ARCTIC Summer Workshop 2023

Day 1- Afternoon Session (1 pm – 3:30 pm)

Topics we will cover

- Variables and Data Types
- Control Flow
- Operators
- Modules
- Functions
- Input and Output Operations
- Lists and Indexing
- Error Handling
- Class
- Virtual Environment

Python Data Types

- Basic Primitive Data Types
  - int
  - float
  - str
  - boolean
- Non primitive data types
  - List
  - Dictionary
  - Tuple
  - Set
- Refer: https://docs.python.org/3/library/datatypes.html

Non-Primitive Data Types

- Lists
  - A list is a container that can store anything you want.
  - Empty list or String list or Integer List or Mixed list or List of list
  - Ordered, Changeable, Allow duplicate values
  - faculty =["Computer Science","Population Health","Social Science","Humanities","Education"]
  - Elements can be modified
    - faculty[2] = "Economics"

Non-Primitive Data Types

- Tuples
  - Tuples are like lists, except that they cannot be modified once created, that is they are immutable.
  - In Python, tuples are created using the syntax (..., ..., ...), or even ..., ...:
    - point = (10, 20)
    - point = 10, 20

Non-Primitive Data Types

- Set
  - Unordered and unindexed
  - Do not allow duplicate values
  - Can contain different data types
  - myset = {"Computer Science","Population Health","Social Science","Humanities","Education", "Computer Science"}
  - print (myset)
    - {'Social Science', 'Computer Science', 'Humanities', 'Education', 'Population Health'}
  - myset = {"Computer Science", 1, 2.45, True}

Non-Primitive Data Types

- Dictionary
  - Dictionary maps keys to the values.
  - Dictionary can hold anything like lists: numbers, strings, mixed of both and other objects
  - faculty ={"Computer Science":101,"Population Science":104,"Arts":109}

Control Flow

- Control flow refers to the order in which statements are executed.
- Conditional statements (if, elif, else) for decision-making and loops (for, while) for repetition.

Python Operators

- Arithmetic Operators
    - Addition (+)
    - Subtraction (-)
    - Multiplication (*)
    - Division (/)
    - Modulus (%)
    - Floor division (//)
    - Exponent (**)

Python Operators

- Comparison Operators
  - (less than) <
  - (less than equal to) <=
  - (equal) ==
  - (greater than) >
  - (greater than equal to) >=
  - (not equal) !=

Python Operators

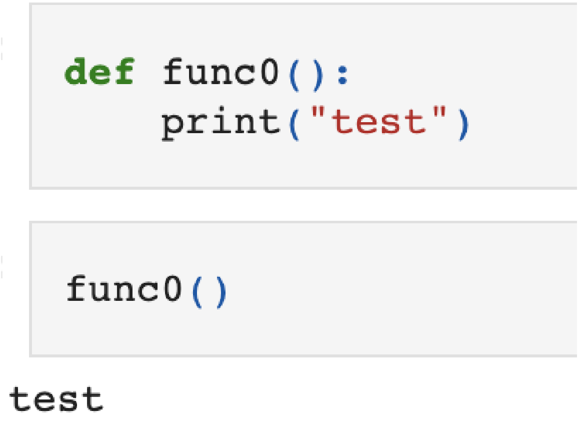
- Logical Operator
  - or
  - and
  - not
- Assignment Operators
  - +=
  - *=
  - /=
  - %=

Modules

- When you install python you get the Python interpreter, its built-in types and function and python standard library
- By using import module we can use it's function in our program.
- To use a module in a Python program it first has to be imported. A module can be imported using the import statement.
- For example, to import the module math, which contains many standard mathematical functions, we can do:
  - import math

Functions

- A function in Python is defined using the keyword def, followed by a function name, a signature within parentheses (), and a colon :.
- The following code, with one additional level of indentation, is the function body.



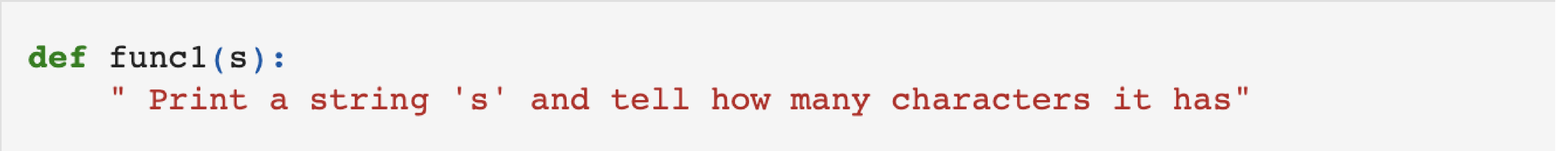
```
def func0():
    print("test")
```

```
func0()
```

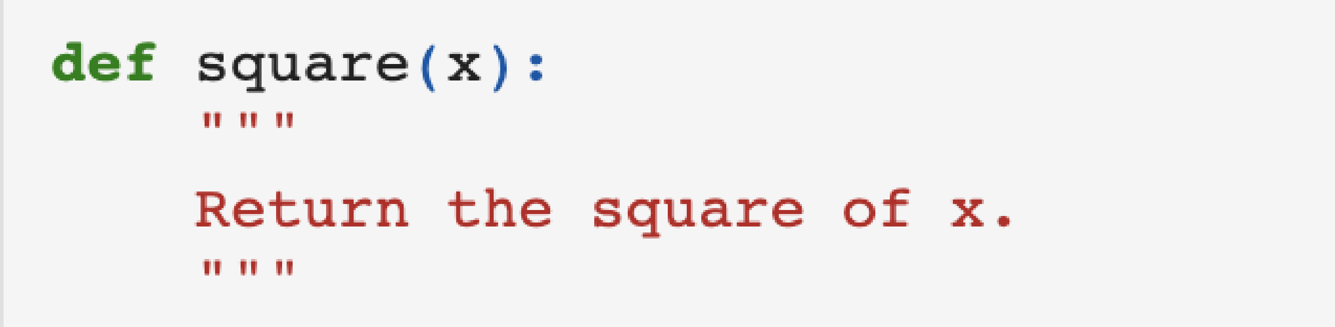
```
test
```

Functions

- Optionally, but highly recommended, we can define a so called "docstring", which is a description of the functions purpose and behaivor.
- The docstring should follow directly after the function definition, before the code in the function body.



```
def func1(s):
    " Print a string 's' and tell how many characters it has"
```

