Introduction to Programming Using Python

Microsoft Python Certification | **Exam 98-381**

Lecture 6:

Object Oriented Programming

Lecturer: **James W. Jiang**, Ph.D. | Summer, 2018

Microsoft™

SAVVY PRO
PROFESSIONAL EDUCATION TRAINING

Object-Oriented Programming: **Introduction**

OOP

Python is a multi-paradigm programming language. Meaning, it supports different programming approach.

One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

- attributes
- behavior

Example: OOP

Let's take an example:

Parrot is an object,

- name, age, color are attributes
- singing, dancing are behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

Basic Principles

In Python, the concept of OOP follows some basic principles:

Inheritance	A process of using details from a new class without modifying existing class.
Encapsulation	Hiding the private details of a class from other objects.
Polymorphism	A concept of using common operation in different ways for different data input.

Object-Oriented Programming: **Class**

Class

A class is a blueprint for the object.

We can think of class as an sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, parrot is an object.

The example for class of parrot can be :

```
class Parrot:
    pass
```

Here, we use class keyword to define an empty class Parrot. From class, we construct instances. An instance is a specific object created from a particular class.

Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

```
obj = Parrot()
```

```
Here, obj is object of class Parrot.
```

Suppose we have details of parrot. Now, we are going to show how to build the class and objects of parrot.

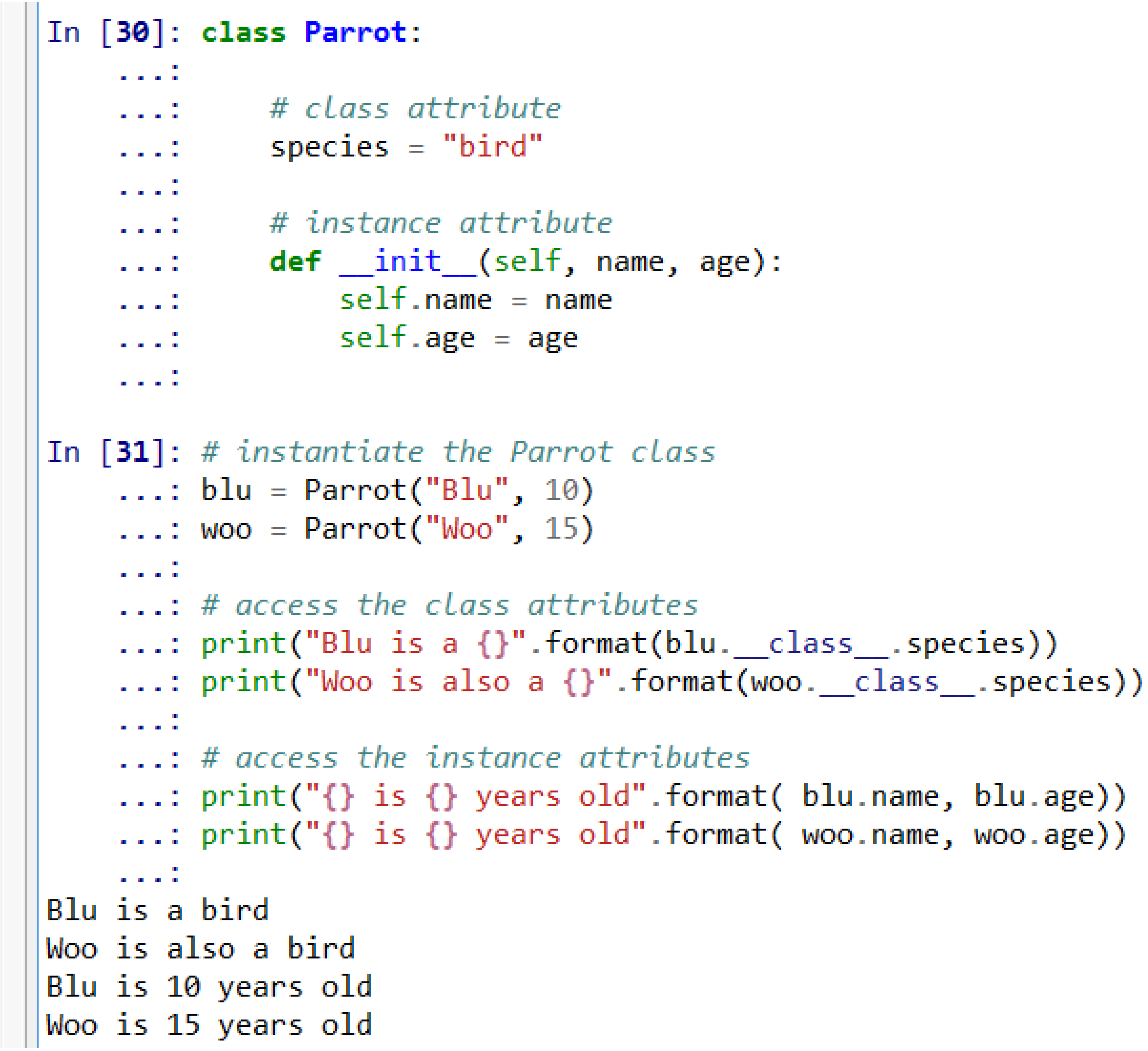
Example: Creating Class and Object in Python

```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Example: Creating Class and Object in Python (cont`d)

```
# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

Example: Creating Class and Object in Python (cont`d)



```
In [30]: class Parrot:
    ...:
    ...:     # class attribute
    ...:     species = "bird"
    ...:
    ...:     # instance attribute
    ...:     def __init__(self, name, age):
    ...:         self.name = name
    ...:         self.age = age
    ...:

In [31]: # instantiate the Parrot class
    ...: blu = Parrot("Blu", 10)
    ...: woo = Parrot("Woo", 15)
    ...:
    ...: # access the class attributes
    ...: print("Blu is a {}".format(blu.__class__.species))
    ...: print("Woo is also a {}".format(woo.__class__.species))
    ...:
    ...: # access the instance attributes
    ...: print("{} is {} years old".format( blu.name, blu.age))
    ...: print("{} is {} years old".format( woo.name, woo.age))
    ...:
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

Docstring

The first string is called docstring and has a brief description about the class. Although not mandatory, this is recommended.

Here is a simple class definition.

```
class MyNewClass:
    '''This is a docstring. I have created a new 
class'''
    pass
```

There are also special attributes in it that begins with double underscores (__). For example, __doc__ gives us the docstring of that class.

Docstring

```
class MyClass:
      "This is my second class"
      a = 10
      def func(self):
            print('Hello')
# Output: 10
print(MyClass.a)
# Output: <function MyClass.func at 0x0000000003079BF8>
print(MyClass.func)
# Output: 'This is my second class'
print(MyClass.__doc__)
```

Constructors in Python

Class functions that begins with double underscore (`__`) are called special functions as they have special meaning.

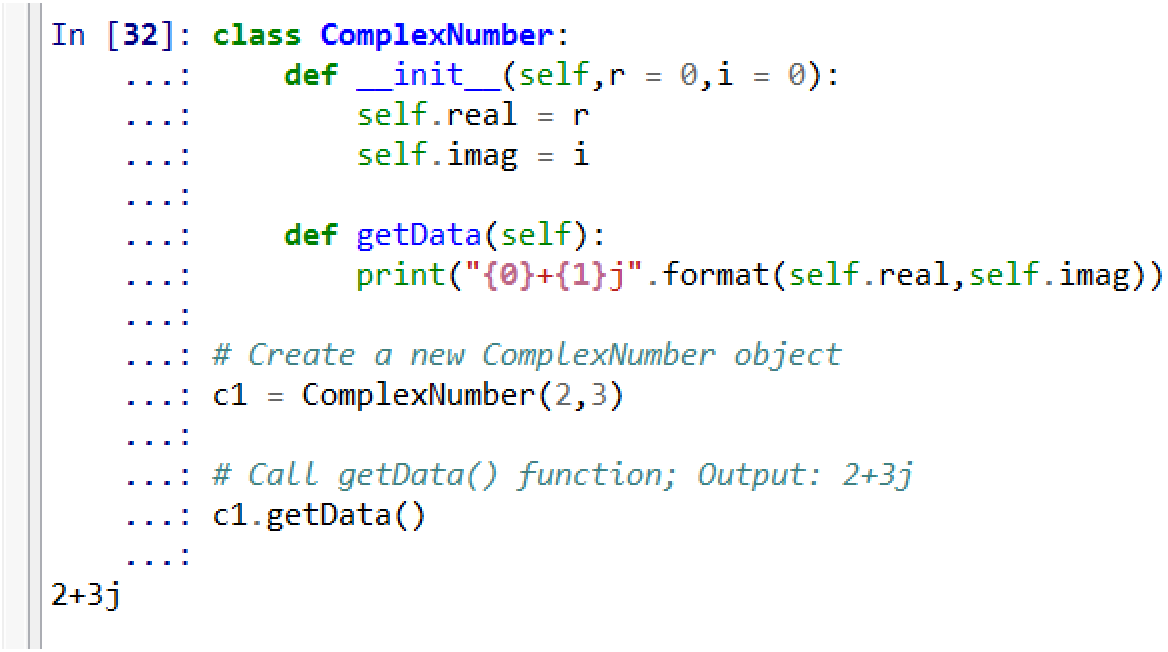
Of one particular interest is the `__init__()` function. This special function gets called whenever a new object of that class is instantiated.

This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

Example: Complex number

```
class ComplexNumber:
    def __init__(self,r = 0,i = 0):
        self.real = r
        self.imag = i

    def getData(self):
        print("{0}+{1}j".format(self.real,self.imag))

# Create a new ComplexNumber object
c1 = ComplexNumber(2,3)

# Call getData() function; Output: 2+3j
c1.getData()
```

Example: Complex number (cont'd)

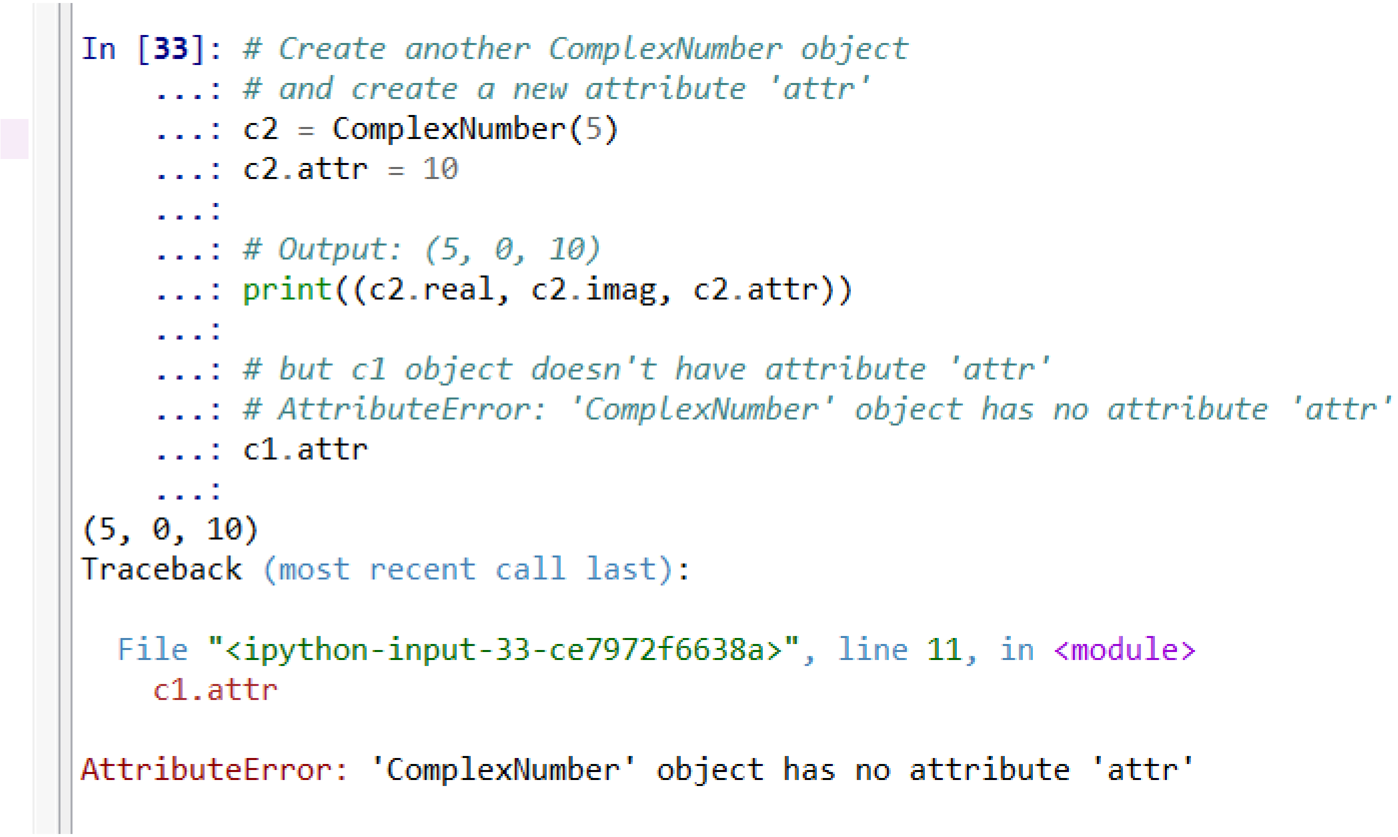


