Exceptions, Testing and File-Handling, Enums

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 5

Recap

- Objects and Classes
- OOP Principles
  - Abstraction
  - Polymorphism
  - Encapsulation
  - Information Hiding
- Single Inheritance
- Multiple Inheritance
- Interfaces, Interfaces, Interfaces….
  - Informal – Ducktyping
  - Informal – Protocols
  - Formal – Abstract Base Classes

Previously On…

COMP 3522

Recap: Kwargs

A FINAL WORD

Variable Keyword Arguments : Kwargs

```
def variable_keyword_args(**kwargs):
    for key, item in kwargs.items():
        print(key, item)

variable_keyword_args(port=21, host='localhost', debug=True)
```

A final note about kwargs

Don’t need to use the variable name kwargs

The specific name kwargs is not a keyword

Variable Keyword Arguments : Kwargs

```
def variable_keyword_args(**dogs):
    for key, item in dogs.items():
        print(key, item)

variable_keyword_args(port=21, host='localhost', debug=True)
```

It can replaced with any variable name, ie: dogs, and the result is functionally the same

Errors and Exceptions

TRY RAISE CATCH ELSE FINALLY

Errors and Exceptions

In all our attempts to write encapsulated and polymorphic OOP code, we tend to forget about all the edge cases and ways our program can fail.

Like many other languages Python raises Errors and exceptions that notify the developer of what went wrong, and where.

Errors usually provide useful messages and a stacktrace showing all the line numbers and function calls involved.

```
Traceback (most recent call last):
  File "D:/Google Drive/BCIT Work Synced Folder/My Courses/COMP3522 OOP 2/Fall 2019/Week 5/Quiz Code/question one.py", line 19, in <module>
    main()
  File "D:/Google Drive/BCIT Work Synced Folder/My Courses/COMP3522 OOP 2/Fall 2019/Week 5/Quiz Code/question one.py", line 16, in main
    print(f'Salary: {employee_account_data["invalidkey"]}')
  File "D:/Google Drive/BCIT Work Synced Folder/My Courses/COMP3522 OOP 2/Fall 2019/Week 5/Quiz Code/question one.py", line 11, in __getitem__
    return self.data[item]
KeyError: 'invalidkey'
```

Exceptions

Open up the python command shell and try some of this code:

```
>>> user_choice = int(input(“Enter Menu Option”))
>>> “Hello”
```

```
ValueError: invalid literal for int() with base 10: 'hello'
```

```
>>> my_list = [1,2,3]
>>> my_list + ‘4’
```

```
TypeError: can only concatenate list (not "str") to list
```

```
>>> my_list.add(4)
```

```
AttributeError: 'list' object has no attribute 'add‘
```

Exceptions

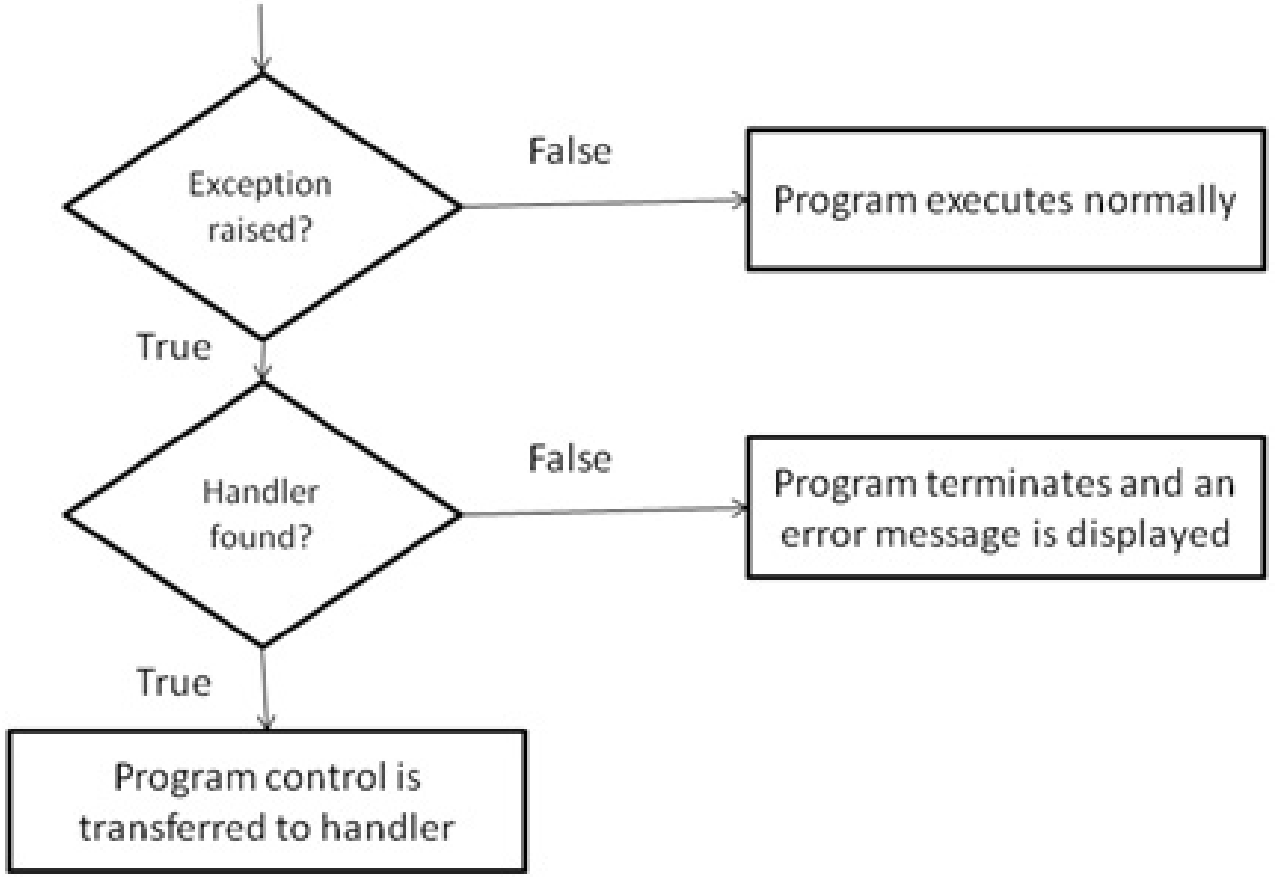
```
>>> 9/0
ZeroDivisionError: division by zero


>>> print “99”  # This used to work in Python 2
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("99")?


>>> print(my_variable)
NameError: name 'my_variable' is not defined
```

