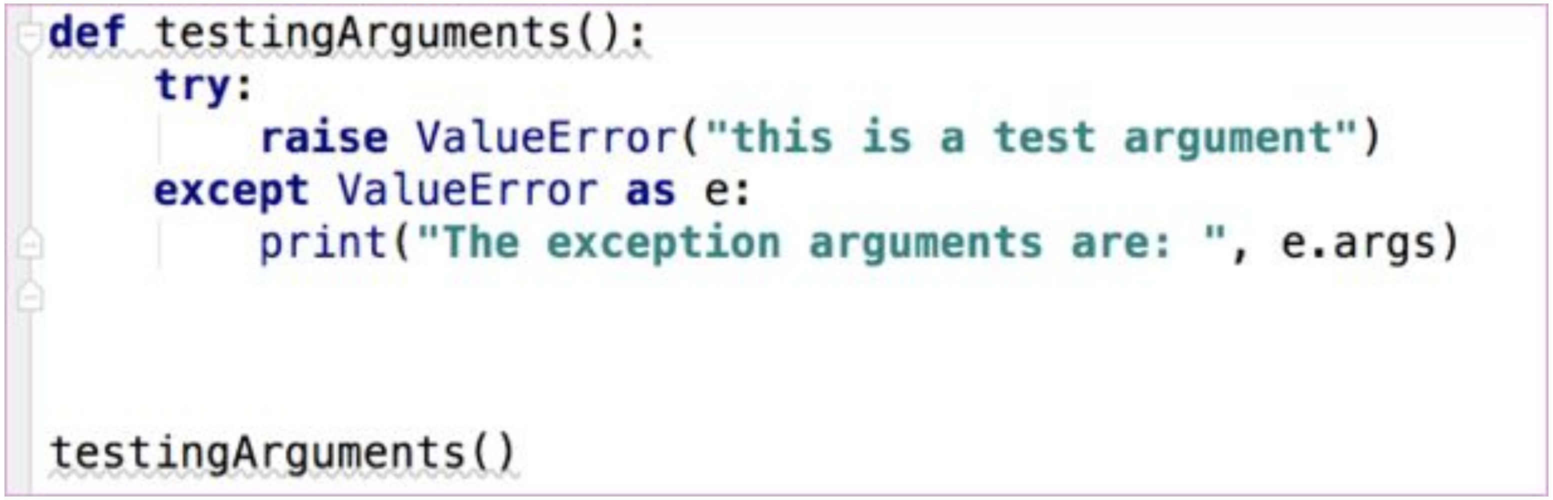
```python
def my_divisions3(myNumber):
    try:
        if myNumber == 13:
            raise ValueError("13 is an unlucky number")
        return 100 / myNumber
    except ZeroDivisionError:
        return "Enter a number other than zero"
    except TypeError:
        return "Enter a numerical value"
    except ValueError:
        print("No, No, not 13!")
        raise

for value in (0, "hello", 50.0, 13):
    print("Testing %s:" % value, end=" ")
    print(my_divisions3(value))
```



```
Testing 0: Enter a number other than zero
Traceback (most recent call last):
Testing hello: Enter a numerical value
Testing 50.0: 2.0
  File "/Users/bianca.schoenphelan/PycharmProje
    print(funny_division3(val))
Testing 13: No, No, not 13!
  File "/Users/bianca.schoenphelan/PycharmProje
    raise ValueError("13 is an unlucky number")
ValueError: 13 is an unlucky number

Process finished with exit code 1
```

Handling different Scenarios cont’d

- Allows us to enter our own text

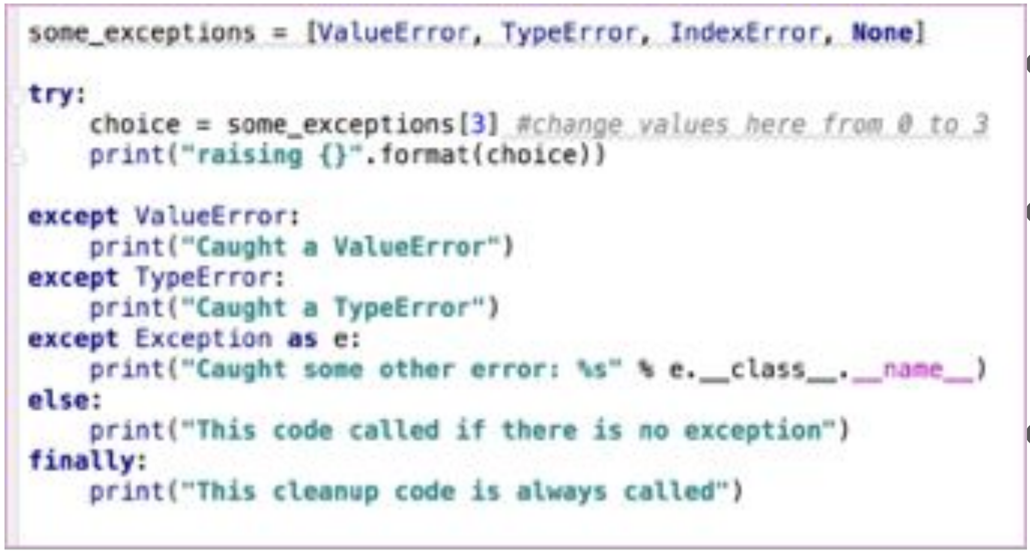


```python
def testingArguments():
    try:
        raise ValueError("this is a test argument")
    except ValueError as e:
        print("The exception arguments are: ", e.args)


testingArguments()
```