```
In [32]: class ComplexNumber:
    ...:     def __init__(self,r = 0,i = 0):
    ...:         self.real = r
    ...:         self.imag = i
    ...:
    ...:     def getData(self):
    ...:         print("{0}+{1}j".format(self.real,self.imag))
    ...:
    ...: # Create a new ComplexNumber object
    ...: c1 = ComplexNumber(2,3)
    ...:
    ...: # Call getData() function; Output: 2+3j
    ...: c1.getData()
    ...:
2+3j
```

Example: Complex number (cont'd)

```
# Create another ComplexNumber object
# and create a new attribute 'attr'
c2 = ComplexNumber(5)
c2.attr = 10

# Output: (5, 0, 10)
print((c2.real, c2.imag, c2.attr))

# but c1 object doesn't have attribute 'attr'
# AttributeError: 'ComplexNumber' object has no
attribute 'attr'
c1.attr
```

Example: Complex number (cont'd)



```
In [33]: # Create another ComplexNumber object
    ...: # and create a new attribute 'attr'
    ...: c2 = ComplexNumber(5)
    ...: c2.attr = 10
    ...:
    ...: # Output: (5, 0, 10)
    ...: print((c2.real, c2.imag, c2.attr))
    ...:
    ...: # but c1 object doesn't have attribute 'attr'
    ...: # AttributeError: 'ComplexNumber' object has no attribute 'attr'
    ...: c1.attr
    ...:
(5, 0, 10)
Traceback (most recent call last):

  File "<ipython-input-33-ce7972f6638a>", line 11, in <module>
    c1.attr

AttributeError: 'ComplexNumber' object has no attribute 'attr'
```

Example: Complex number (cont'd)

In the above example, we define a new class to represent complex numbers. It has two functions, __init__() to initialize the variables (defaults to zero) and getData() to display the number properly.

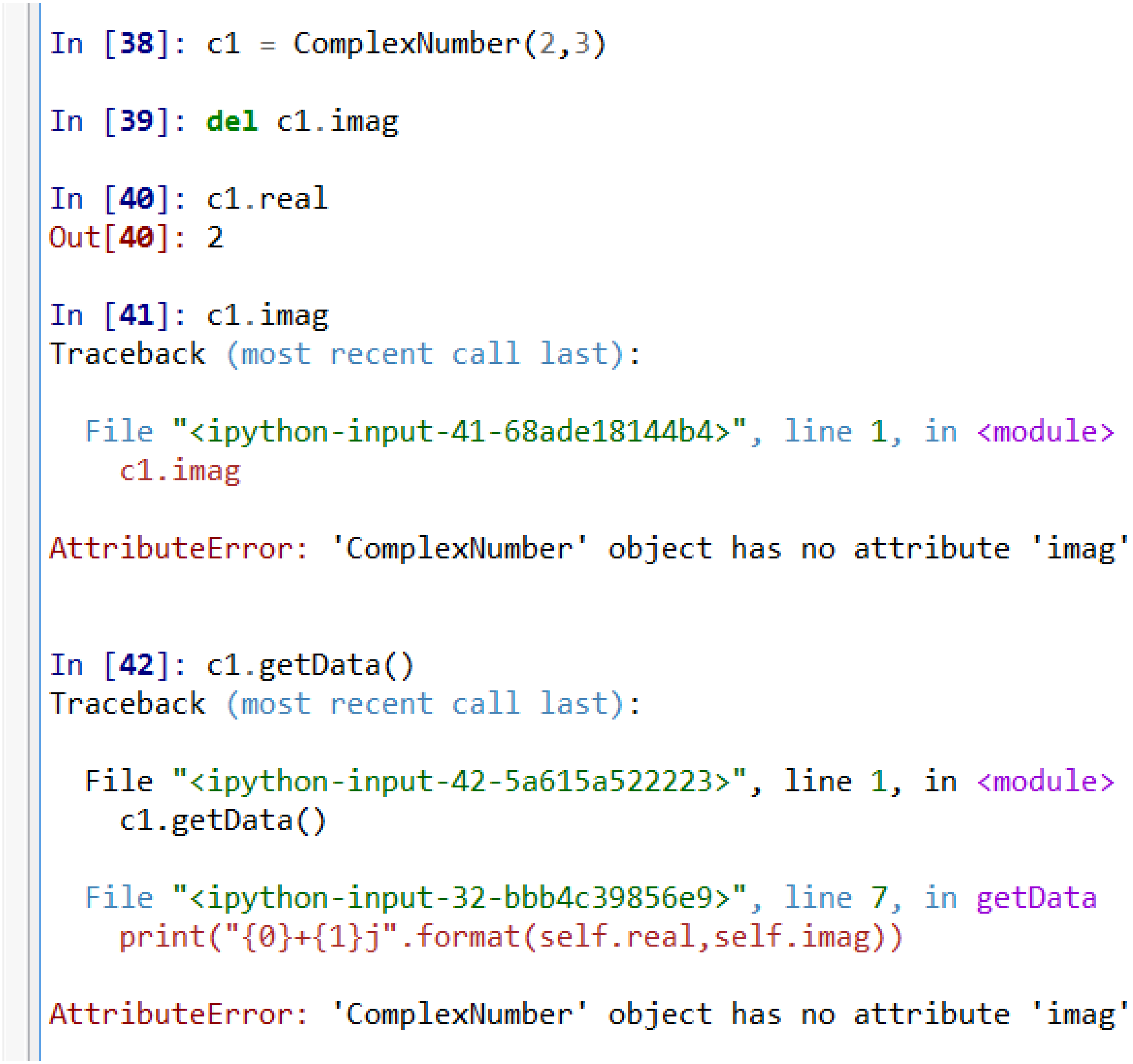
An interesting thing to note in the above step is that attributes of an object can be created on the fly. We created a new attribute attr for object c2 and we read it as well. But this did not create that attribute for object c1.

Deleting Attributes

Any attribute of an object can be deleted anytime, using the del statement. Try the following on the Python shell to see the output.

```
c1 = ComplexNumber(2,3)

del c1.imag

c1.real

c1.imag

c1.getData()
```

Deleting Attributes



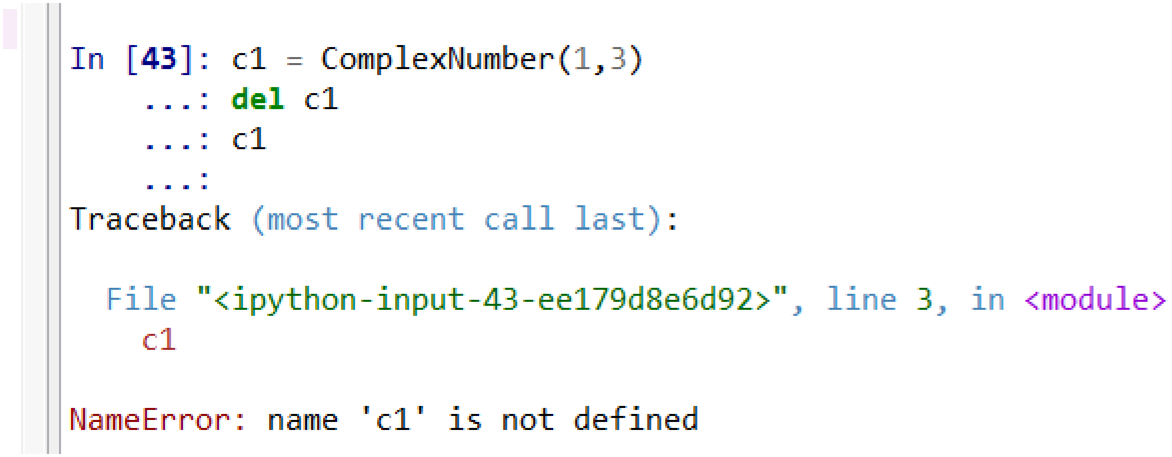
```
In [38]: c1 = ComplexNumber(2,3)

In [39]: del c1.imag

In [40]: c1.real
Out[40]: 2

In [41]: c1.imag
Traceback (most recent call last):

  File "<ipython-input-41-68ade18144b4>", line 1, in <module>
    c1.imag

AttributeError: 'ComplexNumber' object has no attribute 'imag'


In [42]: c1.getData()
Traceback (most recent call last):

  File "<ipython-input-42-5a615a522223>", line 1, in <module>
    c1.getData()

  File "<ipython-input-32-bbb4c39856e9>", line 7, in getData
    print("{0}+{1}j".format(self.real,self.imag))

AttributeError: 'ComplexNumber' object has no attribute 'imag'
```

Deleting Objects

We can even delete the object itself, using the del statement.

```
c1 = ComplexNumber(1,3)

del c1

c1
```

Deleting Objects



```
In [43]: c1 = ComplexNumber(1,3)
    ...: del c1
    ...: c1
    ...:
Traceback (most recent call last):