Common Exceptions and Errors

- AttributeError
- FloatingPointError
- IndexError
- KeyError
- SystemExit
- NotImplementedError
- NameError
- ZeroDivisionError
- SyntaxError
- TypeError
- StopIteration
- ValueError
- AssertionError
- MemoryError

All of these inherit from the BaseException class.

Exceptions & Exception Handling



Exception & Exception Handling

Let’s look at some code

Throwing an exception - **raise**

In languages like Java or C++, we're probably used to throwing exceptions with the keyword throw

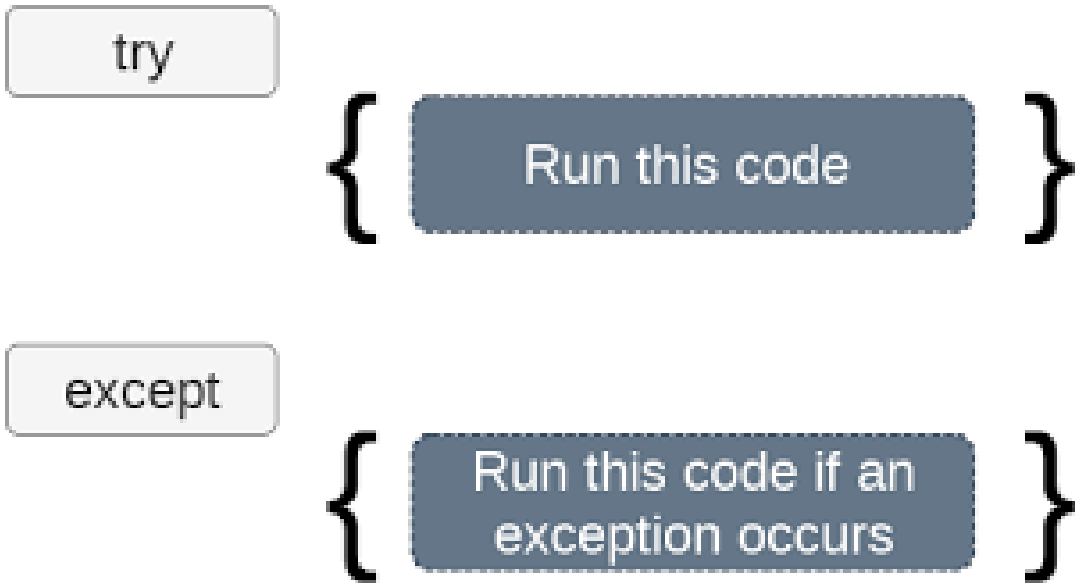
Python uses a different keyword, raise

Functionally does the same thing

```
def divide(num1, num2):
    if num2 == 0:
        raise ZeroDivisionError()
    return num1/num2
```

Try - Except

```
try:
    # The code that can possibly throw an exception
    add_something(1,3)
    add_something('3',4)
except:
    # Executed if an exception occurs
    print("An Exception was caught")
```



The **except** keyword is the python equivalent of **catch**

in Java/C++

This is how we deal with Dynamic Types

```
def add_something(val1, val2):

    if not isinstance(val1, int) or not isinstance(val2, int):
      raise TypeError("Invalid data type. Can only add integers")

    print(val1 + val2)

…

…

try:
    add_something(1,3)
    add_something('3',4)
except:
    print("An Exception was caught")
```

Try – Except (Multiple Exception Types)