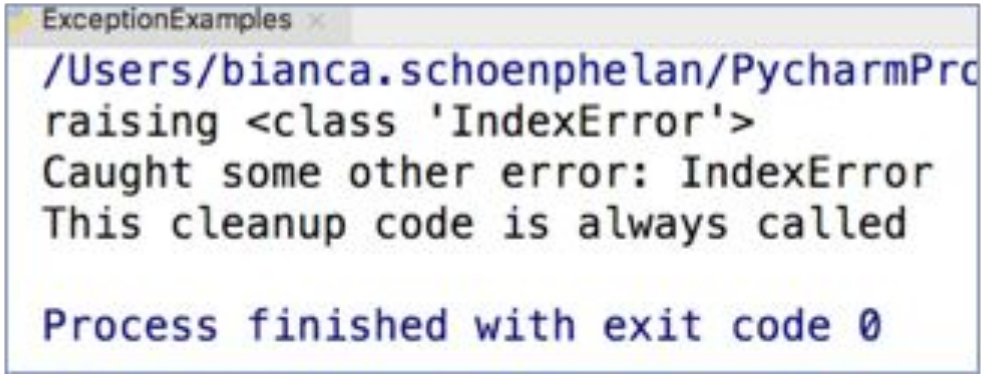
Still missing something

- Execute code regardless of whether or not an exception has occurred
- Specify which code should be executed only if an exception does not occur
- Solution: `finally … else`

Finally...else



```
some_exceptions = [ValueError, TypeError, IndexError, None]

try:
    choice = some_exceptions[3] #change values here from 0 to 3
    print("raising {}".format(choice))

except ValueError:
    print("Caught a ValueError")
except TypeError:
    print("Caught a TypeError")
except Exception as e:
    print("Caught some other error: %s" % e.__class__.__name__)
else:
    print("This code called if there is no exception")
finally:
    print("This cleanup code is always called")
```



```
ExceptionExamples
/Users/bianca.schoenphelan/PycharmPro
raising <class 'IndexError'>
Caught some other error: IndexError
This cleanup code is always called

Process finished with exit code 0
```

- IndexError will be handled in the Exception as e clause
- Print in the finally clause is always executed, no matter what happens, examples:
- Cleaning up database connection
- Closing an open file
- Send closing handshake over network
- Return statements in case of an error
- In case of no exception both else and finally are executed!

Hierarchy

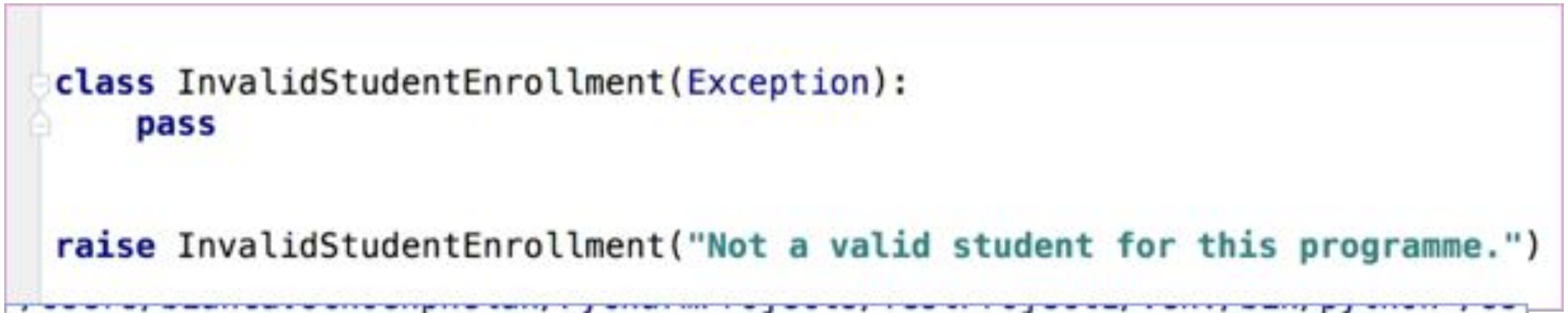
- After a try any of except, else or finally are optional and can be omitted
  - Else by itself is invalid though
- If use more than one, then the except must come first, then the else, then the finally
- Order of exceptions themselves usually go from most specific to most general