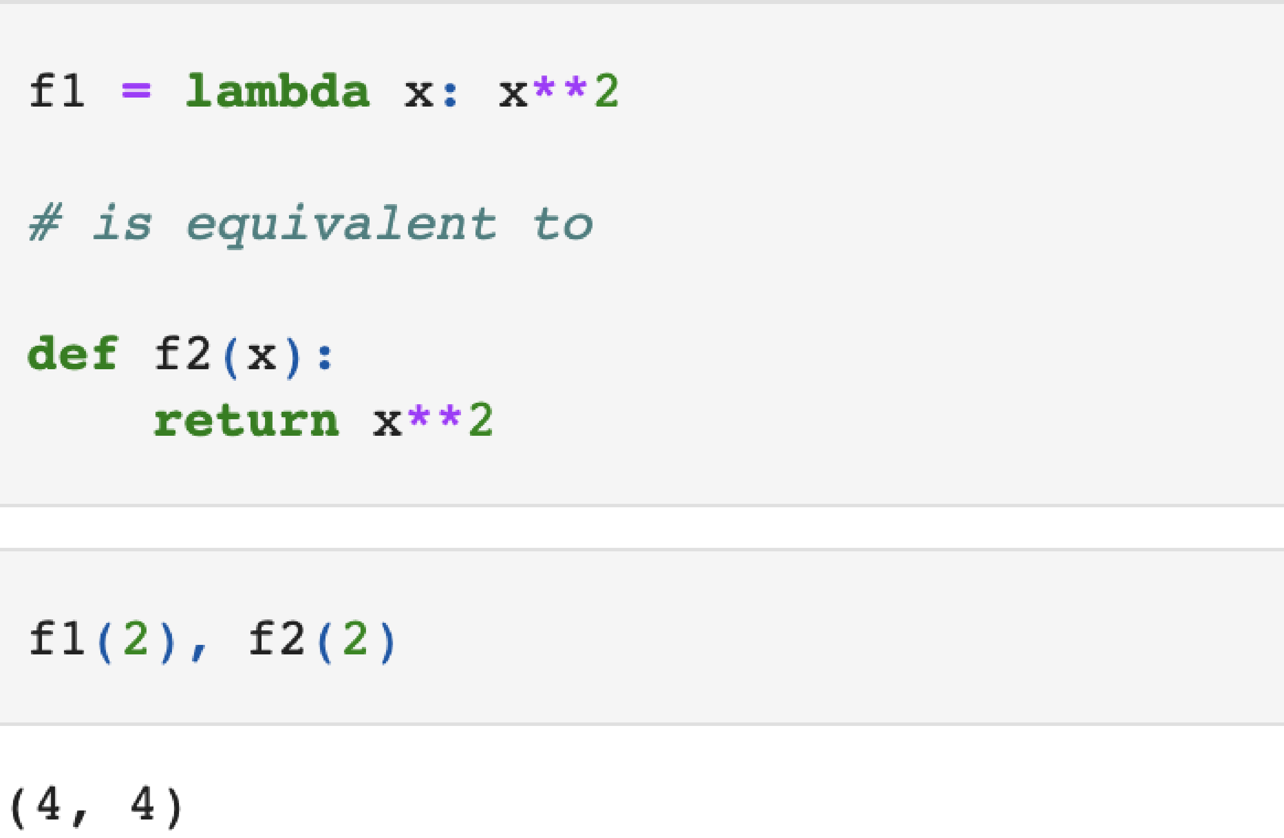


```
def square(x):
    """
    Return the square of x.
    """
```

Functions

- Default argument and keyword argument
    - In a definition of a function, we can give default values to the arguments the function takes:
    - def myfunc(x, p=2, debug=False):
    - If we don't provide a value of the debug argument when calling the the function myfunc it defaults to the value provided in the function definition.
    - myfunc(5,p=4)
    - myfunc(5, debug=True)

Unnamed function (Lambda function)



```
f1 = lambda x: x**2

# is equivalent to

def f2(x):
    return x**2
```

```
f1(2), f2(2)
```