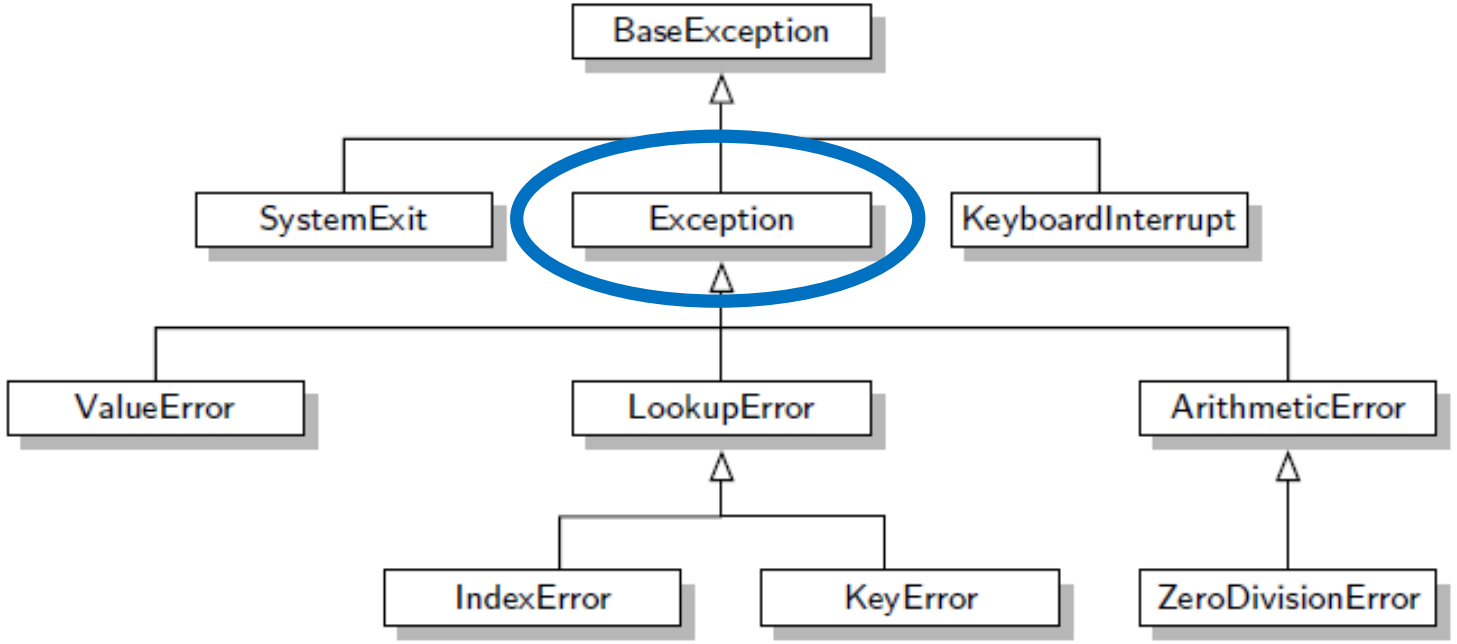
How do we catch different kinds of exceptions?

What if we want to handle each exception differently?

Let's look at some more code!

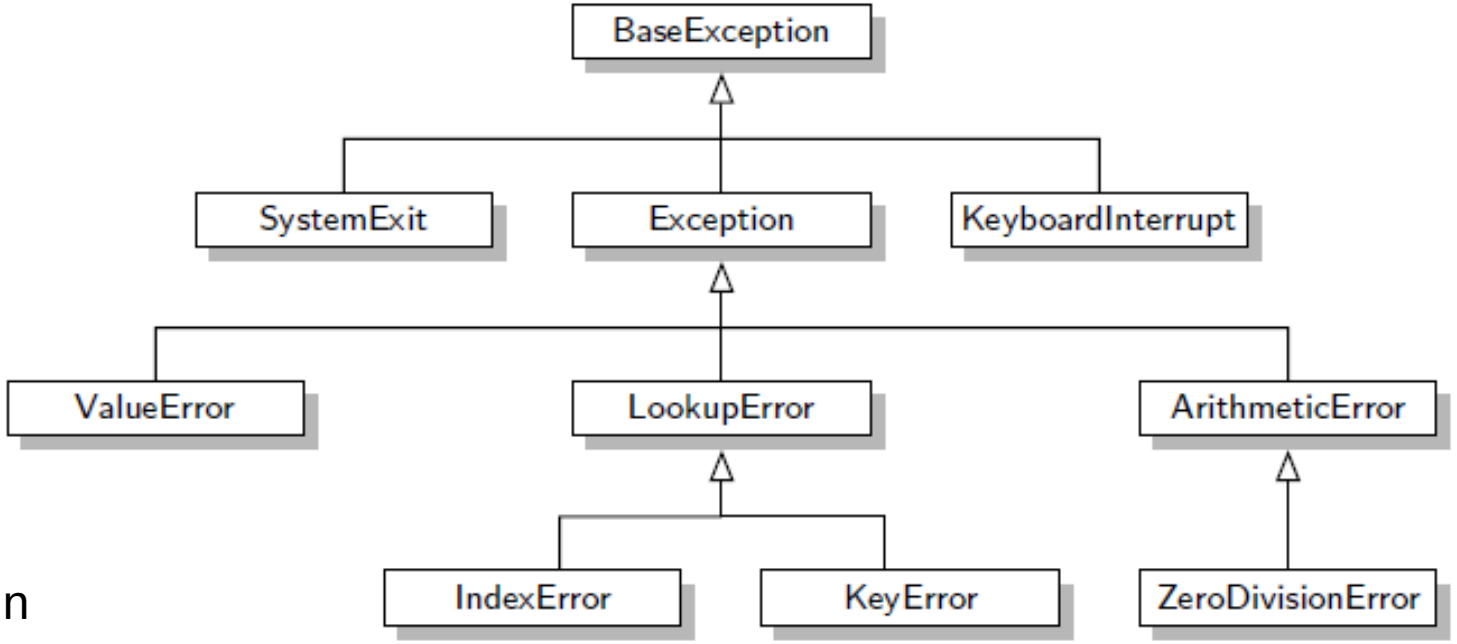
A Note on Exception Inheritance Hierarchy

- All Exceptions are objects that inherit from *BaseException*
- Most Exceptions we handle inherit from *Exception*, which inherits from *BaseException*.
- If we want an except clause that will handle any exception, we would try to handle the exception of type '*Exception*'



A Note on Exception Inheritance Hierarchy

- SystemExit and KeyboardInterrupt are special exception types that do not inherit from *Exception* . This is because we generally do not want to catch these exceptions. They are used to exit the program, or interrupt a program from the terminal (Ctrl + C).
- We should **NEVER** have an empty *except* clause since that would catch all the exceptions.