Define your own

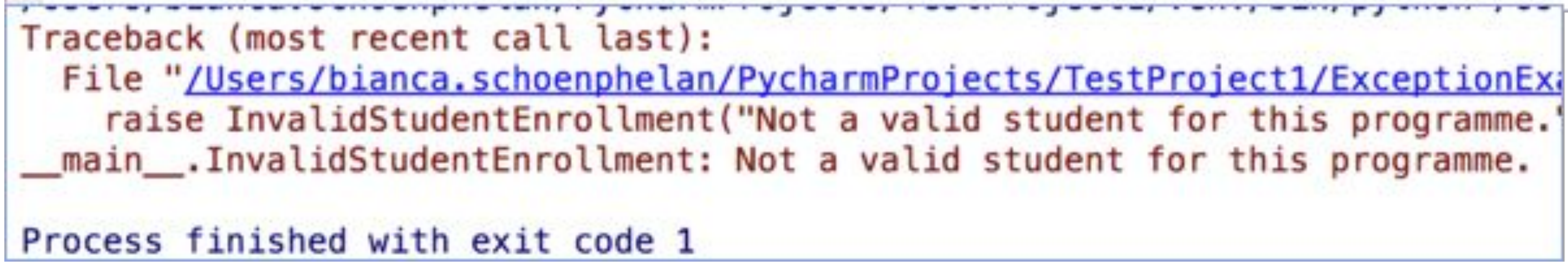
- Name of the class should communicate what went wrong
- Inherit from Exception
- We can define arguments too

Your own Exception

- You need to inherit from Exception
- That’s it
- Can use any argument type, string probably most commonly used.



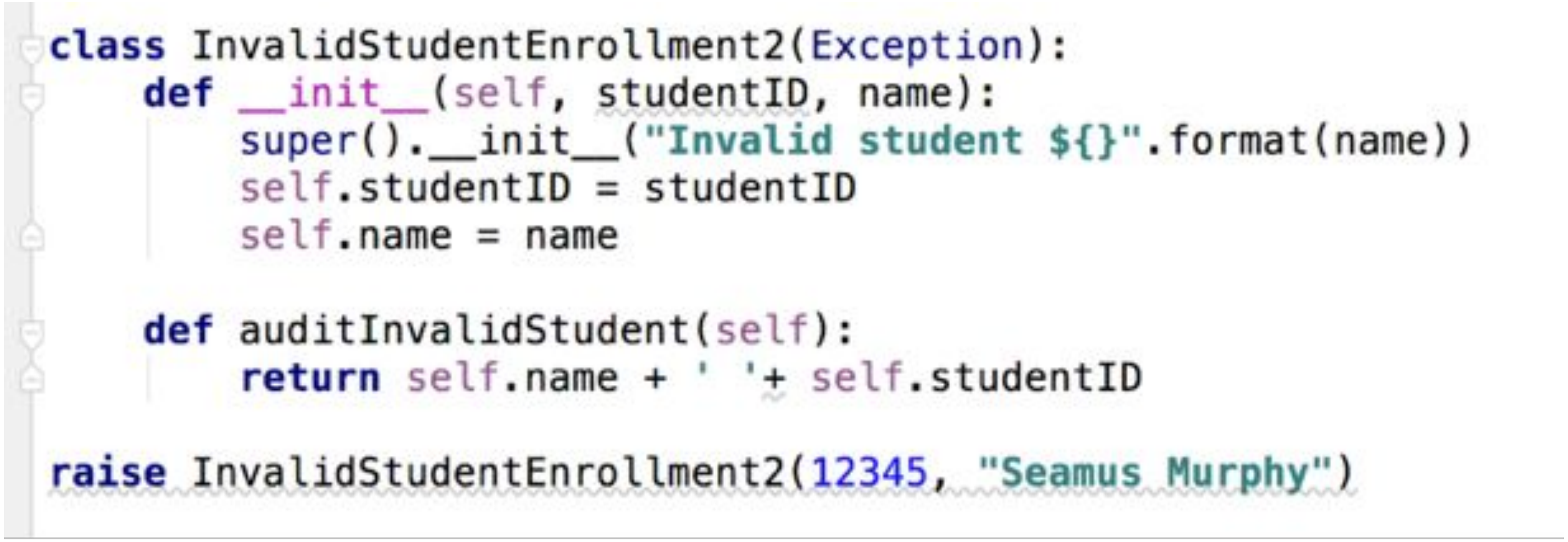
```
class InvalidStudentEnrollment(Exception):
    pass


raise InvalidStudentEnrollment("Not a valid student for this programme.")
```