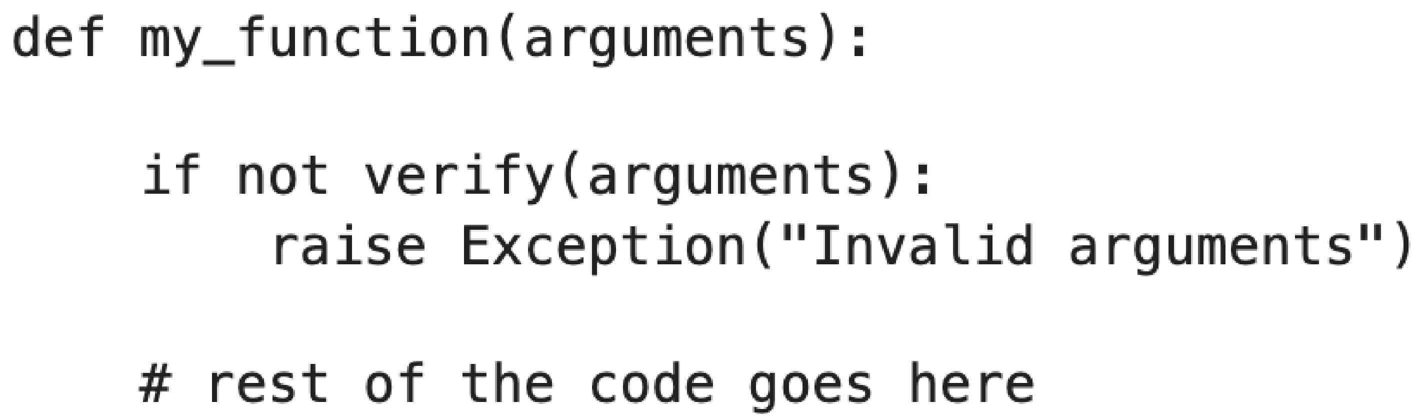
```
(4, 4)
```

Input and Output

- Input and output operations allow interaction between the program and the user.
- Read input from the user and display output using functions like input() and print().

Exceptions

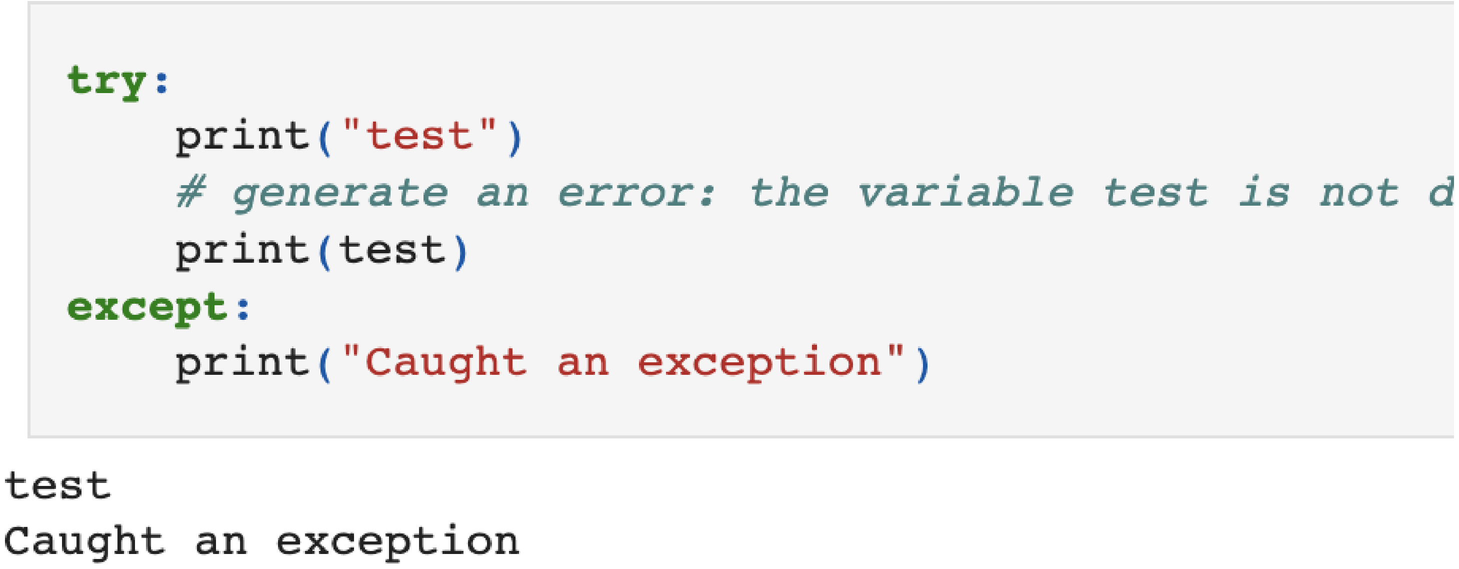
- A typical use of exceptions is to abort functions when some error condition occurs, for example:



```
def my_function(arguments):

    if not verify(arguments):
        raise Exception("Invalid arguments")

    # rest of the code goes here
```

Exceptions

- To gracefully catch errors that are generated by functions and class methods, or by the Python interpreter itself, use the try and except statements:

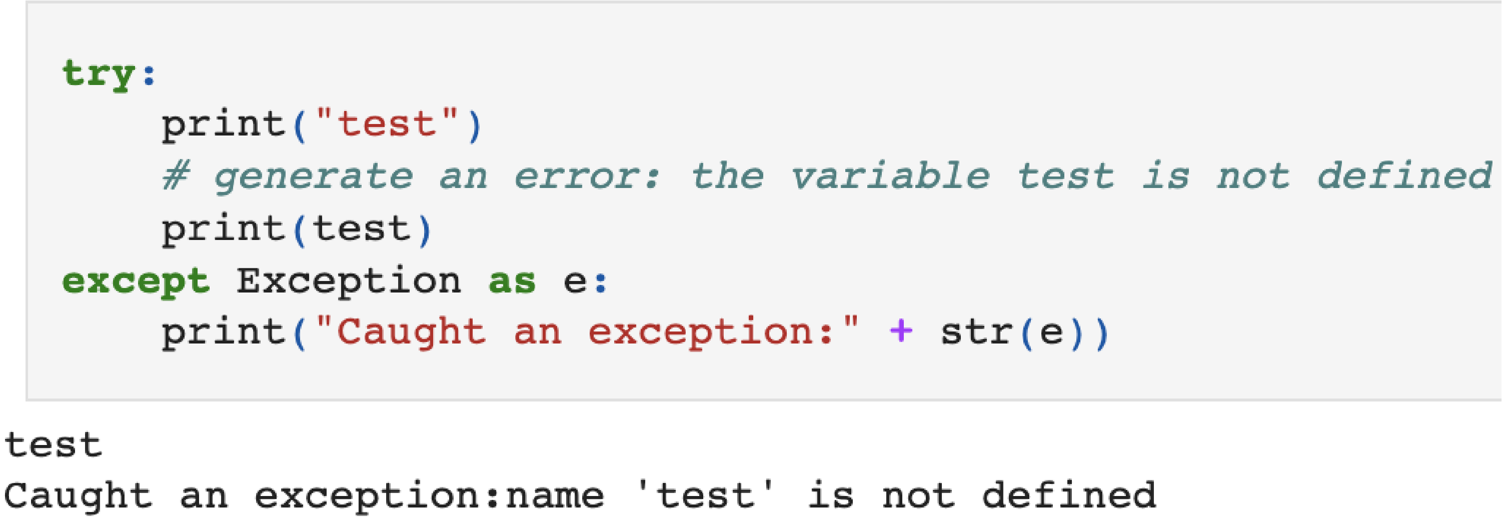


```
try:
    print("test")
    # generate an error: the variable test is not d
    print(test)
except:
    print("Caught an exception")
```

```
test
Caught an exception
```