  File "<ipython-input-43-ee179d8e6d92>", line 3, in <module>
    c1

NameError: name 'c1' is not defined
```

Object-Oriented Programming: **Methods**

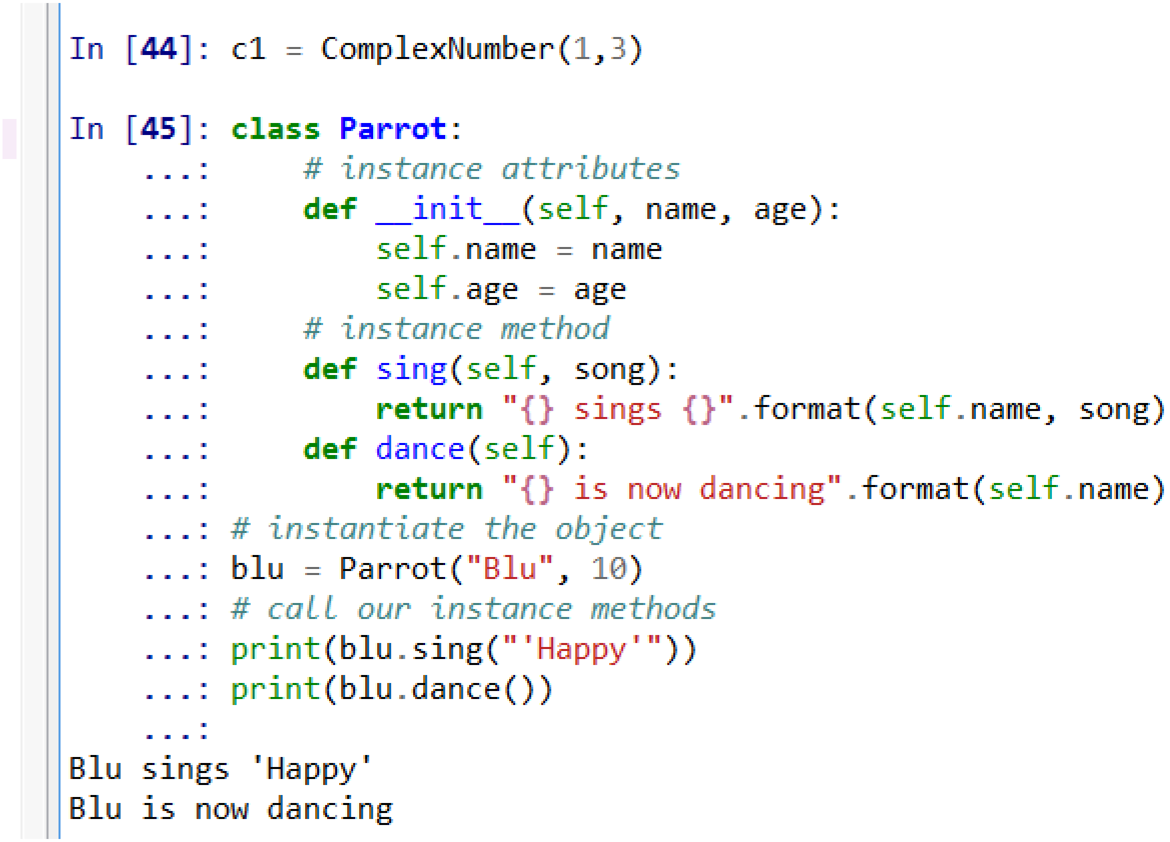
Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

Example: Creating Methods in Python

```
class Parrot:
    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)
    def dance(self):
        return "{} is now dancing".format(self.name)
# instantiate the object
blu = Parrot("Blu", 10)
# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

Example: Creating Methods in Python



```
In [44]: c1 = ComplexNumber(1,3)

In [45]: class Parrot:
    ...:     # instance attributes
    ...:     def __init__(self, name, age):
    ...:         self.name = name
    ...:         self.age = age
    ...:     # instance method
    ...:     def sing(self, song):
    ...:         return "{} sings {}".format(self.name, song)
    ...:     def dance(self):
    ...:         return "{} is now dancing".format(self.name)
    ...: # instantiate the object
    ...: blu = Parrot("Blu", 10)
    ...: # call our instance methods
    ...: print(blu.sing("'Happy'"))
    ...: print(blu.dance())
    ...:
Blu sings 'Happy'
Blu is now dancing
```

Object-Oriented Programming:
Inheritance

Derived Class

Inheritance is a way of creating new class for using details of existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

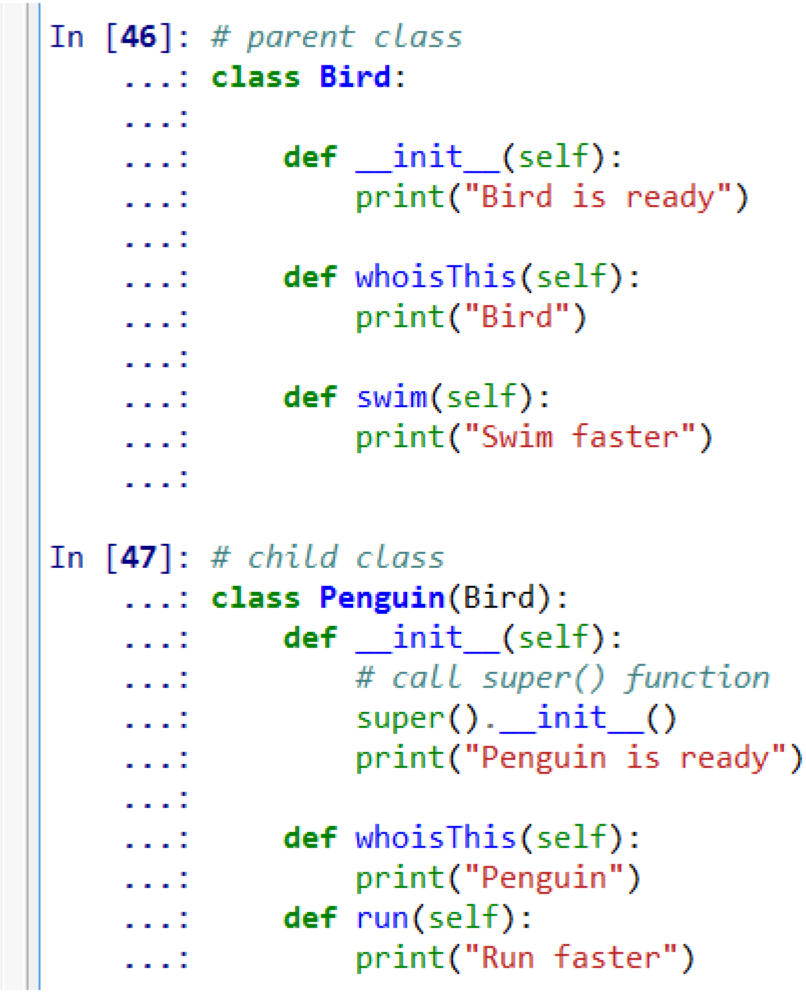
Example: Use of Inheritance in Python

```
# parent class
class Bird:

    def __init__(self):
        print("Bird is ready")

    def whoisThis(self):
        print("Bird")

    def swim(self):
        print("Swim faster")
```

Example: Use of Inheritance in Python (cont`d)

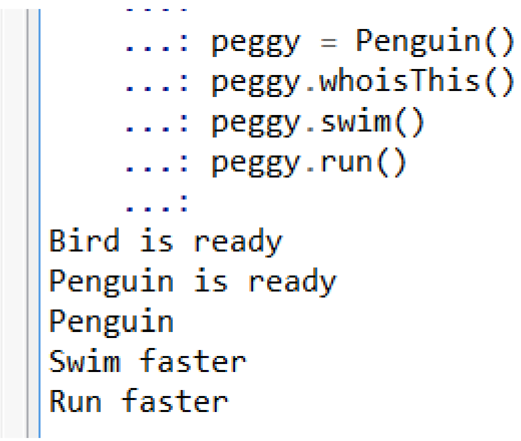
```
# child class
class Penguin(Bird):
    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")

    def whoisThis(self):
        print("Penguin")
    def run(self):
        print("Run faster")

peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

Example: Use of Inheritance in Python (cont`d)



```
In [46]: # parent class
    ...: class Bird:
    ...:
    ...:     def __init__(self):
    ...:         print("Bird is ready")
    ...:
    ...:     def whoisThis(self):
    ...:         print("Bird")
    ...:
    ...:     def swim(self):
    ...:         print("Swim faster")
    ...:

In [47]: # child class
    ...: class Penguin(Bird):
    ...:     def __init__(self):
    ...:         # call super() function
    ...:         super().__init__()
    ...:         print("Penguin is ready")
    ...:
    ...:     def whoisThis(self):
    ...:         print("Penguin")
    ...:     def run(self):
    ...:         print("Run faster")
```

Example: Use of Inheritance in Python (cont`d)



```
    ....
    ...: peggy = Penguin()
    ...: peggy.whoisThis()
    ...: peggy.swim()
    ...: peggy.run()
    ...:
Bird is ready
Penguin is ready
Penguin
Swim faster
Run faster
```

Example: Use of Inheritance in Python (cont`d)

In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from swim() method. Again, the child class modified the behavior of parent class. We can see this from whoisThis() method. Furthermore, we extend the functions of parent class, by creating a new run() method.

Additionally, we use super() function before __init__() method. This is because we want to pull the content of __init__() method from the parent class into the child class.

Object-Oriented Programming:
Encapsulation

Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e single " _ " or double " __".

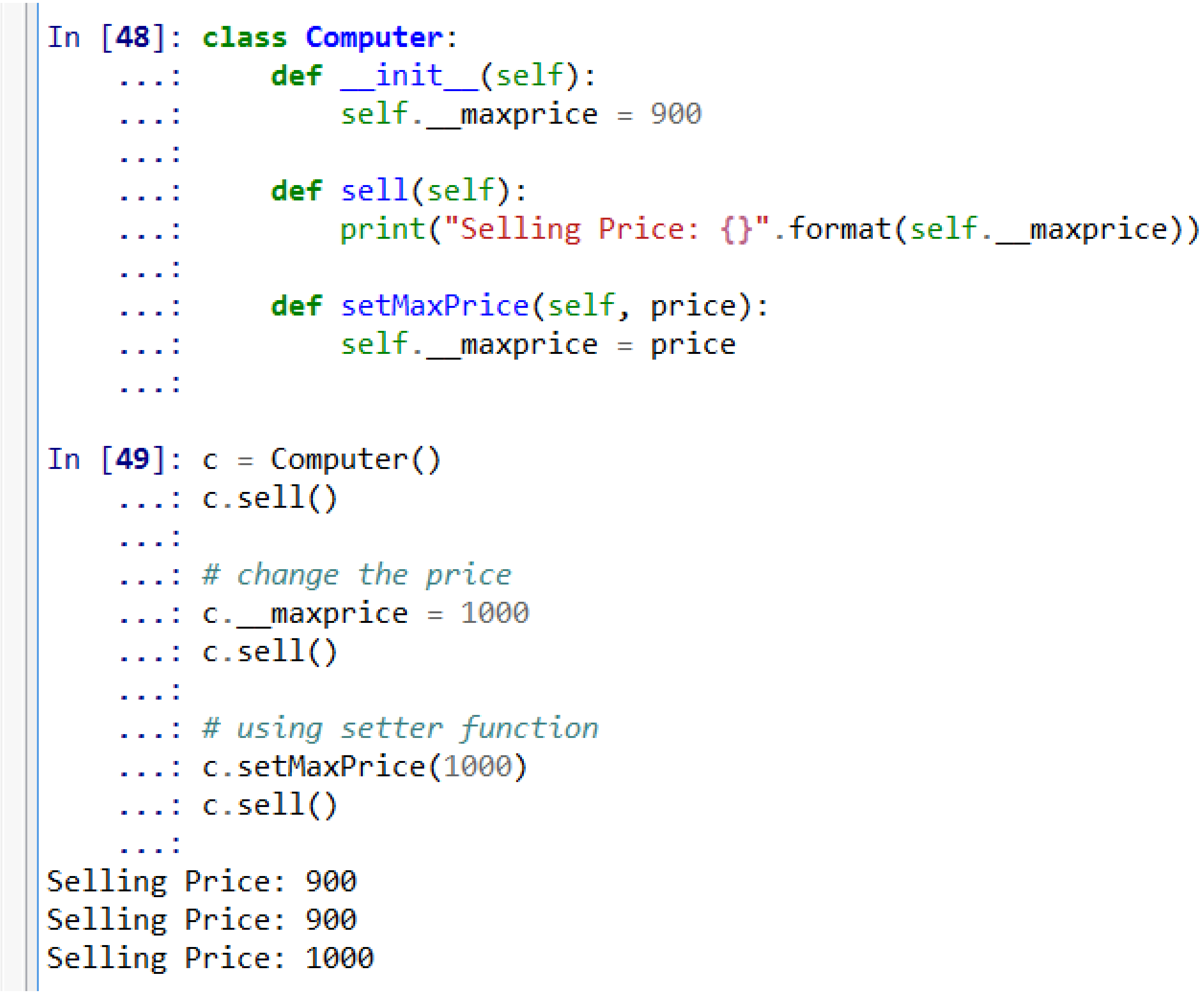
Example: Data Encapsulation in Python

```python
class Computer:
    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):
        self.__maxprice = price
```

Example: Data Encapsulation in Python (cont’d)

```
c = Computer()
c.sell()

# change the price
c.__maxprice = 1000
c.sell()

# using setter function
c.setMaxPrice(1000)
c.sell()
```

Example: Data Encapsulation in Python (cont'd)



```
In [48]: class Computer:
    ...:     def __init__(self):
    ...:         self.__maxprice = 900
    ...:
    ...:     def sell(self):
    ...:         print("Selling Price: {}".format(self.__maxprice))
    ...:
    ...:     def setMaxPrice(self, price):
    ...:         self.__maxprice = price
    ...:

In [49]: c = Computer()
    ...: c.sell()
    ...:
    ...: # change the price
    ...: c.__maxprice = 1000
    ...: c.sell()
    ...:
    ...: # using setter function
    ...: c.setMaxPrice(1000)
    ...: c.sell()
    ...:
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

Example: Data Encapsulation in Python (cont’d)

In the above program, we defined a class Computer. We use __init__() method to store the maximum selling price of computer. We tried to modify the price. However, we can’t change it because Python treats the __maxprice as private attributes. To change the value, we used a setter function i.e setMaxPrice() which takes price as parameter.