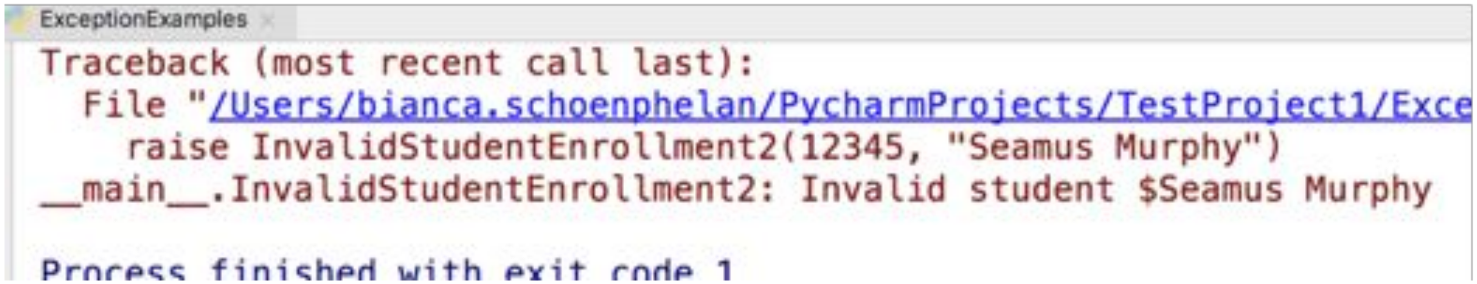
```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/PycharmProjects/TestProject1/ExceptionEx
    raise InvalidStudentEnrollment("Not a valid student for this programme.'
__main__.InvalidStudentEnrollment: Not a valid student for this programme.

Process finished with exit code 1
```

Your own Exception cont'd

You can include any amount of arguments.



```python
class InvalidStudentEnrollment2(Exception):
    def __init__(self, studentID, name):
        super().__init__("Invalid student ${}".format(name))
        self.studentID = studentID
        self.name = name

    def auditInvalidStudent(self):
        return self.name + ' '+ self.studentID

raise InvalidStudentEnrollment2(12345, "Seamus Murphy")
```



ExceptionExamples

```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/PycharmProjects/TestProject1/Exce
    raise InvalidStudentEnrollment2(12345, "Seamus Murphy")
__main__.InvalidStudentEnrollment2: Invalid student $Seamus Murphy

Process finished with exit code 1
```

Files

Files

- A file is a set of data stored on secondary storage such as a usb drive
- Accessing a file means establishing the connection to a file.
- Two types:
  - Text file
    - Human readable
    - Control characters, such as \n are typically interpreted
  - Binary file
    - Not human readable
    - Contains non-readable information
    - All information is directly translated

Open a File

- You open a stream or a file object steam
    - That’s the connection between any disk and your programme
    - Stream contains a buffer of the data from the file and provides that data to the program

Input/Output Stream

[2]