Exceptions

- To get information about the error, we can access the Exception class instance that describes the exception by using for example: "except Exception as e:"



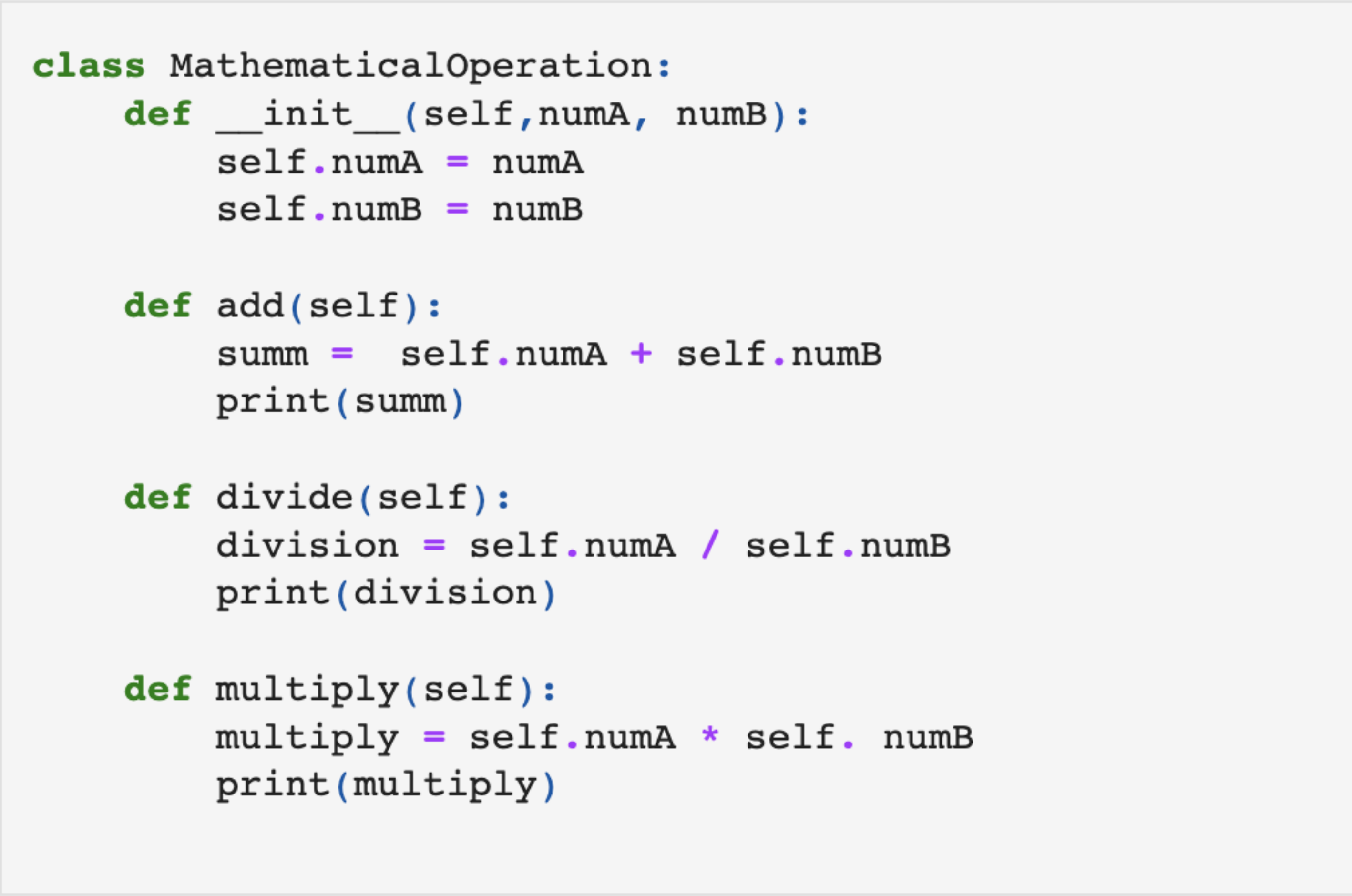
```
try:
    print("test")
    # generate an error: the variable test is not defined
    print(test)
except Exception as e:
    print("Caught an exception:" + str(e))
```

```
test
Caught an exception:name 'test' is not defined
```