Object-Oriented Programming: **Polymorphism**

Polymorphism

Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).

Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called Polymorphism.

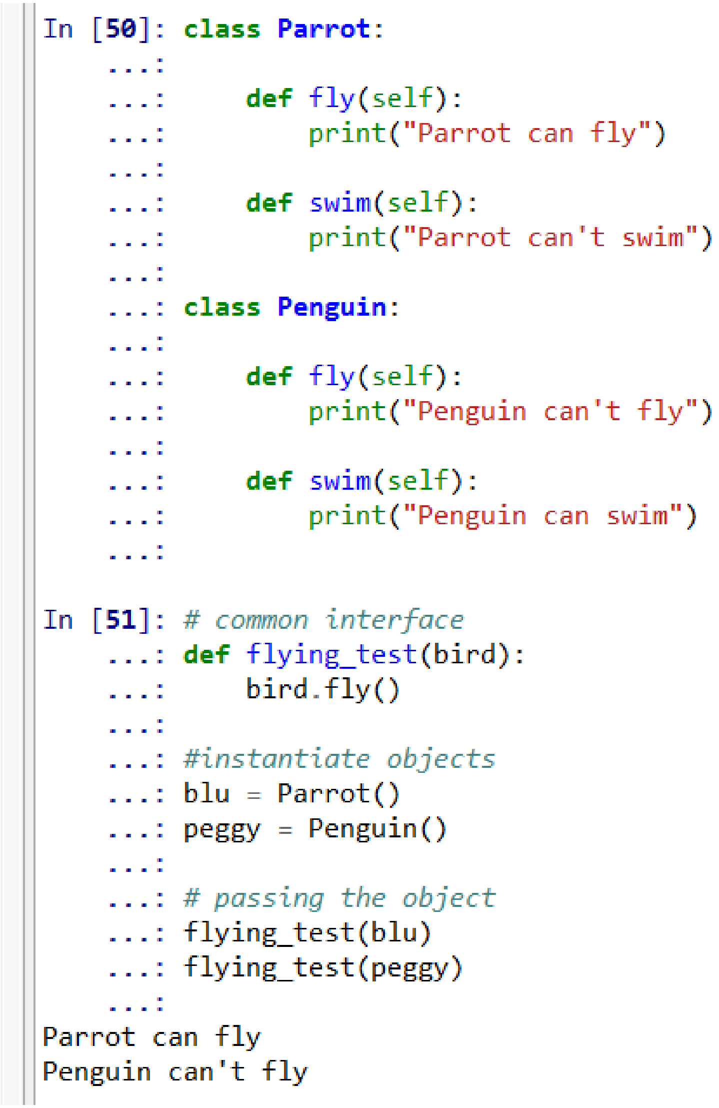
Example: Using Polymorphism in Python

```
class Parrot:

    def fly(self):
        print("Parrot can fly")

    def swim(self):
        print("Parrot can't swim")

class Penguin:

    def fly(self):
        print("Penguin can't fly")

    def swim(self):
        print("Penguin can swim")
```

Example: Using Polymorphism in Python (cont'd)

```
# common interface
def flying_test(bird):
    bird.fly()

#instantiate objects
blu = Parrot()
peggy = Penguin()

# passing the object
flying_test(blu)
flying_test(peggy)
```

Example: Using Polymorphism in Python (cont'd)



```
In [50]: class Parrot:
    ...:
    ...:     def fly(self):
    ...:         print("Parrot can fly")
    ...:
    ...:     def swim(self):
    ...:         print("Parrot can't swim")
    ...:
    ...: class Penguin:
    ...:
    ...:     def fly(self):
    ...:         print("Penguin can't fly")
    ...:
    ...:     def swim(self):
    ...:         print("Penguin can swim")
    ...:

In [51]: # common interface
    ...: def flying_test(bird):
    ...:     bird.fly()
    ...:
    ...: #instantiate objects
    ...: blu = Parrot()
    ...: peggy = Penguin()
    ...:
    ...: # passing the object
    ...: flying_test(blu)
    ...: flying_test(peggy)
    ...:
Parrot can fly
Penguin can't fly
```

Example: Using Polymorphism in Python (cont'd)

In the above program, we defined two classes Parrot and Penguin. Each of them have common method fly() method. However, their functions are different. To allow polymorphism, we created common interface i.e flying_test() function that can take any object. Then, we passed the objects blu and peggy in the flying_test() function, it ran effectively.

Object-Oriented Programming:
Multiple Inheritance

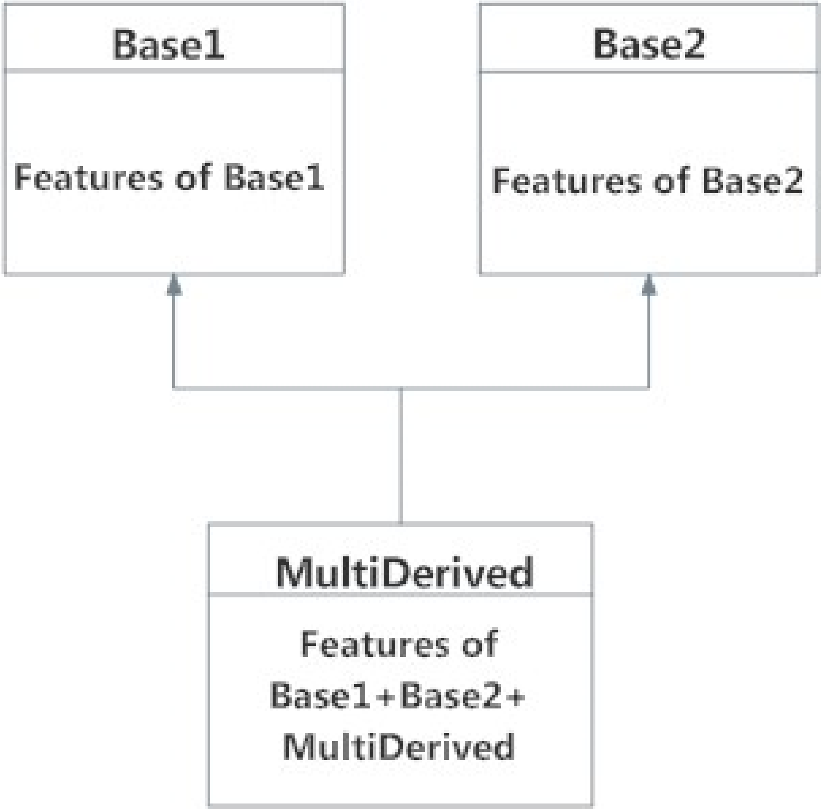
Multiple Inheritance

Like C++, a class can be derived from more than one base classes in Python. This is called multiple inheritance.

In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single inheritance.

```
class Base1:

    pass

class Base2:

    pass

class MultiDerived(Base1, Base2):

    pass
```

Multiple Inheritance

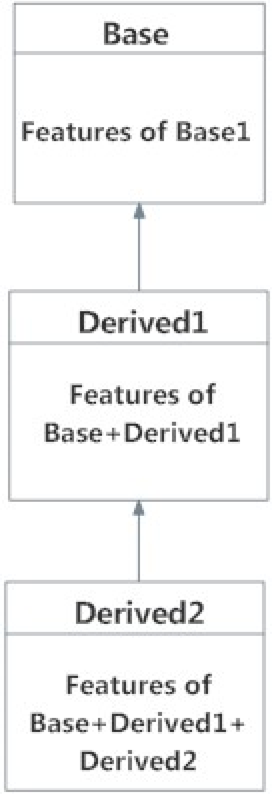


Multilevel Inheritance

On the other hand, we can also inherit form a derived class. This is called multilevel inheritance. It can be of any depth in Python. In multilevel inheritance, features of the base class and the derived class is inherited into the new derived class. An example with corresponding visualization is given below.

```
class Base:
    pass
class Derived1(Base):
    pass
class Derived2(Derived1):
    pass
```

Multiple Inheritance


Base
Features of Base1
Derived1
Features of Base+Derived1
Derived2
Features of Base+Derived1+ Derived2

Formatting:

Python Shortcuts

Shortcuts

ctrl + N	Open a new file
F9	Run current line
Ctrl + 1	Comment/uncomment line
Ctrl 4, ctrl 5	Comment uncomment (block)
Ctrl + L	Clean console
Tab or shift tab	Indent or unindent
F11	Full screen, or no

Python Keywords

In Python, keywords are case sensitive. There are 33 keywords in Python 3.3. This number can vary slightly in course of time. All the keywords except True, False and None are in lowercase and they must be written as it is. The list of all the keywords are given below.

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Formatting:

Lines and Indentation

Python Blocks

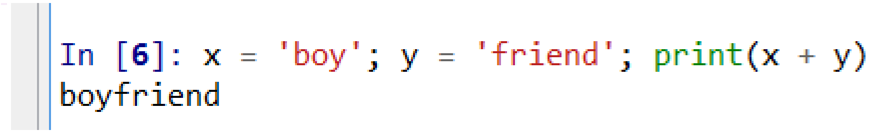
Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.

Multiple Statements on a Single Line

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon

```
x = 'boy'; y = 'friend'; print(x + y)
```



```
In [6]: x = 'boy'; y = 'friend'; print(x + y)
boyfriend
```

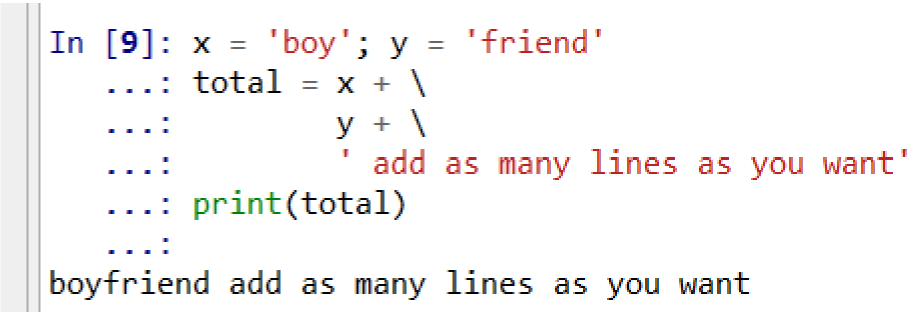
Formatting:

Multi-Line Statements

Line Continuation

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example

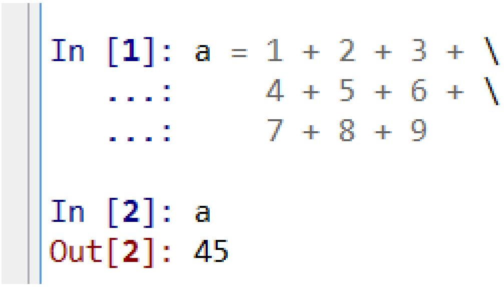
```
x = 'boy'; y = 'friend'
total = x + \
        y + \
        ' add as many lines as you want'
print(total)
```



```
In [9]: x = 'boy'; y = 'friend'
   ...: total = x + \
   ...:         y + \
   ...:         ' add as many lines as you want'
   ...: print(total)
   ...:
boyfriend add as many lines as you want
```