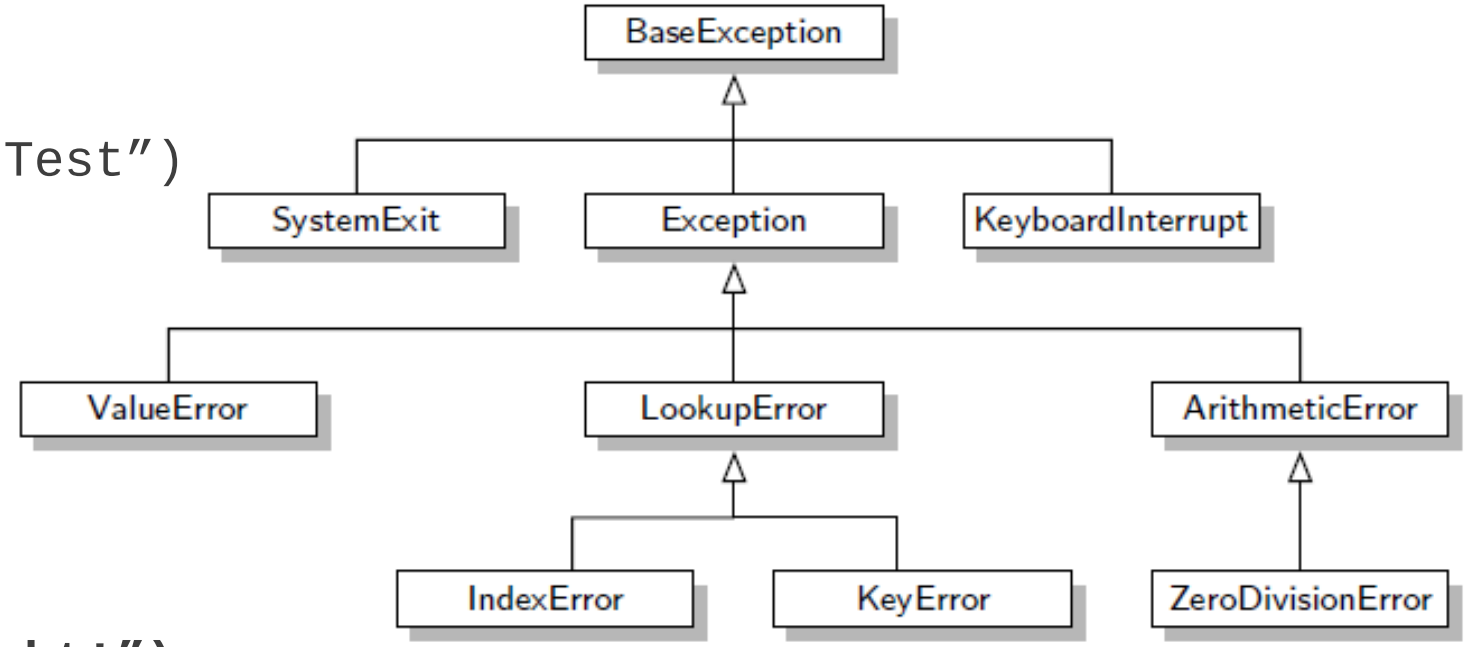



```
try:
    # code that throws exception
except:
    # catches all exceptions
```

A Note on Exception Inheritance Hierarchy

What would get printed?

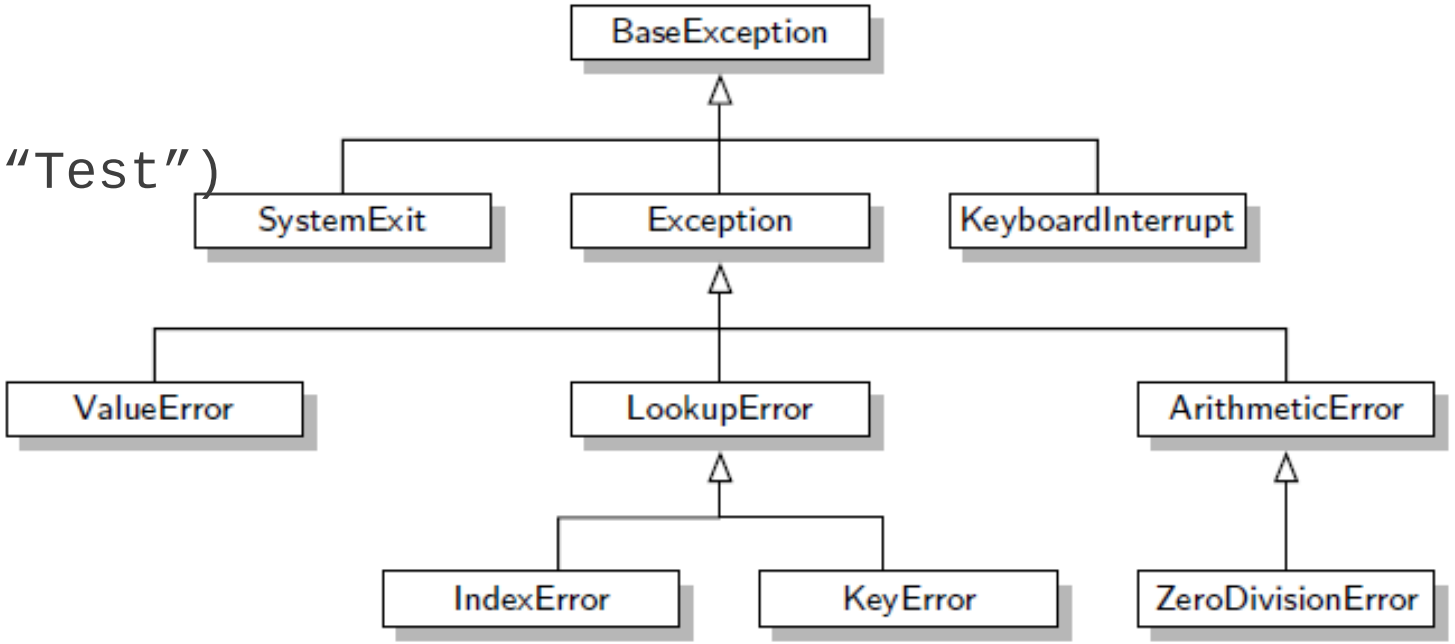
```
class Test():
    def some_method(self):
        raise ZeroDivisionError("Test")


test_obj = Test()
try:
    test_obj.some_method()
except Exception:
    print("Exception caught!")
except ZeroDivisionError:
    print("ZeroDivisionError caught!")
```