Exception Handling

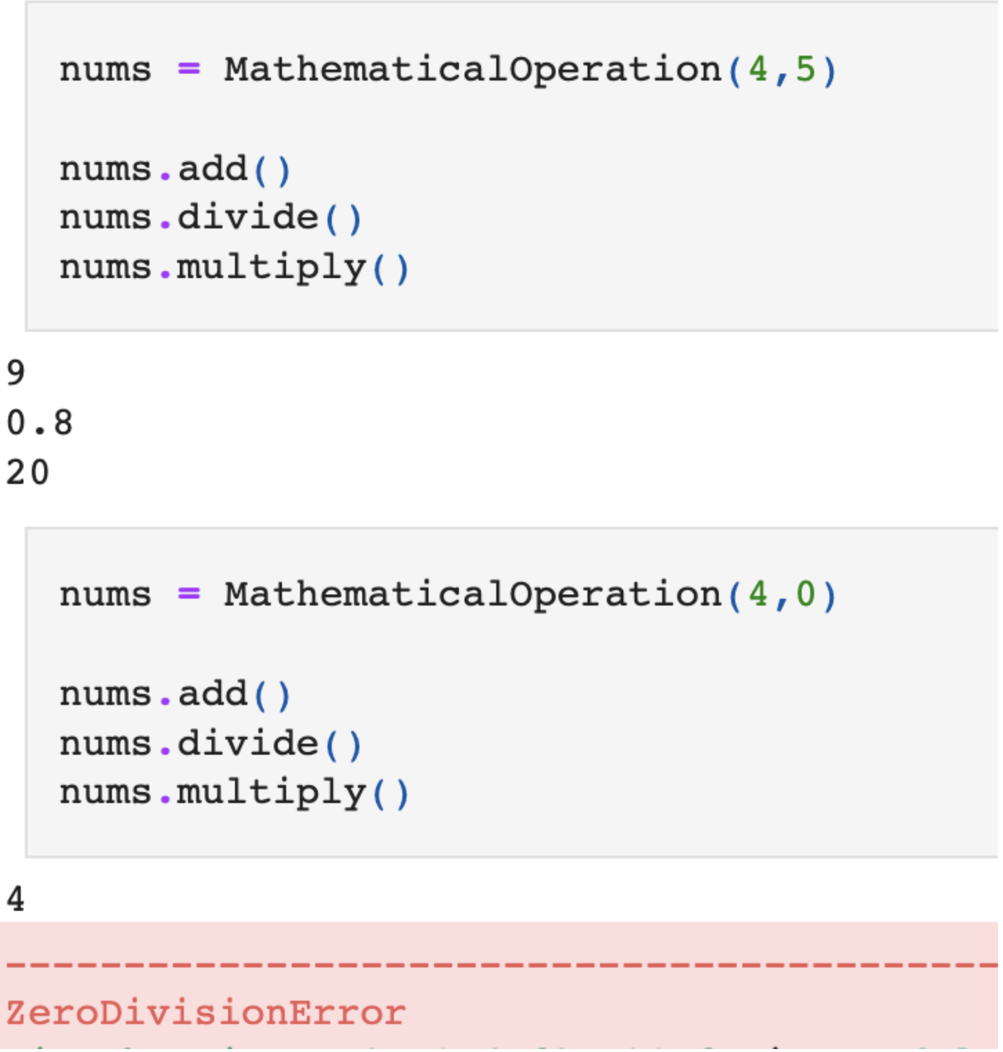
- Mechanism for interrupting normal program flow and continuing in surrounding context
- try .... finally

Exception Handling



```
class MathematicalOperation:
    def __init__(self,numA, numB):
        self.numA = numA
        self.numB = numB

    def add(self):
        summ =  self.numA + self.numB
        print(summ)

    def divide(self):
        division = self.numA / self.numB
        print(division)

    def multiply(self):
        multiply = self.numA * self. numB
        print(multiply)
```

Exception Handling



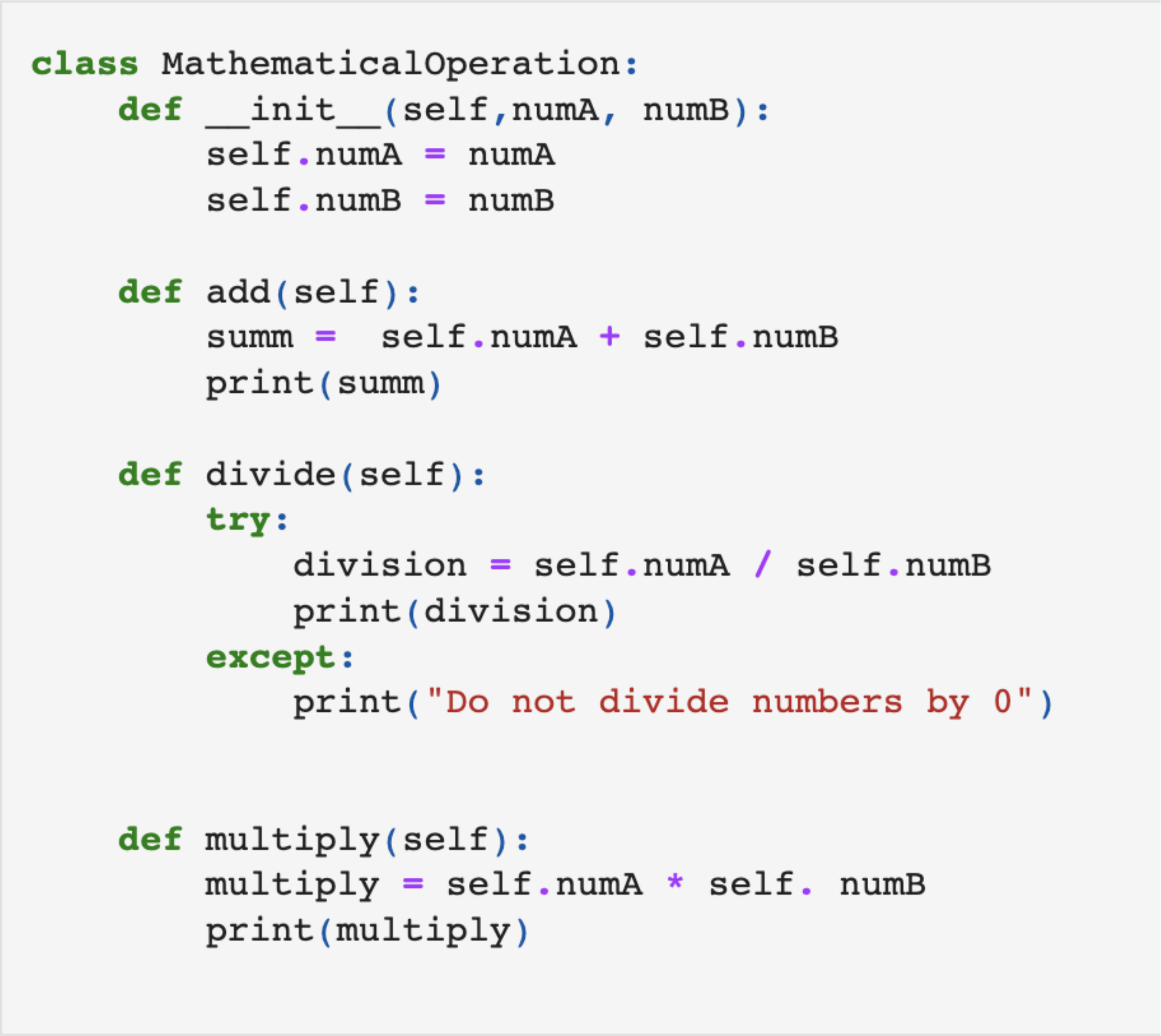
```
nums = MathematicalOperation(4,5)

nums.add()
nums.divide()
nums.multiply()
```

```
9
0.8
20
```

```
nums = MathematicalOperation(4,0)

nums.add()
nums.divide()
nums.multiply()
```