Line Continuation

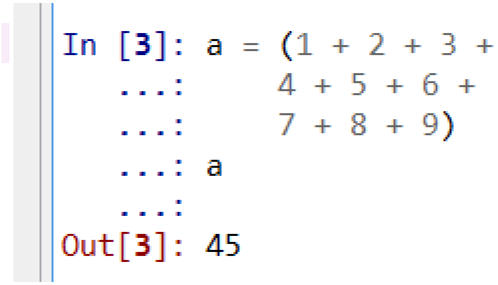
```
a = 1 + 2 + 3 + \
    4 + 5 + 6 + \
    7 + 8 + 9

a
```



```
In [1]: a = 1 + 2 + 3 + \
   ...:     4 + 5 + 6 + \
   ...:     7 + 8 + 9

In [2]: a
Out[2]: 45
```

Line Continuation

```
a = (1 + 2 + 3 +
     4 + 5 + 6 +
     7 + 8 + 9)
a
```

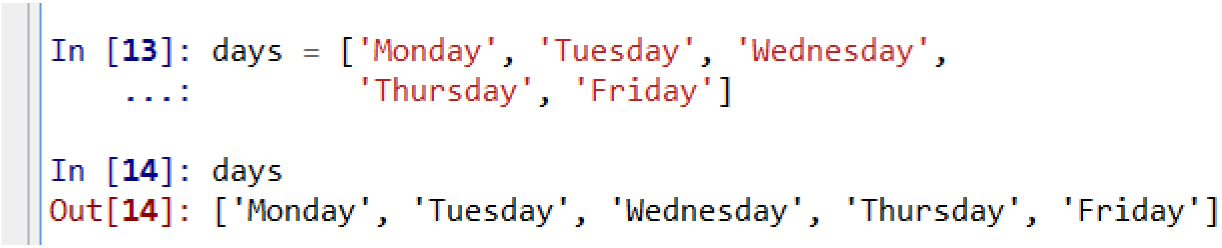


```
In [3]: a = (1 + 2 + 3 +
   ...:      4 + 5 + 6 +
   ...:      7 + 8 + 9)
   ...: a
   ...:
Out[3]: 45
```

Line Continuation

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```



```
In [13]: days = ['Monday', 'Tuesday', 'Wednesday',
    ...:          'Thursday', 'Friday']

In [14]: days
Out[14]: ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Formatting:

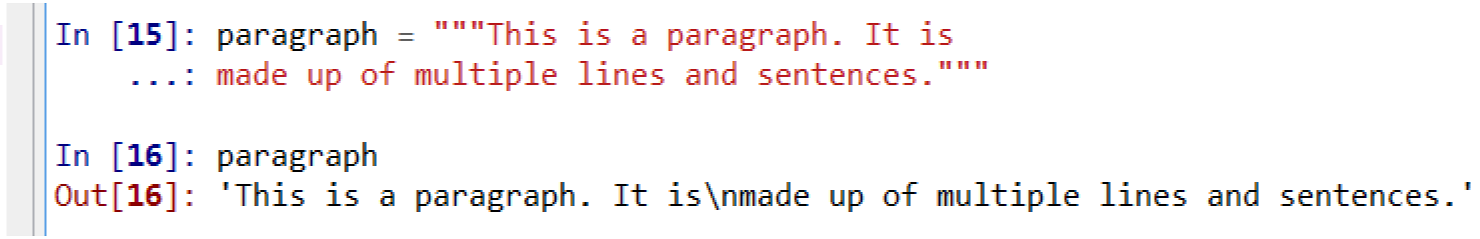
Quotation in Python

Quotation

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal

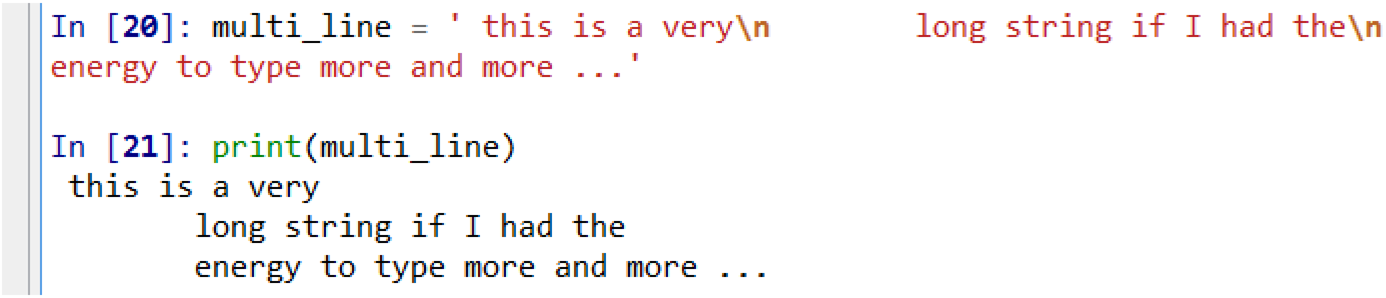
```
word = 'word'

sentence = "This is a sentence."

paragraph = """This is a paragraph. It is

made up of multiple lines and sentences."""
```



```
In [15]: paragraph = """This is a paragraph. It is
    ...: made up of multiple lines and sentences."""

In [16]: paragraph
Out[16]: 'This is a paragraph. It is\nmade up of multiple lines and sentences.'
```

Multiple Lines

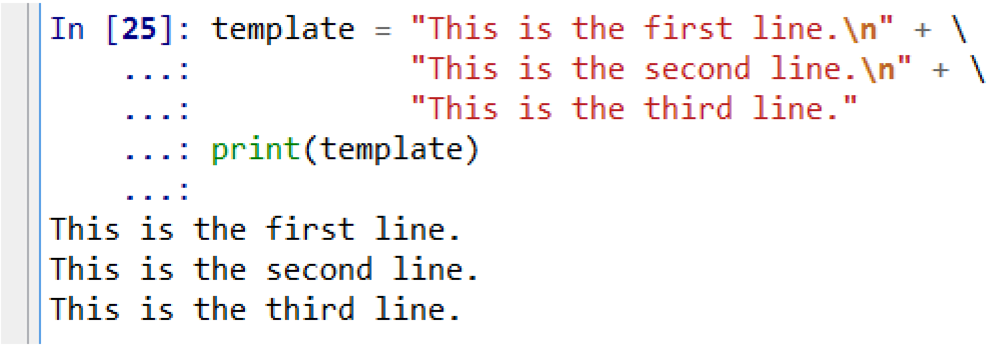
```
multi_line = ' this is a very\n        long string if I
had the\n        energy to type more and more ...'
```



```
In [20]: multi_line = ' this is a very\n        long string if I had the\n
energy to type more and more ...'

In [21]: print(multi_line)
 this is a very
        long string if I had the
        energy to type more and more ...
```

Concatenation

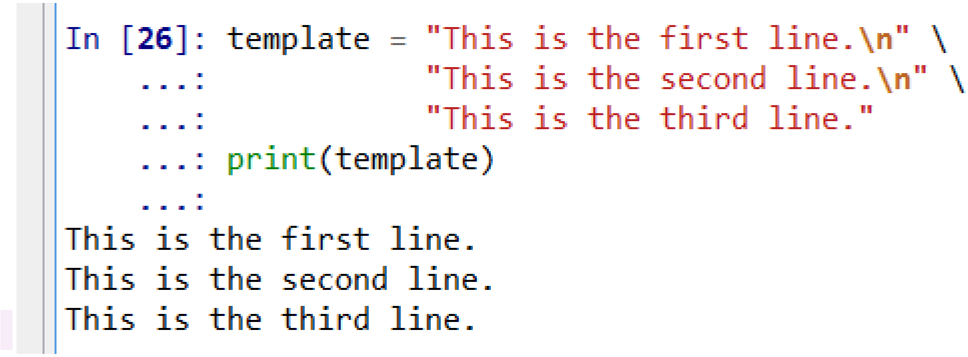
```
template = "This is the first line.\n" + \
           "This is the second line.\n" + \
           "This is the third line."
print(template)
```



```
In [25]: template = "This is the first line.\n" + \
    ...:            "This is the second line.\n" + \
    ...:            "This is the third line."
    ...: print(template)
    ...:
This is the first line.
This is the second line.
This is the third line.
```

Concatenation

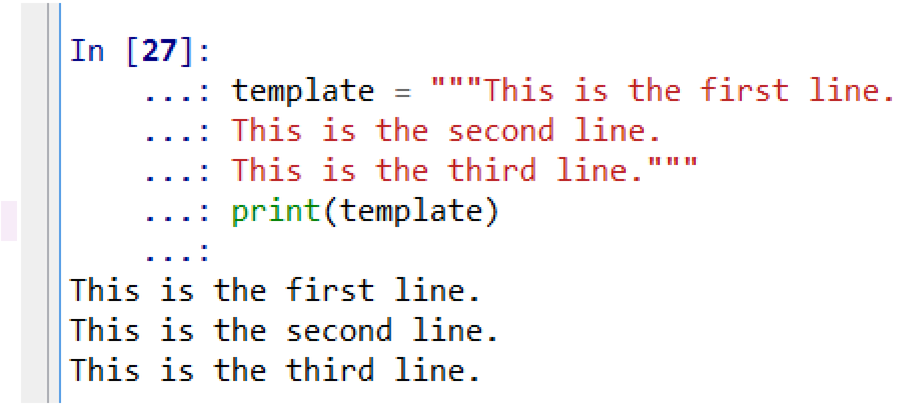
```
template = "This is the first line.\n" \
           "This is the second line.\n" \
           "This is the third line."
print(template)
```



```
In [26]: template = "This is the first line.\n" \
    ...:            "This is the second line.\n" \
    ...:            "This is the third line."
    ...: print(template)
    ...:
This is the first line.
This is the second line.
This is the third line.
```

Triple Quotes

```
template = """This is the first line.
This is the second line.
This is the third line."""
print(template)
```



```
In [27]:
    ...: template = """This is the first line.
    ...: This is the second line.
    ...: This is the third line."""
    ...: print(template)
    ...:
This is the first line.
This is the second line.
This is the third line.
```

Formatting:

Comments in Python

Hash Sign

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

```
#This is a long comment

#and it extends

#to multiple lines
```

Triple Quotes

Another way of doing this is to use triple quotes, either ''' or """.

These triple quotes are generally used for multi-line strings. But they can be used as multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.