A Note on Exception Inheritance Hierarchy

What would get printed?

```
class Test():
    def some_method(self):
        raise ZeroDivisionError("Test")


test_obj = Test()
try:
    test_obj.some_method()
except Exception:
    print("Exception caught!")
except ZeroDivisionError:
    print("ZeroDivisionError caught!")
```



A Note on Exception Inheritance Hierarchy

What would get printed?

```
class Test():
    def some_method(self):
        raise ZeroDivisionError("Test")


test_obj = Test()
try:
    test_obj.some_method()
except Exception:
    print("Exception caught!")
except ZeroDivisionError:
    print("ZeroDivisionError caught!")
```

- Order of catching exceptions matters
- ZeroDivisionError inherits from Exception
- Therefore except Exception catches ZeroDivisionError since it is a subclass of Exception
- We should always catch the subclass versions of exceptions before wider parent exceptions
- In fact, PyCharm will warn you that except Exception is too broad

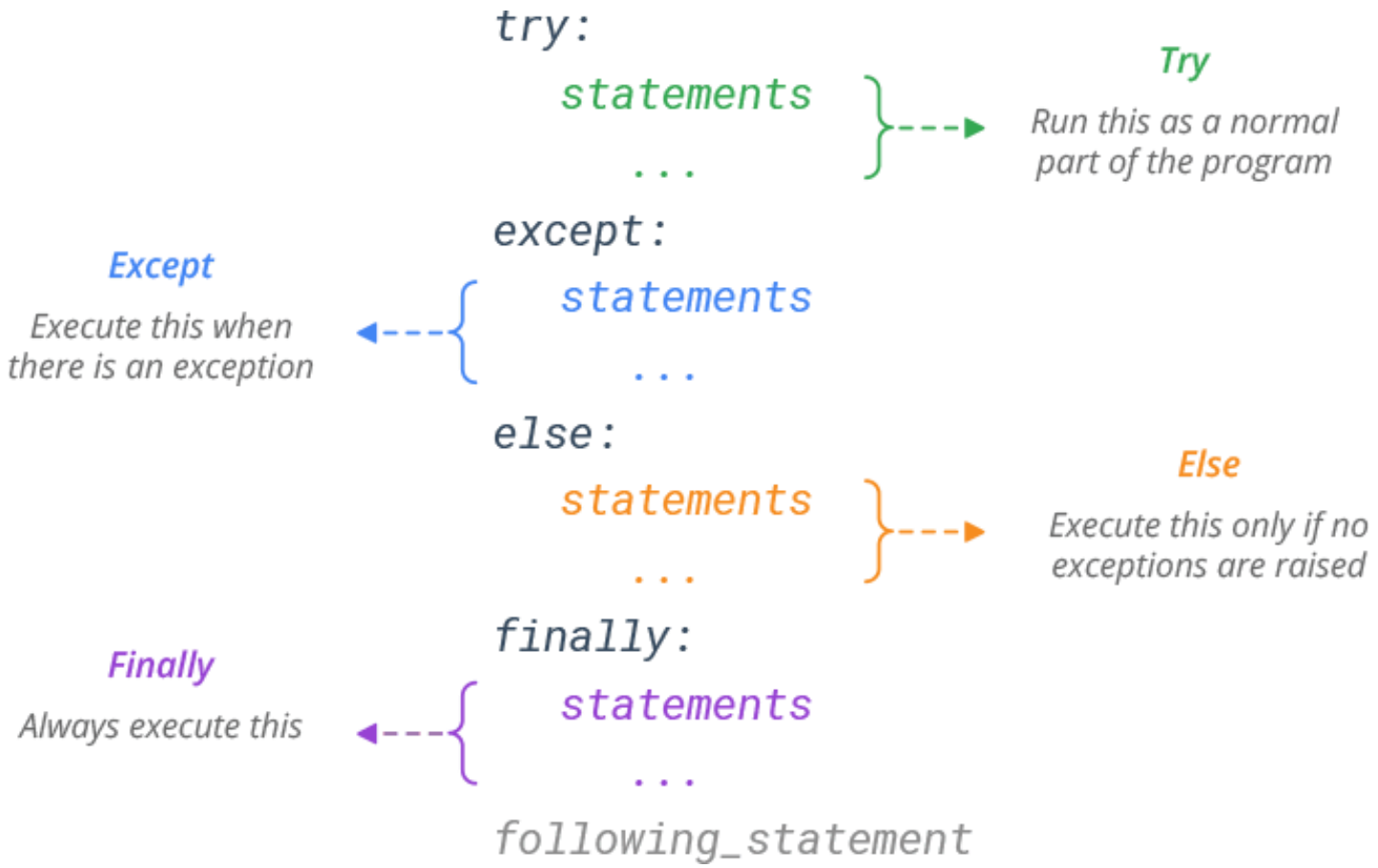
A Note on Exception Inheritance Hierarchy

What would get printed?

```
class Test():
    def some_method(self):
        raise ZeroDivisionError(“Test”)


test_obj = Test()
try:
    test_obj.some_method()
except ZeroDivisionError:
    print(“ZeroDivisionError caught!”)
except Exception:
    print(“Exception caught!”)
```

- The code on the left moves ZeroDivisionError higher up
- This allows us to properly get the more specific error message “ZeroDivisionError caught”

Try – Catch – Else - Finally



Easier To Ask For Forgiveness Than Permission

A Philosophy that python has adopted.