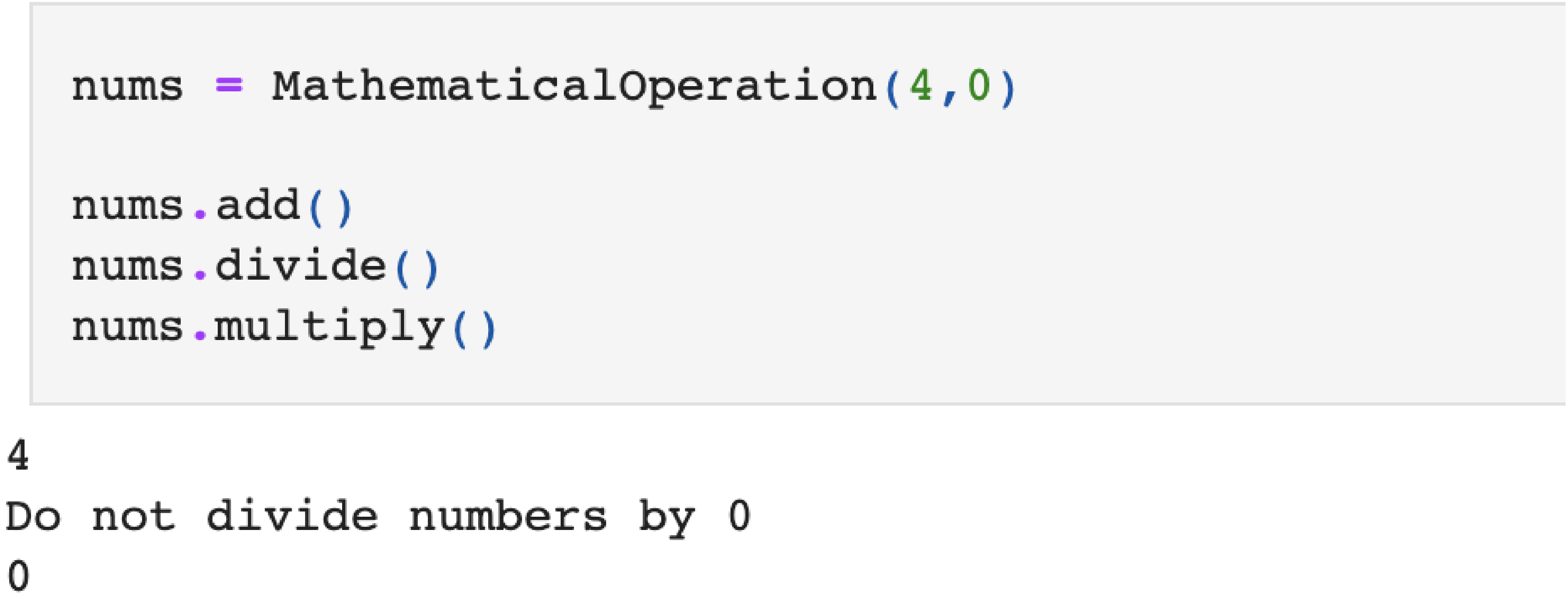
```
4
---------------------------------------------------------------------------
ZeroDivisionError
```

Exception Handling



```python
class MathematicalOperation:
    def __init__(self,numA, numB):
        self.numA = numA
        self.numB = numB

    def add(self):
        summ =  self.numA + self.numB
        print(summ)

    def divide(self):
        try:
            division = self.numA / self.numB
            print(division)
        except:
            print("Do not divide numbers by 0")


    def multiply(self):
        multiply = self.numA * self. numB
        print(multiply)
```

Exception Handling



```python
nums = MathematicalOperation(4,0)

nums.add()
nums.divide()
nums.multiply()
```

```
4
Do not divide numbers by 0
0
```