```
"""This is also a

perfect example of

multi-line comments"""
```

Formatting:
Suites

Suites

A group of individual statements, which make a single code block are called suites in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite. For example:

```
if expression :
   suite
elif expression :
   suite
else :
   suite
```

Handling Errors:

Runtime errors

Errors

There are a lot of different situations that can raise errors in our code

Converting between datatypes
Opening files
Mathematical calculations
Trying to access a value in a list that does not exist

Runtime Errors

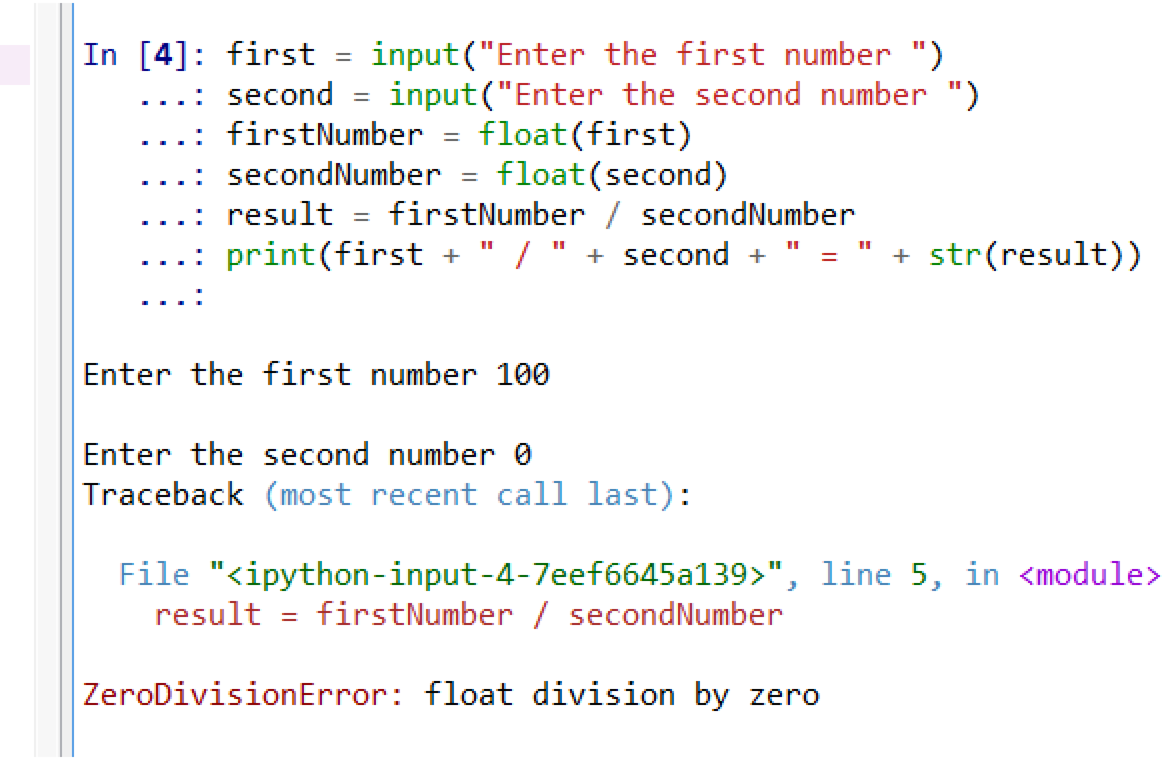
Runtime errors occur when the code basically works but something out of the ordinary ‘crashes’ the code. For instance,

- You write a calculator program and a user tries to divide a number by zero
- Your program tries to read a file, and the file is missing
- Your program is trying to perform a date calculation and the date provided is in the wrong format

No error handling

```
first = input("Enter the first number ")
second = input("Enter the second number ")
firstNumber = float(first)
secondNumber = float(second)
result = firstNumber / secondNumber
print(first + " / " + second + " = " + str(result))
```

No error handling



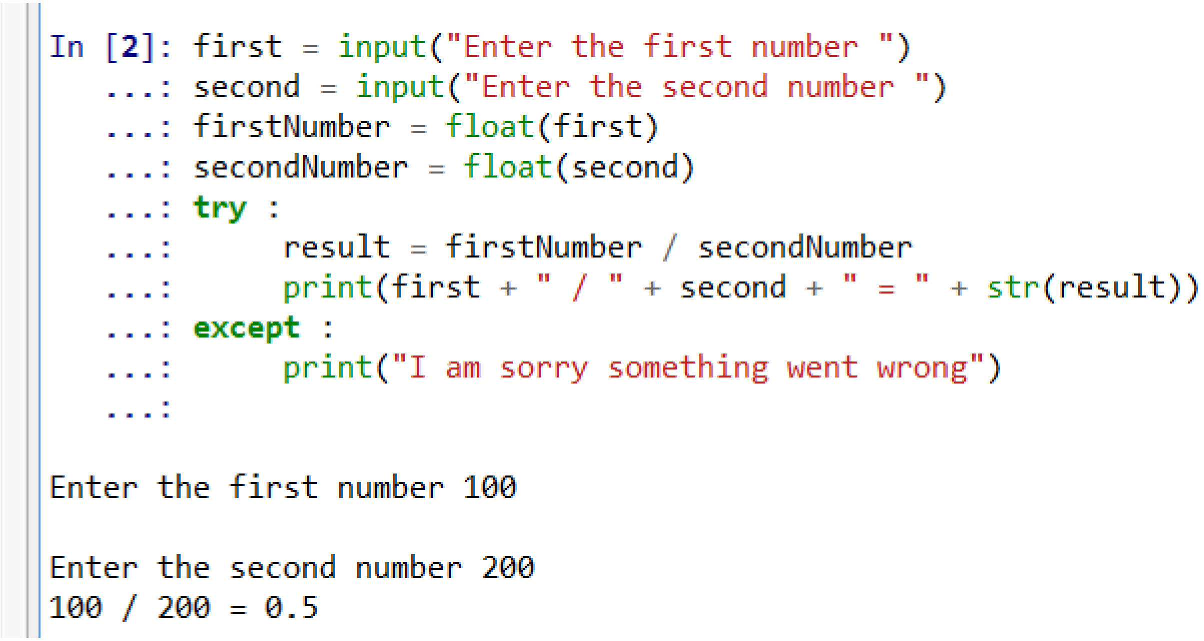
```
In [4]: first = input("Enter the first number ")
   ...: second = input("Enter the second number ")
   ...: firstNumber = float(first)
   ...: secondNumber = float(second)
   ...: result = firstNumber / secondNumber
   ...: print(first + " / " + second + " = " + str(result))
   ...:

Enter the first number 100

Enter the second number 0
Traceback (most recent call last):

  File "<ipython-input-4-7eef6645a139>", line 5, in <module>
    result = firstNumber / secondNumber

ZeroDivisionError: float division by zero
```

add error handling

You can add a try/except around the code that generates the error to handle it gracefully. The code in the except only runs if there is an error generated when executing the code in the try:

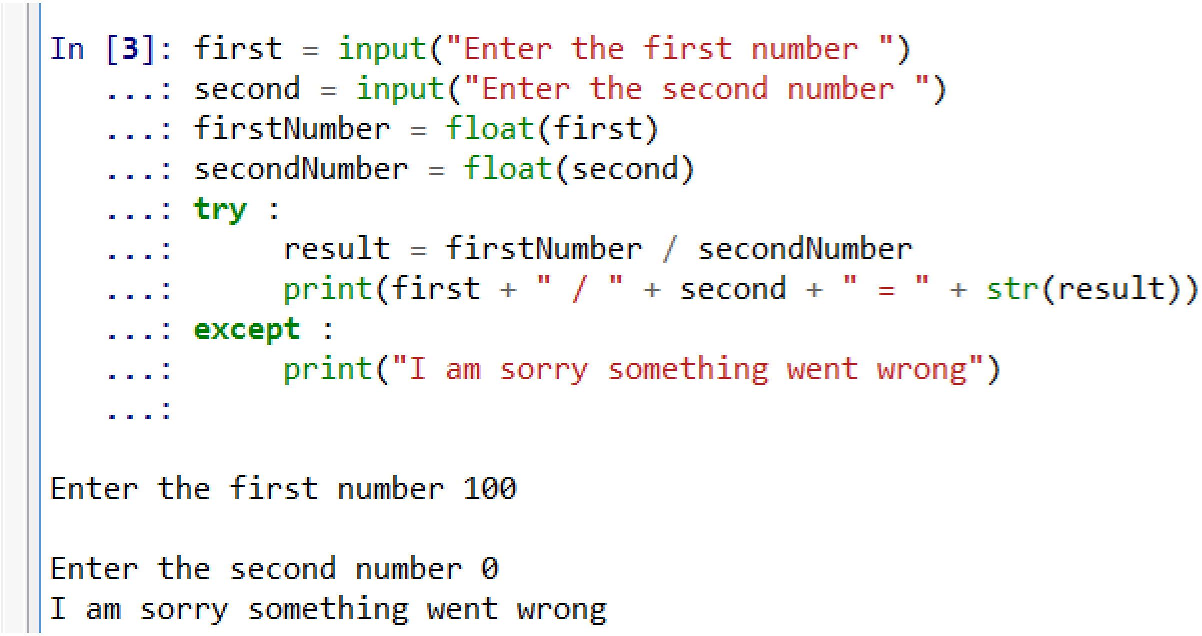
```
first = input("Enter the first number ")
second = input("Enter the second number ")
firstNumber = float(first)
secondNumber = float(second)
try :
     result = firstNumber / secondNumber
     print(first + " / " + second + " = " + str(result))
except :
     print("I am sorry something went wrong")
```

add error handling



```
In [2]: first = input("Enter the first number ")
   ...: second = input("Enter the second number ")
   ...: firstNumber = float(first)
   ...: secondNumber = float(second)
   ...: try :
   ...:     result = firstNumber / secondNumber
   ...:     print(first + " / " + second + " = " + str(result))
   ...: except :
   ...:     print("I am sorry something went wrong")
   ...:

Enter the first number 100

Enter the second number 200
100 / 200 = 0.5
```

add error handling



```
In [3]: first = input("Enter the first number ")
   ...: second = input("Enter the second number ")
   ...: firstNumber = float(first)
   ...: secondNumber = float(second)
   ...: try :
   ...:     result = firstNumber / secondNumber
   ...:     print(first + " / " + second + " = " + str(result))
   ...: except :
   ...:     print("I am sorry something went wrong")
   ...:

Enter the first number 100

Enter the second number 0
I am sorry something went wrong
```

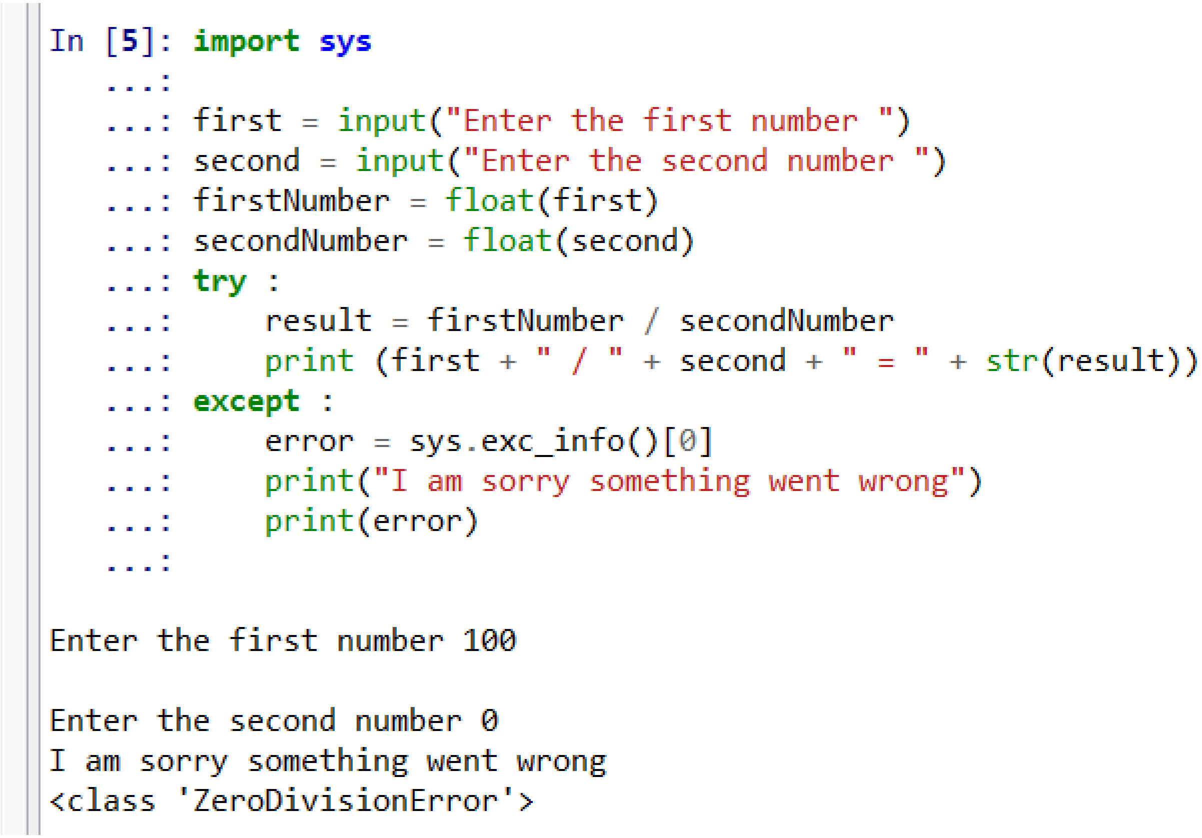
Handling Errors:

Handling Errors

want to know what the error was

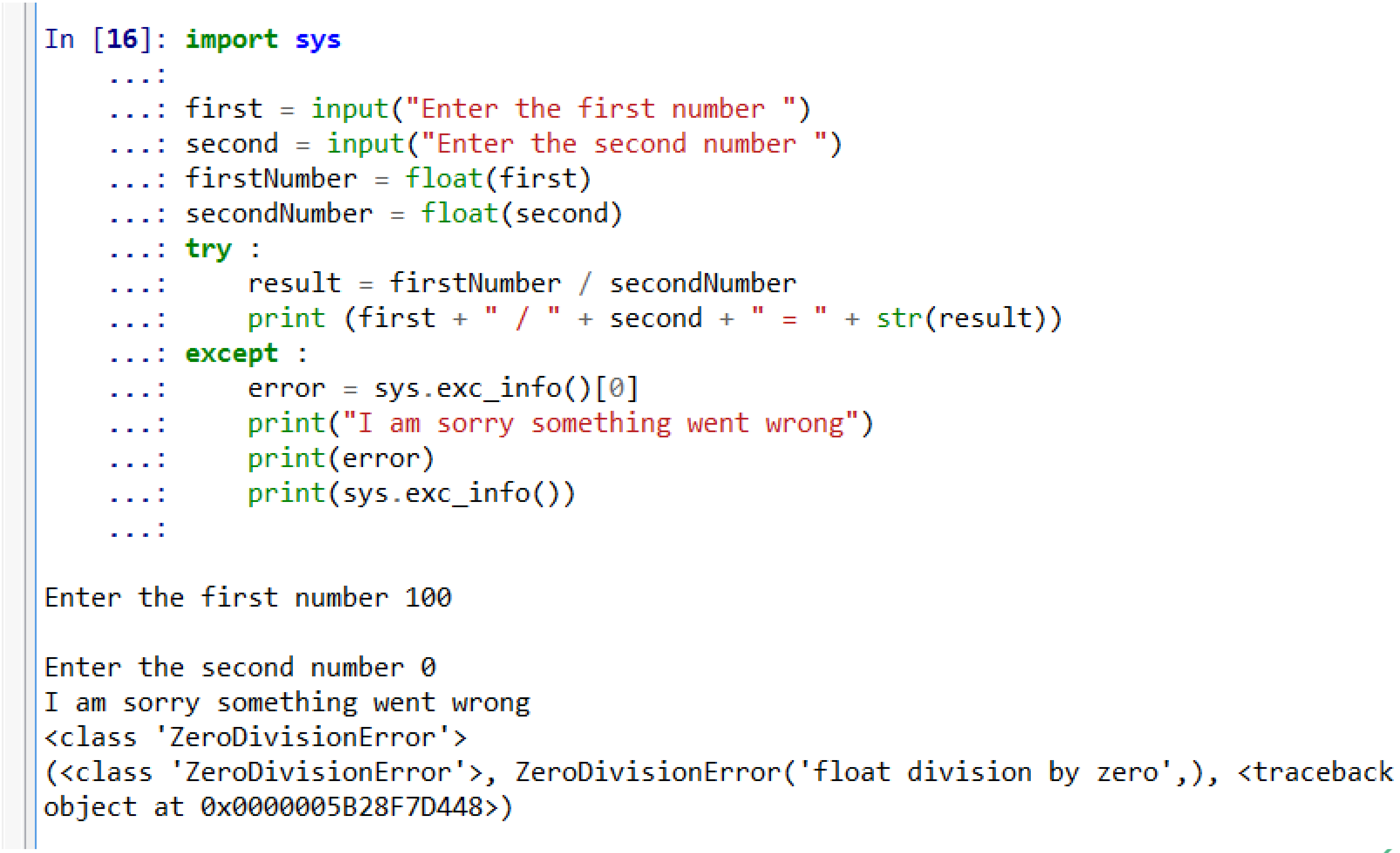
```
import sys

first = input("Enter the first number ")
second = input("Enter the second number ")
firstNumber = float(first)
secondNumber = float(second)
try :
    result = firstNumber / secondNumber
    print (first + " / " + second + " = " + str(result))
except :
    error = sys.exc_info()[0]
    print("I am sorry something went wrong")
    print(error)
```

want to know what the error was



```
In [5]: import sys
   ...:
   ...: first = input("Enter the first number ")
   ...: second = input("Enter the second number ")
   ...: firstNumber = float(first)
   ...: secondNumber = float(second)
   ...: try :
   ...:     result = firstNumber / secondNumber
   ...:     print (first + " / " + second + " = " + str(result))
   ...: except :
   ...:     error = sys.exc_info()[0]
   ...:     print("I am sorry something went wrong")
   ...:     print(error)
   ...:

Enter the first number 100

Enter the second number 0
I am sorry something went wrong
<class 'ZeroDivisionError'>
```

want to know what the error was



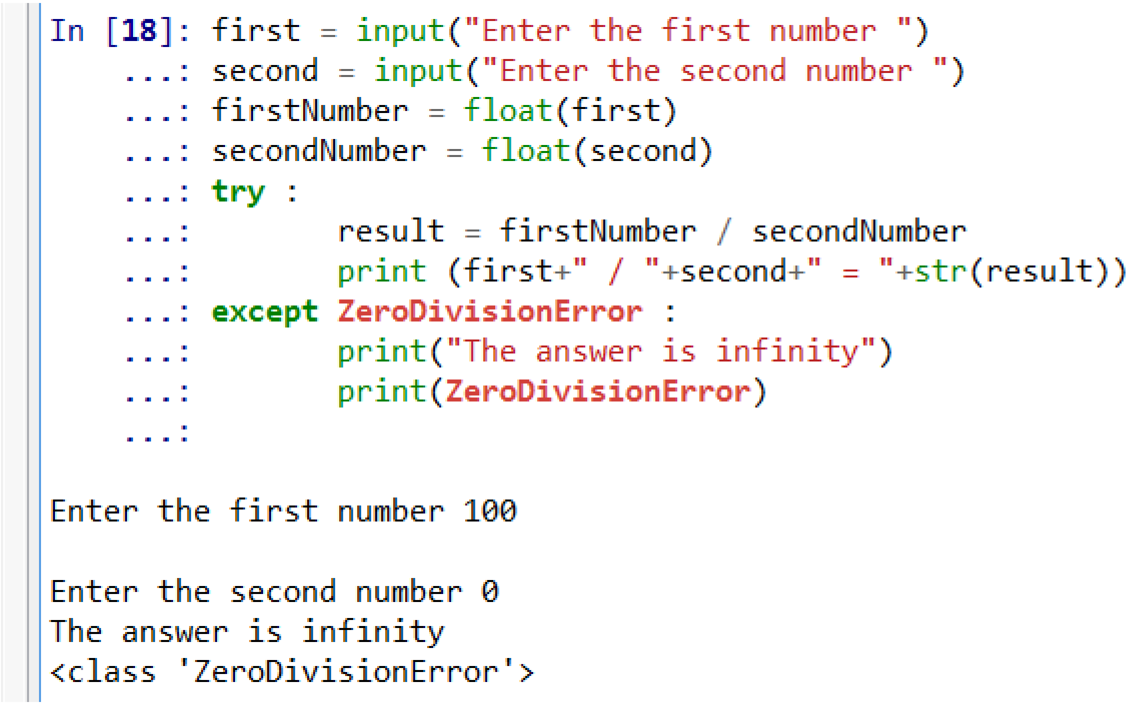
```
In [16]: import sys
    ...:
    ...: first = input("Enter the first number ")
    ...: second = input("Enter the second number ")
    ...: firstNumber = float(first)
    ...: secondNumber = float(second)
    ...: try :
    ...:     result = firstNumber / secondNumber
    ...:     print (first + " / " + second + " = " + str(result))
    ...: except :
    ...:     error = sys.exc_info()[0]
    ...:     print("I am sorry something went wrong")
    ...:     print(error)
    ...:     print(sys.exc_info())
    ...:

Enter the first number 100

Enter the second number 0
I am sorry something went wrong
<class 'ZeroDivisionError'>
(<class 'ZeroDivisionError'>, ZeroDivisionError('float division by zero',), <traceback
object at 0x0000005B28F7D448>)
```

Handle That Exact Error

If you know exactly what error is occurring, you can specify how to handle that exact error

```
first = input("Enter the first number ")
second = input("Enter the second number ")
firstNumber = float(first)
secondNumber = float(second)
try :
      result = firstNumber / secondNumber
      print (first+" / "+second+" = "+str(result))
except ZeroDivisionError :
      print("The answer is infinity")
      print(ZeroDivisionError)
```

Handle That Exact Error



```
In [18]: first = input("Enter the first number ")
    ...: second = input("Enter the second number ")
    ...: firstNumber = float(first)
    ...: secondNumber = float(second)
    ...: try :
    ...:        result = firstNumber / secondNumber
    ...:        print (first+" / "+second+" = "+str(result))
    ...: except ZeroDivisionError :
    ...:        print("The answer is infinity")
    ...:        print(ZeroDivisionError)
    ...:

Enter the first number 100

Enter the second number 0
The answer is infinity
<class 'ZeroDivisionError'>
```

One or More Specific Errors

Ideally you should handle one or more specific errors and then have a generic error handler as well

```
first = input("Enter the first number ")
second = input("Enter the second number ")
firstNumber = float(first)
secondNumber = float(second)
try :
      result = firstNumber / secondNumber
      print (first + " / " +second+ " = "+str(result))
except ZeroDivisionError :
      print("The answer is infinity")
except :
      error = sys.exc_info()[0]
      print("I am sorry something went wrong")
      print(error)
```

Code After Except

Any code you place after the try except will always execute

```
first = input("Enter the first number ")
second = input("Enter the second number ")
firstNumber = float(first)
secondNumber = float(second)
try :
      result = firstNumber / secondNumber
      print (first + " / " +second+" = "+str(result))
except ZeroDivisionError :
      print("The answer is infinity")
except :
      error = sys.exc_info()[0]
      print("I am sorry something went wrong")
      print(error)
print("This message always displays!")
```

Force the Program to Exit

How can I force my program to exit if an error occurs and I don’t want to continue? You can use the function sys.exit() in the sys library

```
try :
      result = firstNumber / secondNumber
      print (first+" / " +second + " = " + str(result))
except ZeroDivisionError :
      print("The answer is infinity")
      sys.exit()
print("This message only displays if there is no
error!")
```

Error Flag

You can also use variables and an if statement to control what happens after an error

```
try :
      result = firstNumber / secondNumber
      print (first+" / " + second + " = " + str(result))
      errorFlag = False # default value
except ZeroDivisionError :
      print("The answer is infinity")
      errorFlag = True
if not errorFlag :
      print("This message only displays if there is no
error!")
```

Handling Errors:
a list of standard errors

a list of standard Python errors

You can test it yourself and when an error occurs use the sys.exc_info() function to get the name of the error. There is a list of standard Python errors

https://docs.python.org/3/c-api/exceptions.html#standard-exceptions

Standard Errors

Exception	Cause of Error
AssertionError	Raised when assert statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the input() functions hits end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits interrupt key (Ctrl+c or delete).
MemoryError	Raised when an operation runs out of memory.

Standard Errors

Exception	Cause of Error
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when system operation causes system related error.
OverflowError	Raised when result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by next() function to indicate that there is no further item to be returned by iterator.

Standard Errors

Exception	Cause of Error
SyntaxError	Raised by parser when syntax error is encountered.
IndentationError	Raised when there is incorrect indentation.
TabError	Raised when indentation consists of inconsistent tabs and spaces.
SystemError	Raised when interpreter detects internal error.
SystemExit	Raised by sys.exit() function.
TypeError	Raised when a function or operation is applied to an object of incorrect type.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.

Standard Errors

Exception	Cause of Error
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translating.
ValueError	Raised when a function gets argument of correct type but improper value.
ZeroDivisionError	Raised when second operand of division or modulo operation is zero.

Handling Errors:
Custom Exceptions

Self-Defined Exceptions

In Python, users can define such exceptions by creating a new class. This exception `class` has to be derived, either directly or indirectly, from `Exception` `class`. Most of the built-in exceptions are also derived form this `class`.

Class: Self-Defined Exceptions