Multiple If-Checks are bad (Look Before You Leap)

Assume code will work

Handle any exceptions/errors as they arise

Why are If-Checks bad?

Easier To Ask For Forgiveness Than Permission

A Philosophy that python has adopted.

Multiple If-Checks are bad

Assume code will work

Handle any exceptions/errors as they arise

Why are If-Checks bad?

- Exceptions are fast
- If-checks waste CPU cycles for situations that generally don't occur often
- Keep our functions Atomic. They only do one thing. Let someone else deal with the exceptions.

Custom Exceptions

Since all exceptions are objects that inherit from Exception, we can define our own custom exceptions!

Make a regular class inherit from Exception

```
class MyCustomClass(Exception):
    #class code
```

Custom Exceptions

Since all exceptions are objects that inherit from Exception, we can define our own custom exceptions!

Optional : Create an init with a custom error message

```
def __init__(self, invalid_char):
    pass
```

args and as in Exceptions

When catching an exception, we might want to get the instance of an exception to retrieve some data

This is done by adding the keyword **as**, and a variable name after the exception we just caught

```
except MyException as e:
    print(e.args[0])
```

This catches the exception into the instance variable `e`

After that we can access the exception like an instance of any other class

args and as in Exceptions

The BaseException's initializer contains an args to store a tuple of arguments

Typically there is only one argument, which is a string describing the error

However additional arguments can be accessed using [] on the args

```
try:
    raise MyException("Custom Error message", 999)
except MyException as e:
    print(e.args) #('Custom Error message', 999)
    print(e.args[0]) #Custom Error message
    print(e.args[1]) #999
```

character_example.py, lecture_exception_example.py

A Quick Recap: Unit Tests

What is Unit Testing?

We need to test our code often!

Part of test Driven development involved writing our test cases first, and THEN writing the code.

Once code is written, we cross check it with our test cases to ensure it meets standards and works as intended.

Unit Tests

WHAT: A piece of code that exercises a very small, specific area of functionality of a given system.

HOW: A unit test invokes a particular function with a given input. It then asserts whether the output received matches a given expected output.

WHY?

It is an organized approach which helps break down our program into small bite sized pieces of functionality. Each chunk of functionality can be tested in isolation.

Unit Tests

Steps:

- Assemble – Create an environment in which the function can run
- Act – Run the code with a given input(s).
- Assert – Compare the result

Rules:

- A collection of unit tests
- Each test is a function that contains an assertion
- Assertions must test individual functions of your code
- The unit test functions go inside the test class
- **The functions must all begin with test**

Assertions

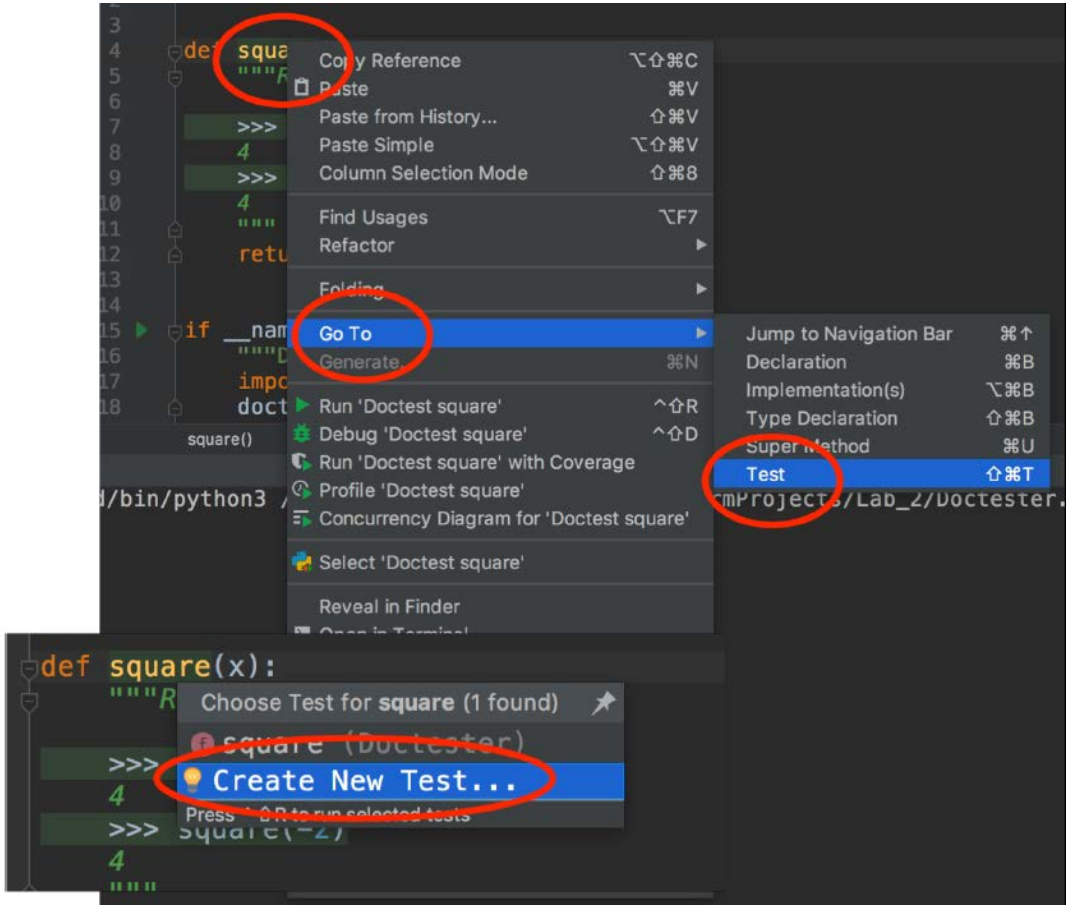
Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