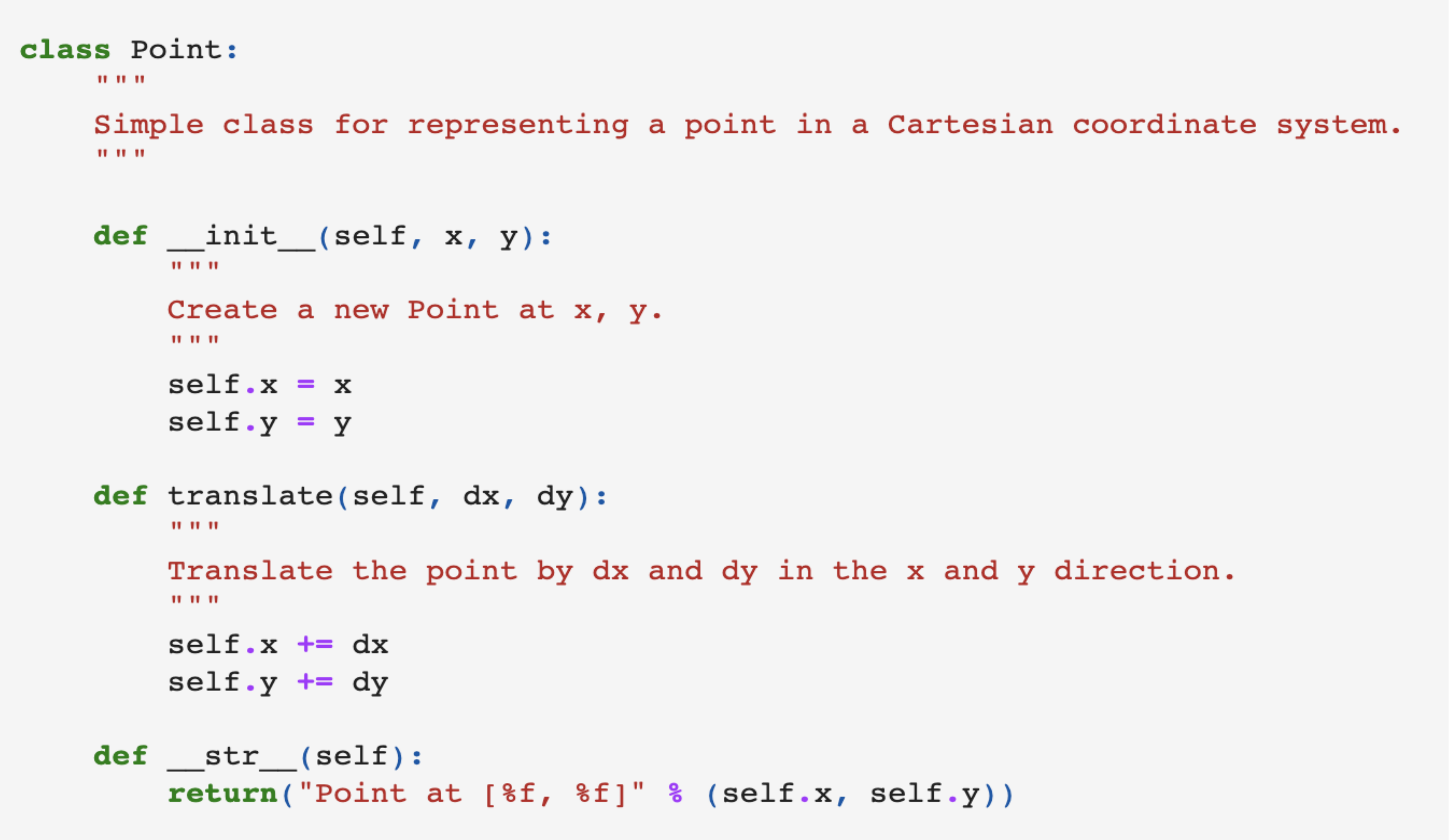
Class

- Define the structure and behavior of objects
- Acts as a template for creating objects
- Classes control an object's initial state, attributes and methods
- Classes are the key features of object-oriented programming. A class is a structure for representing an object and the operations that can be performed on the object.
- In Python a class can contain attributes (variables) and methods (functions).
- A class is defined almost like a function, but using the class keyword, and the class definition usually contains a number of class method definitions (a function in a class).

Class

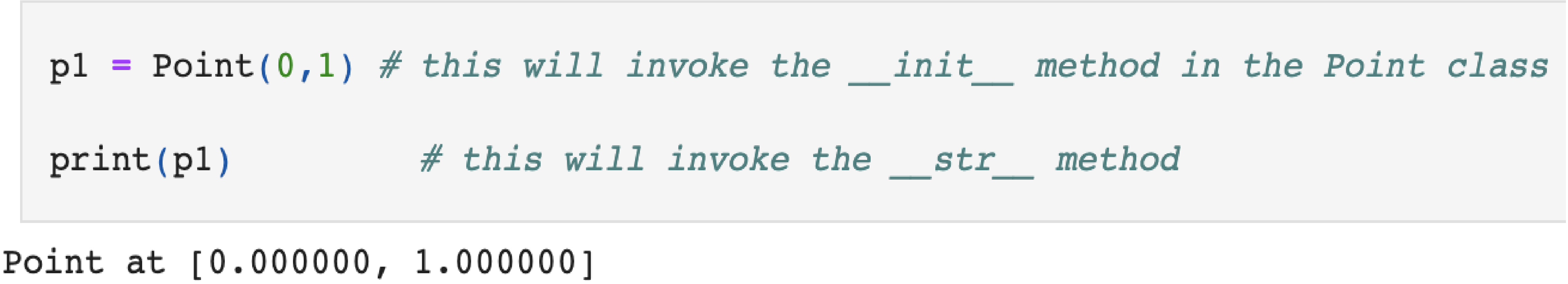
- Each class method should have an argument self as its first argument. This object is a self-reference.
- Some class method names have special meaning, for example:
  - __init__: The name of the method that is invoked when the object is first created.
  - __str__ : A method that is invoked when a simple string representation of the class is needed, as for example when printed.