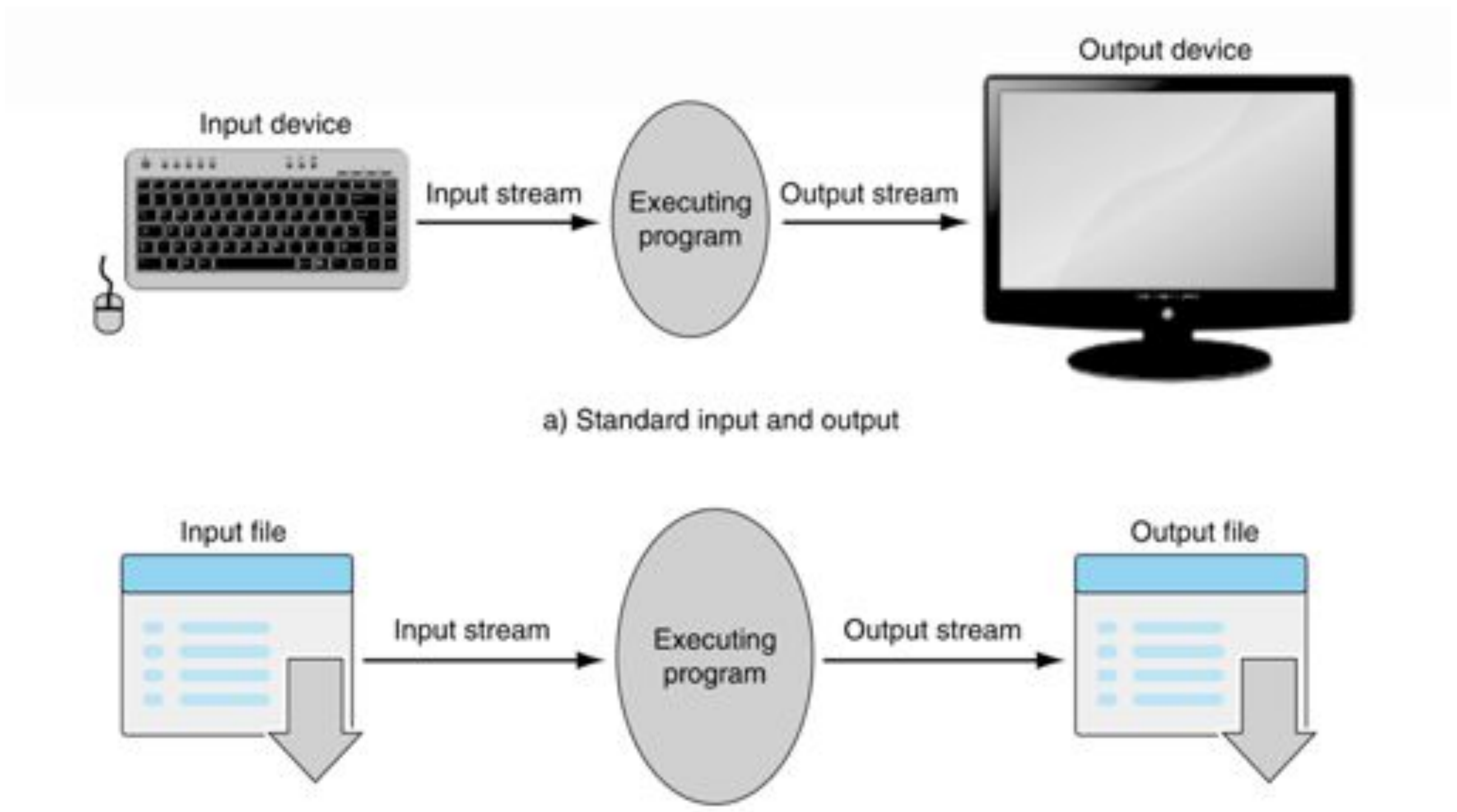


FIGURE 5.1 Input-output streams.

Buffering

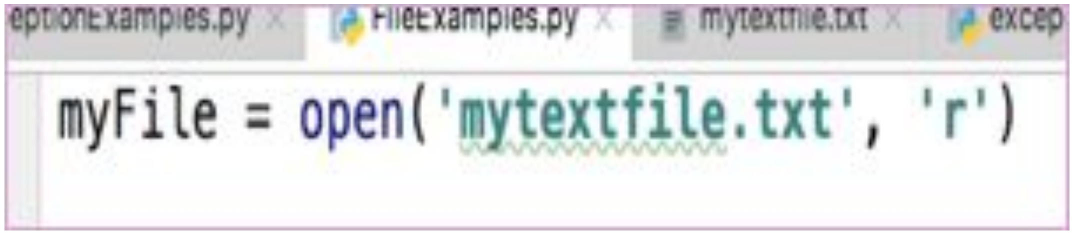
[2]

- Reading from a disk is typically very slow
- Latency and throughput
- Computer reads a lot in the hope that you might need further data later
- The copy of the data in the computer is in the cache