Unit testing in PyCharm

Right Click on the function we want to use.

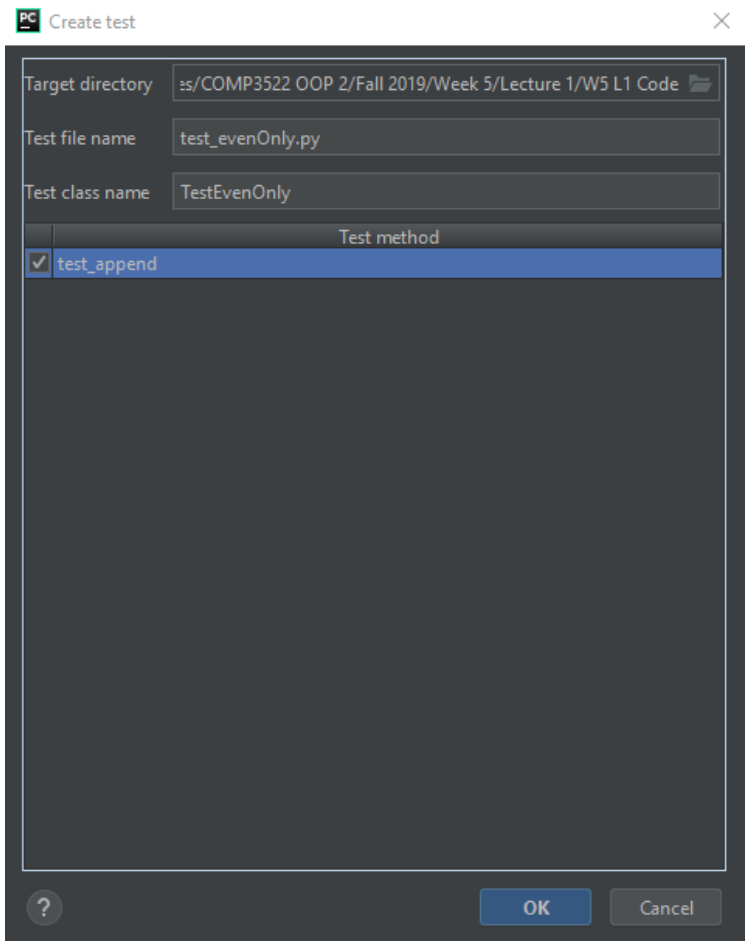
Select Go To -> Test

In the dialog select Create New Test if this is the first unit test for that function



Unit testing in PyCharm

Make sure your function is checked



Unit testing in PyCharm

```
#created by default
class Test(TestCase):
    def test_sum(self):
        self.fail()
```

```
#after edit
class Test(TestCase):
    def test_sum(self):
        self.assertTrue(MyClass.sum(1, 2)==3)
```

A new Test class will be generated for you, along with a new test function

The generated test function will be set to fail by default. You will need to write code to call your code and create assertions that evaluate to true

To create additional test cases, just add additional “def test...” functions to the Test class

Note you MUST begin functions with the word **test** or they will not appear as runnable tests

calculator.py, test_calculator.py

File Handling

Files

To work with a file in Python, the following sequence needs to be followed

1. Open a file – Specify the encoding, file mode and file name
2. Conduct File Operations – Read, Write, Append
3. Close the file – Release memory and tie up any loose ends.

Python Docs:

https://docs.python.org/3/tutorial/inputoutput.html

File Mode	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

File – Example

```
#Step1: Open a file
my_text_file = open("sample.txt", mode='r', encoding='utf-8')

#Step2: File operations
data = my_text_file.read()
print(f"File Data:\n{data}")

#Step3: Close the file
my_text_file.close()
```

With keyword

Manually closing files is not good practice

In the event of an exception, the program will halt execution and the close() function will not be executed.

To circumvent this, Python implements a **`with`** keyword.

The with keyword defines an **identifier** which represents an object like a file, an active connection to a server, anything that needs to be initialized and de-initialized.

To adhere to this protocol for user defined types, the `__enter__(self)` and `__exit__(self)` methods need to be implemented

Refer to PEP 345 for more information:

https://www.python.org/dev/peps/pep-0343/

With keyword - Example

Syntax:

```
with EXPR as VAR:

    BLOCK1
```

Example:

```
with open("test.txt",'w',encoding = 'utf-8') as f:
    f.write("my first file\n")
    f.write("This file\n\n")
    #  This will cause an exception. The file will close safely
    f.read()
    f.write("contains three lines\n")
```

Using with vs not using with

```
# Using with

with open("sample.txt", mode='r', encoding='utf-8') as my_text_file:

      data = my_text_file.read()
       print(f"File Data:\n{data}")



# Not using with
my_text_file = open("sample.txt", mode='r', encoding='utf-8')

data = my_text_file.read()
print(f"File Data:\n{data}")

my_text_file.close()
```