Class (Example)



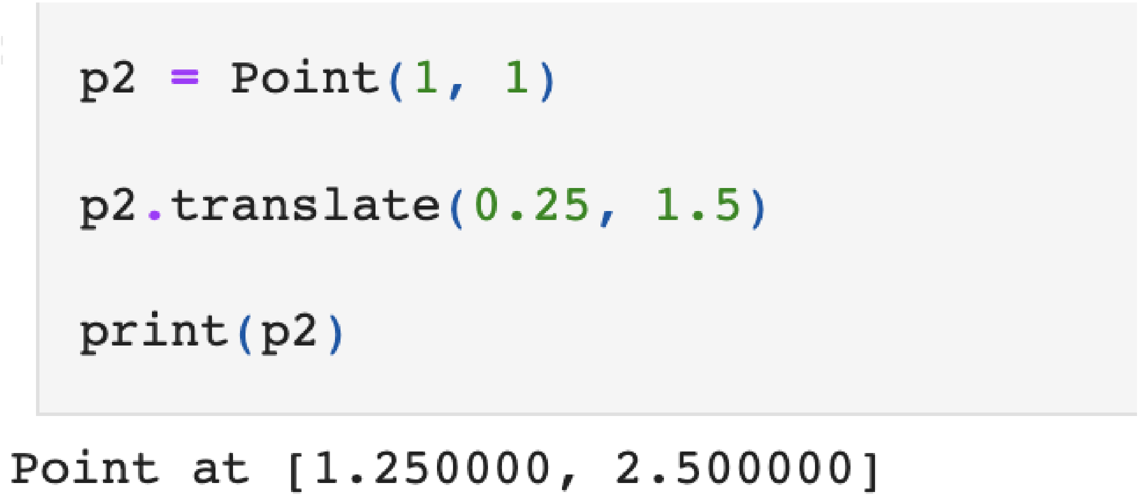
```
class Point:
    """
    Simple class for representing a point in a Cartesian coordinate system.
    """

    def __init__(self, x, y):
        """
        Create a new Point at x, y.
        """
        self.x = x
        self.y = y

    def translate(self, dx, dy):
        """
        Translate the point by dx and dy in the x and y direction.
        """
        self.x += dx
        self.y += dy

    def __str__(self):
        return("Point at [%f, %f]" % (self.x, self.y))
```

Class (Example)



```
p1 = Point(0,1) # this will invoke the __init__ method in the Point class
print(p1)         # this will invoke the __str__ method
```

```
Point at [0.000000, 1.000000]
```



```
p2 = Point(1, 1)
p2.translate(0.25, 1.5)
print(p2)
```

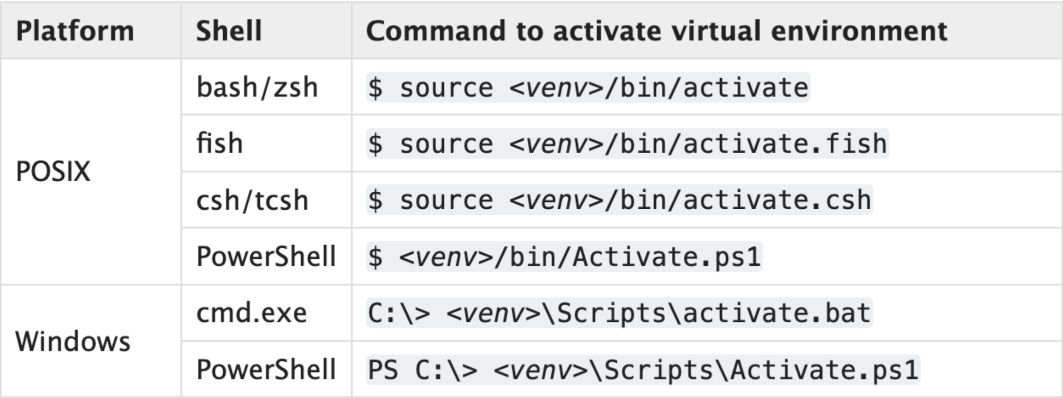
```
Point at [1.250000, 2.500000]
```

Virtual Environments

- A virtual environment is created on top of an existing Python installation, known as the virtual environment’s “base” Python.
- May optionally be isolated from the packages in the base environment, so only those explicitly installed in the virtual environment are available.
- Refer: https://docs.python.org/3/library/venv.html

Virtual Environments

- Creating virtual environments
  - python -m venv /path/to/new/virtual/environment.
- The invocation of the activation script is platform-specific (<venv> must be replaced by the path to the directory containing the virtual environment):



Platform	Shell	Command to activate virtual environment
POSIX	bash/zsh	`$ source <venv>/bin/activate`
	fish	`$ source <venv>/bin/activate.fish`
	csh/tcsh	`$ source <venv>/bin/activate.csh`
	PowerShell	`$ <venv>/bin/Activate.ps1`
Windows	cmd.exe	`C:\> <venv>\Scripts\activate.bat`
	PowerShell	`PS C:\> <venv>\Scripts\Activate.ps1`

Installing Packages

- Installing packages
  - pip install packageName
  - pip install packageName==packageVersion
  - pip install --user packageName
  - pip install --user packageName==packageVersion
- Install from requirement file
  - pip install -r requirements.txt
- Refer: https://packaging.python.org/en/latest/tutorials/installing-packages/