Making a File Object

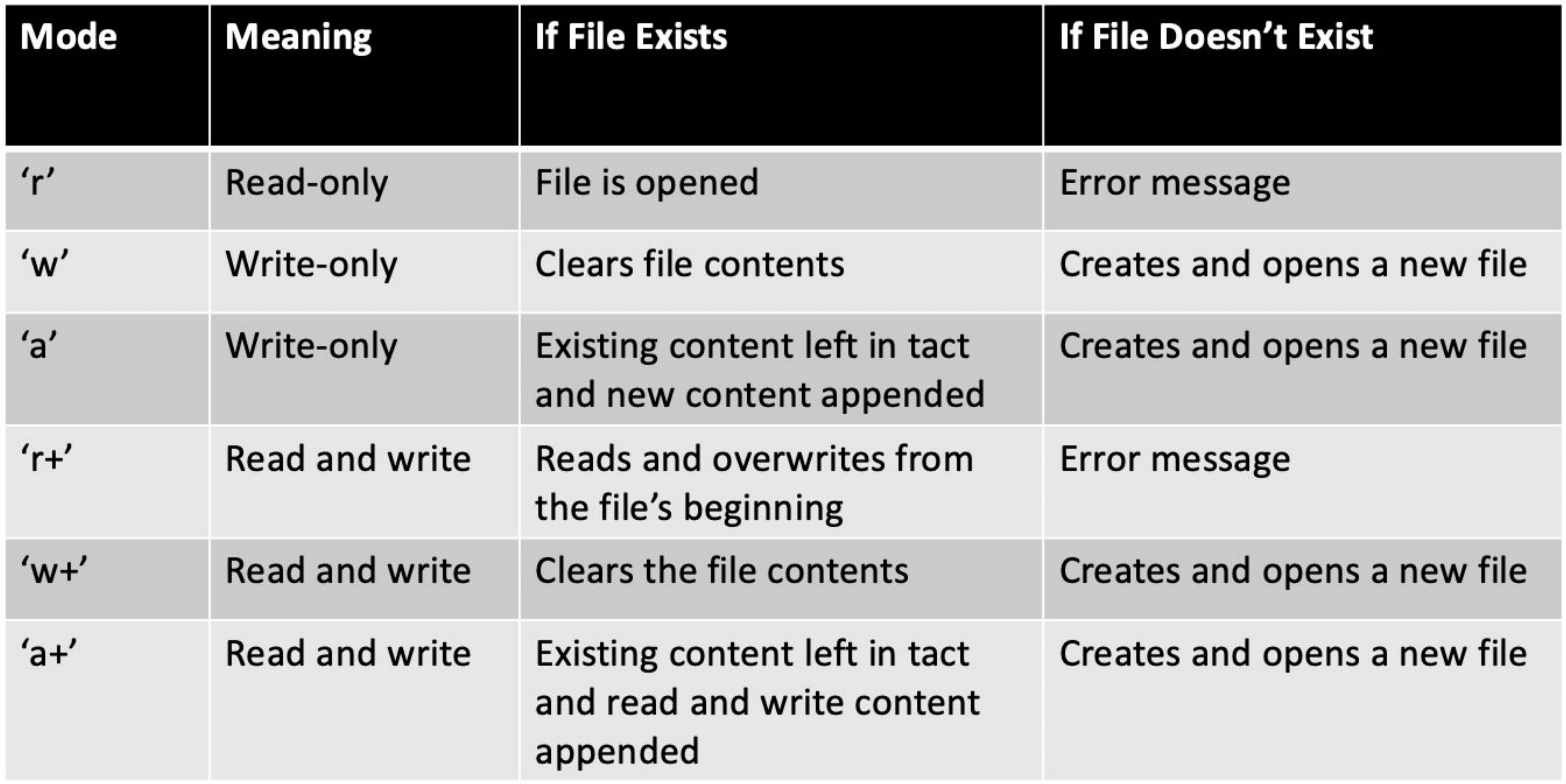
- myFile is a file object
  - Contains the buffer of information
- open function creates the connection
  - First parameter is the file name
  - Second is the mode to open, here r

- Two different forms possible for the name:
- Relative: "mytextfile.txt"
- Absolute: "C:\MyDocs\mytextfile.txt"



```
myFile = open('mytextfile.txt', 'r')
```

File Open Options

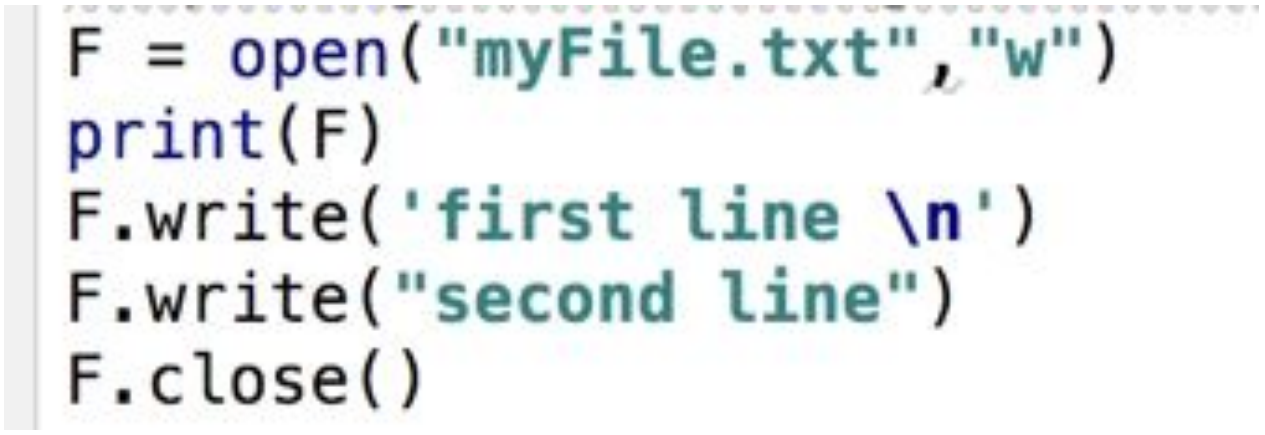


Mode	Meaning	If File Exists	If File Doesn't Exist
'r'	Read-only	File is opened	Error message
'w'	Write-only	Clears file contents	Creates and opens a new file
'a'	Write-only	Existing content left in tact and new content appended	Creates and opens a new file
'r+'	Read and write	Reads and overwrites from the file's beginning	Error message
'w+'	Read and write	Clears the file contents	Creates and opens a new file
'a+'	Read and write	Existing content left in tact and read and write content appended	Creates and opens a new file