Seek

```
with open("sample.txt", mode='r') as my_text_file:

    data = my_text_file.read()
    print(f"File Data:\n{data}")

    # This wont print anything since the file pointer
    # is at the end of the file
    data = my_text_file.read()
    print(f"Printing Data again: {data}")

    # You need to seek back to the beginning for
    # multiple reads
    my_text_file.seek(0)
    data = my_text_file.read()
    print(f"Printing data after seeking to the beginning:\n
          {data}")
```

file_handling_example.py

Load json from file

Import json module

Use with open just like a text file

Data is formatted in a special way, need to use json.load(identifier)

Data converted and stored into a dictionary

```
import json

with open("data.json", mode='r', encoding='utf-8') as data_file:
    data = json.load(data_file)
    print(data)
```

Write to json file

Import json module

Open a new person.json file, set to ‘w’rite mode

- Open file as json_file

Convert dictionary into json, then write json file out

```
import json
person_dict = {"name": "Home", "children": ["Bart", "Lisa","Maggie"],
"married": True, "fav_food": 'Donut' }

with open('person.json', 'w') as json_file:
  json_data = json.dumps(person_dict)
  json_file.write(json_data)
  # json.dump(person_dict, json_file) #same as above 2 lines
```

Enums

I LOVE ENUMS!

Enums

Coding problem

Imagine I ask you to write a calendar program

You might represent the days in a week by the numbers 0-6, with 0 meaning Sunday -> 6 meaning Saturday

However if we write something like

- `day = 0`

It's a little difficult to realize that 0 in this case means Sunday

Enums

A slightly better approach is to define a series of constants to represent each day, then later on we assign the constants to the day

```
SUNDAY = 0

MONDAY = 1

…

SATURDAY = 6
```

Then we write something like

- `day = SUNDAY`

The code is definitely easier to read, but we can do better.

Enums

There are only 7 days in a week, it'd be great if we could wrap the 7 day constants in some data type

An enumeration (Enum) is a user-defined type that consists of a set of named value constants that are known as enumerators

The members of an enumeration can be compared by these names, and the enumeration itself can be iterated over

Whenever we need to represent simple constant data over a finite range, Enums are a great option

- Makes code easier to use and read!
- What looks easier to understand?
  - `if day == 1:`
  - `if day == Days.Monday:`

Enums

```
import enum
class Days(enum.Enum):
   Sun = 0
   Mon = 1
   Tue = 2
   Wed = 3
   Thu = 4
   Fri = 5
   Sat = 6
```

Let’s represent the previous days of the week example using enums

Enums in Python are classes that inherit from the Enum class

Make sure to import from the enum module

Variables that are set in the enum are the enumerators

- Effectively setting constants to a specific value

Enums

```
import enum
class Days(enum.Enum):
   Sun = 0
   Mon = 1
   Tue = 2
   Wed = 3
   Thu = 4
   Fri = 5
   Sat = 6
```

Enums can be accessed through their name or value.

Notice how there's no need to instantiate the Days class (d = Days(3))

```
day = Days.Mon
print(type(day)) #<enum 'Days'>
print(day) #Days.Mon
print(day.name) #Mon
print(day.value) #1
print (Days['Sat']) #Days.Sat
print (Days(4)) #Days.Thu
```

Enums

```
import enum
class Days(enum.Enum):
  Sun = 0
  Mon = 1
  Tue = 2
  Wed = 3
  Thu = 4
  Fri = 5
  Sat = 6
```

Can not modify the name or value of an enum after it's been created

```
day = Days.Mon
day.name = Days.Thu.name #ERROR
day.value = 5 #ERROR
day = Days.Tue
```

Enums

```
import enum
class Days(enum.Enum):
  Sun = 0
  Mon = 1
  Tue = 2
  Wed = 3
  Thu = 4
  Fri = 5
  Sat = 6
```

Can print out all enum members as though the Enum class is an iterable object

```
for day in Days:
  print(day)
```

Output:

Days.Sun
Days.Mon
Days.Tue
Days.Wed
Days.Thu
Days.Fri
Days.Sat

Enums

```
import enum
class Days(enum.Enum):
  Sun = 0
  Mon = 1
  Tue = 2
  Wed = 3
  Thu = 4
  Fri = 5
  Sat = 6
  Sat = 9 #ERROR
  Sat2 = 6 #OK
```

Can NOT have member within an enum have the same name

BUT

You can have different members within an enum that have the same value

Real world example: Representing clothing in a videogame

You need to have a finite number of clothing options in your game

Each piece of clothing needs to have a unique id

There must be a way to find the number of:

- Only shoes
- Only pants
- All items

How can we do this with only enums?

Real world example: Representing clothing in a videogame

```
import enum
from enum import auto
class ClothingId(enum.Enum):

    shoe1 = auto() #1
    shoe2 = auto() #2
    shoe3 = auto() #3
    pant1 = auto() #4
    pant2 = auto() #5
    pant3 = auto() #6
    pant4 = auto() #7
    pant5 = auto() #8
```

Real world example: Representing clothing in a videogame

```
import enum
from enum import auto

class ClothingId(enum.Enum):
    shoeBegin = shoe1 = auto() #1
    shoe2 = auto() #2
    shoe3 = auto() #3
    shoeEnd = pantBegin = pant1 = auto() #4
    pant2 = auto() #5
    pant3 = auto() #6
    pant4 = auto() #7
    pant5 = auto() #8
    pantEnd = numClothes = auto() #9
```

clothing_enum.py

That’s it for today!

Lab 5 deals with file IO

Assignment 1 due Feb 14th

Quiz on Friday covering last week’s material