```
>>> class CustomError(Exception):
...     pass
...
>>> raise CustomError
Traceback (most recent call last):
...
__main__.CustomError
>>> raise CustomError("An error occurred")
Traceback (most recent call last):
...
__main__.CustomError: An error occurred
```

Custom Errors

Here, we have created a user-defined exception called `CustomError` which is derived from the Exception class. This new exception can be raised, like other exceptions, using the raise statement with an optional error message.

When we are developing a large Python program, it is a good practice to place all the user-defined exceptions that our program raises in a separate file. Many standard modules do this. They define their exceptions separately as exceptions.py or errors.py (generally but not always).

User-defined exception class can implement everything a normal class can do, but we generally make them simple and concise. Most implementations declare a custom base class and derive others exception classes from this base class. This concept is made clearer in the following example.

User-Defined Exception in Python

In this example, we will illustrate how user-defined exceptions can be used in a program to raise and catch errors.

This program will ask the user to enter a number until they guess a stored number correctly. To help them figure it out, hint is provided whether their guess is greater than or less than the stored number.

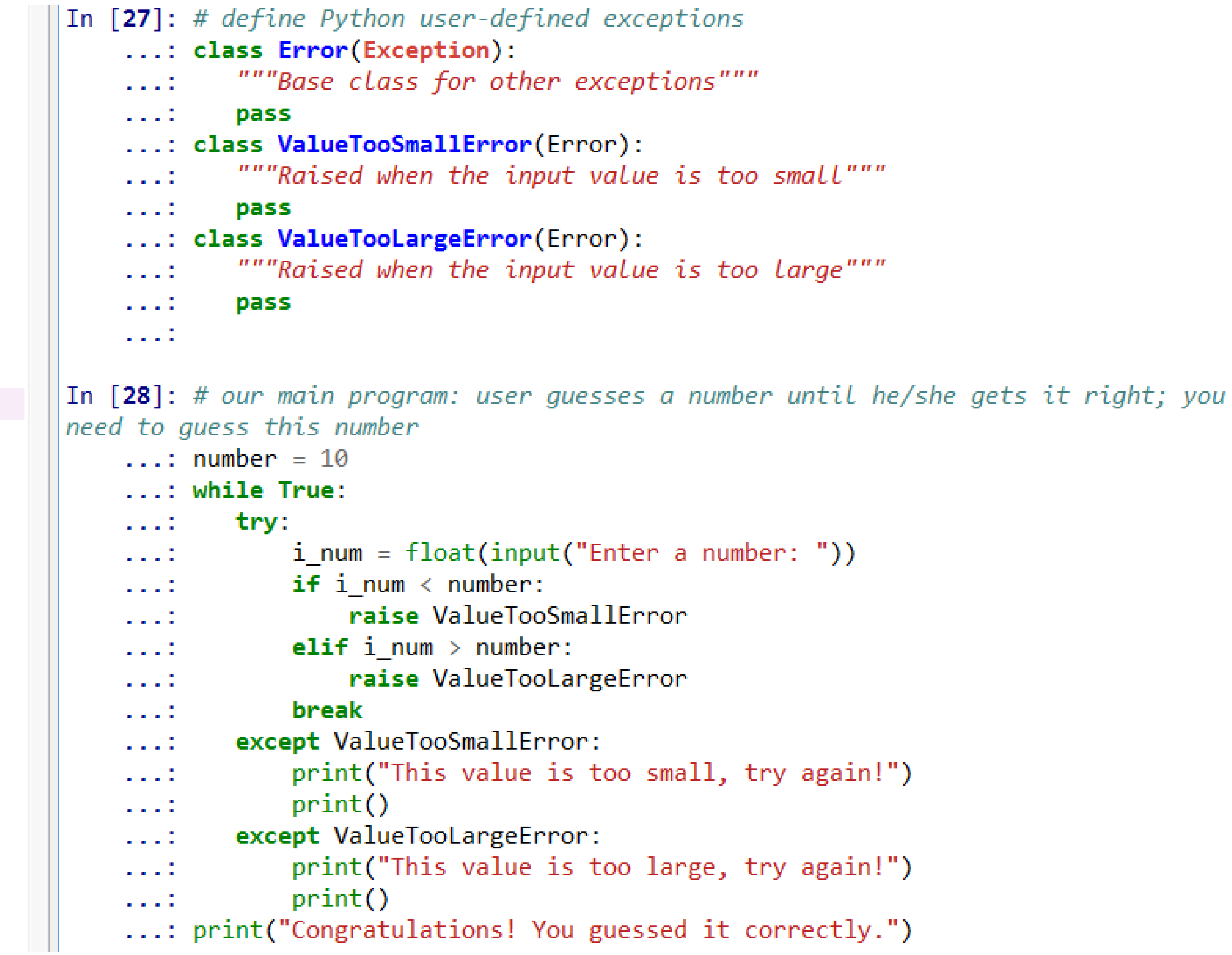
Example: User-Defined Exception

```
# define Python user-defined exceptions
class Error(Exception):
   """Base class for other exceptions"""
   pass
class ValueTooSmallError(Error):
   """Raised when the input value is too small"""
   pass
class ValueTooLargeError(Error):
   """Raised when the input value is too large"""
   pass
```

Example: User-Defined Exception (cont’d)

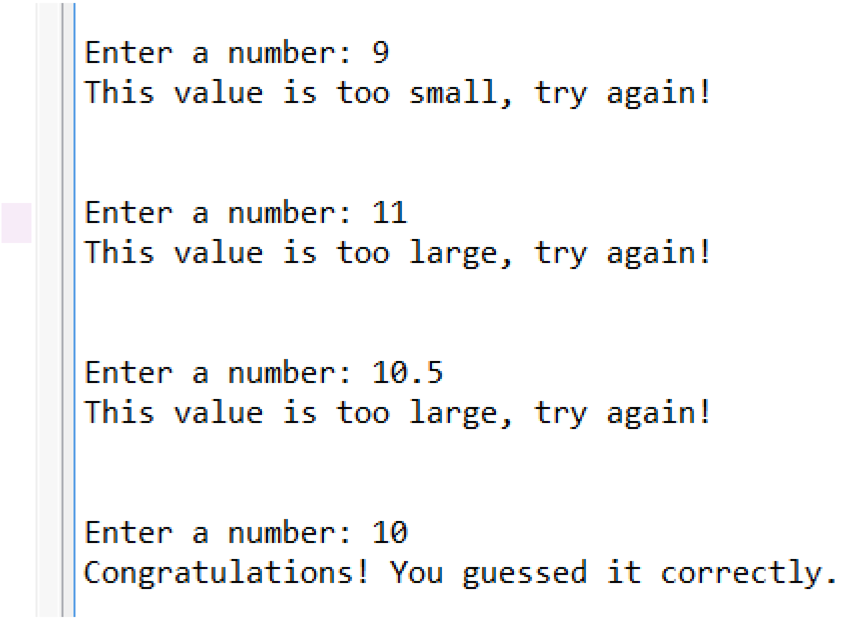
```
# our main program: user guesses a number until he/she gets it right; you need to guess this number
number = 10
while True:
   try:
       i_num = float(input("Enter a number: "))
       if i_num < number:
           raise ValueTooSmallError
       elif i_num > number:
           raise ValueTooLargeError
       break
   except ValueTooSmallError:
       print("This value is too small, try again!")
       print()
   except ValueTooLargeError:
       print("This value is too large, try again!")
       print()
print("Congratulations! You guessed it correctly.")
```

Example: User-Defined Exception (cont’d)



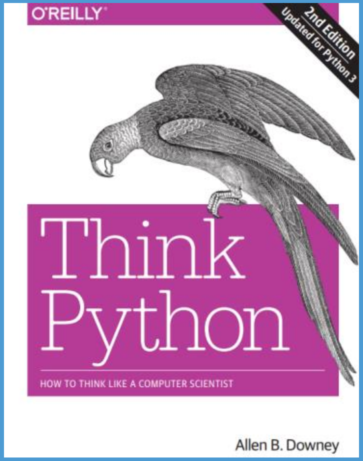
```
In [27]: # define Python user-defined exceptions
    ...: class Error(Exception):
    ...:     """Base class for other exceptions"""
    ...:     pass
    ...: class ValueTooSmallError(Error):
    ...:     """Raised when the input value is too small"""
    ...:     pass
    ...: class ValueTooLargeError(Error):
    ...:     """Raised when the input value is too large"""
    ...:     pass
    ...:

In [28]: # our main program: user guesses a number until he/she gets it right; you
need to guess this number
    ...: number = 10
    ...: while True:
    ...:     try:
    ...:         i_num = float(input("Enter a number: "))
    ...:         if i_num < number:
    ...:             raise ValueTooSmallError
    ...:         elif i_num > number:
    ...:             raise ValueTooLargeError
    ...:         break
    ...:     except ValueTooSmallError:
    ...:         print("This value is too small, try again!")
    ...:         print()
    ...:     except ValueTooLargeError:
    ...:         print("This value is too large, try again!")
    ...:         print()
    ...: print("Congratulations! You guessed it correctly.")
```

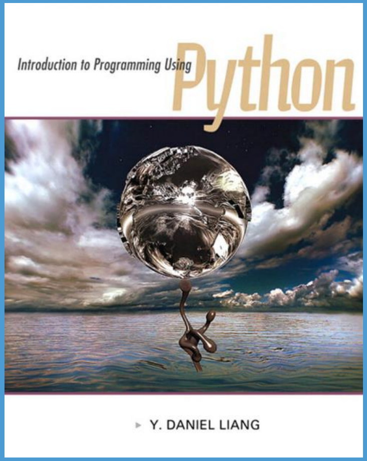
Example: User-Defined Exception (cont’d)

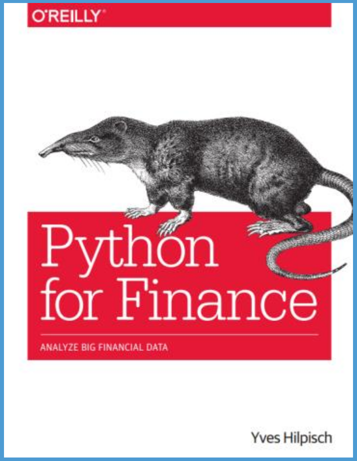


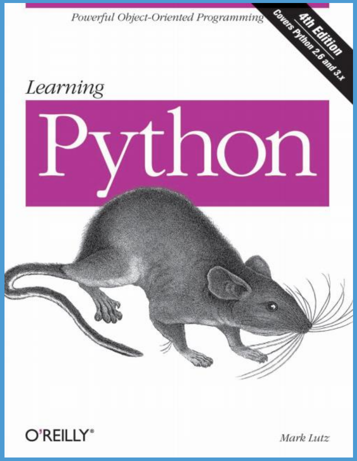
```
Enter a number: 9
This value is too small, try again!


Enter a number: 11
This value is too large, try again!


Enter a number: 10.5
This value is too large, try again!


Enter a number: 10
Congratulations! You guessed it correctly.
```

References


O'REILLY
2nd Edition
Updated for Python 3
Think Python
HOW TO THINK LIKE A COMPUTER SCIENTIST
Allen B. Downey


Introduction to Programming Using Python
Y. DANIEL LIANG


O'REILLY
Python for Finance
ANALYZE BIG FINANCIAL DATA
Yves Hilpisch


Powerful Object-Oriented Programming
4th Edition
Covers Python 2.6 and 3.x
Learning Python
O'REILLY
Mark Lutz