Text Files

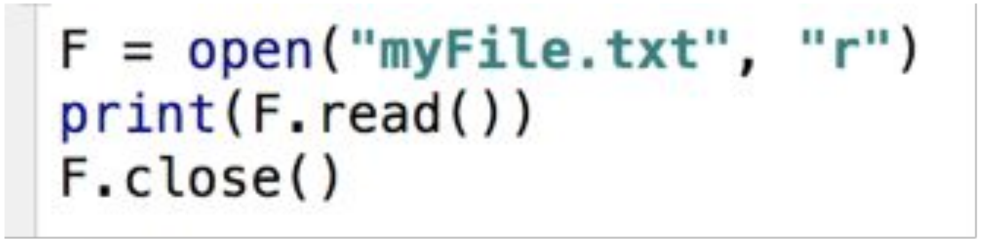
- Everything is a string
  - Everything read is a string
  - Everything written must be a string
- Once opened, print can be used to write to a file

Writing to a File

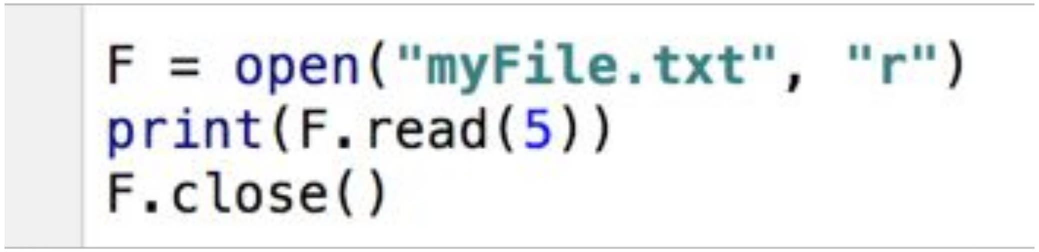

```
F = open("myFile.txt","w")
print(F)
F.write('first line \n')
F.write("second line")
F.close()
```

`write()` by itself does not add a newline character.

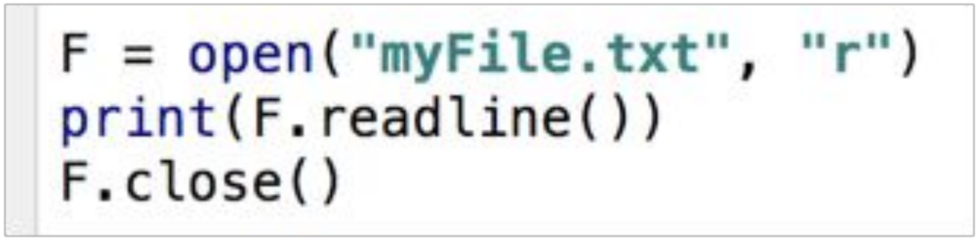
Reading from a File



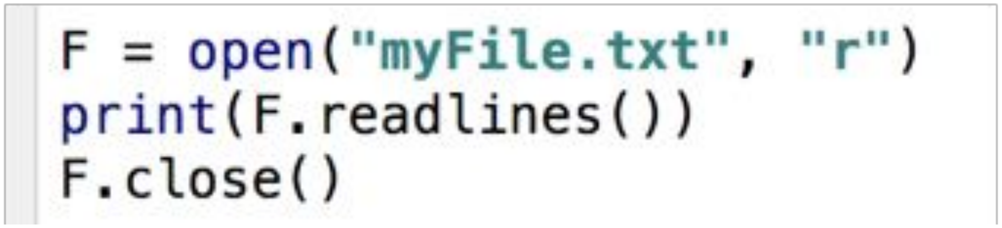
```
F = open("myFile.txt", "r")
print(F.read())
F.close()
```



```
F = open("myFile.txt", "r")
print(F.read(5))
F.close()
```



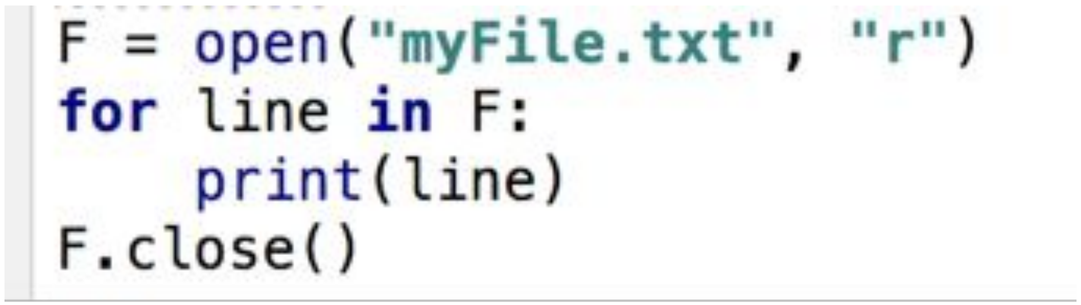
```
F = open("myFile.txt", "r")
print(F.readline())
F.close()
```



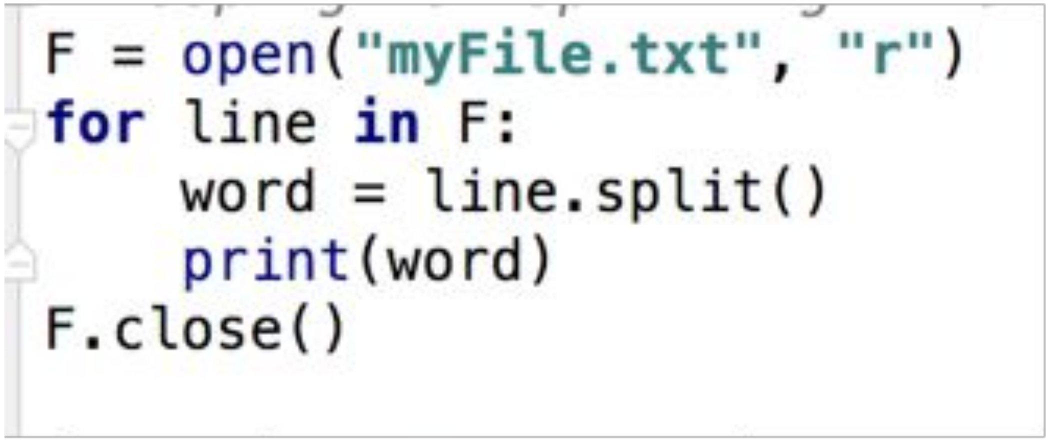
```
F = open("myFile.txt", "r")
print(F.readlines())
F.close()
```

`readline()` reads a string and automatically adds newline character apart from the last line

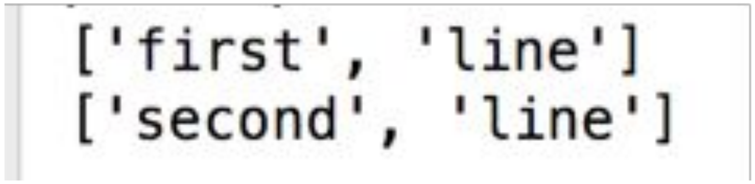
Looping through a File



```python
F = open("myFile.txt", "r")
for line in F:
    print(line)
F.close()
```



```python
F = open("myFile.txt", "r")
for line in F:
    word = line.split()
    print(word)
F.close()
```



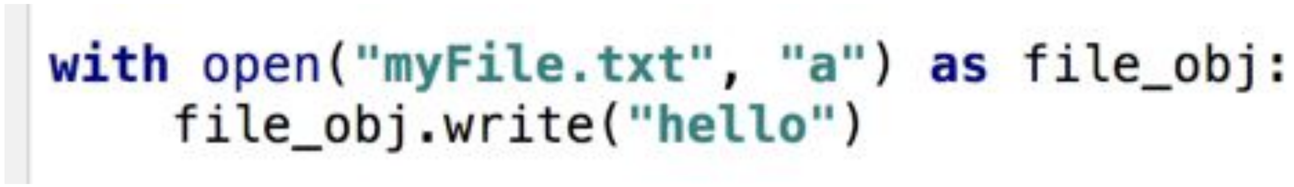
```
['first', 'line']
['second', 'line']
```

Why do we close()

- Flush the buffer contents from the computer to the file
- Stop the connection to the file

All open files should be closed, if you don't, you won't see any immediate effects but over time might get interesting memory effects.

- For an automatic close use with



```
with open("myFile.txt", "a") as file_obj:
    file_obj.write("hello")
```

Summary

- ★ Built-in exceptions
- ★ Build your own exceptions
- ★ Handle files

References

1. Python 3: Object Oriented Programming, Dusty Phillips, 2nd edition, 2015
2. The Practice of Computing Using Python, Punch, Enbody , 2013 Pearson Addison-Wesley.