ST2195 Programming for Data Science
Block 5 Variables, Mutability and Aliasing in Python and R

VLE Resources

There are extra resources for this Block on the VLE – including quizzes and videos please make sure you review them as part of your learning. You can find them here: https://emfss.elearning.london.ac.uk/course/view.php?id=382#section-5

** Note: The code chunks below should be run in the following order **

Outline

In this page we focus on presenting some programming concepts in Python. Before doing this, we first return to the definitions of data types we have learned previously in the notes on Python data structures, and describe their association with objects. We then talk about mutability and aliasing in Python. Finally, we compare how objects in Python and R behave differently.

Object and Type

You can consider everything in Python as an *object*. Each object has a *type* and the type dictates the behaviour of the object. We have seen in the page "Python and Data Structures" about different data type and structure, and here we revisit what we have learned.

The table below lists some of the built-in types (i.e. the types you can use straight away) in Python:

Type	Example
`int`	`1`, `123`
`float`	`3.14`, `1.0`
`bool`	`True`, `False`
`None`	`None`
`string`	`"hello"`, `"1"`

Note that `1` is `int` but `1.0` is `float` and `"1"` is `string`.

Type Checking and Casting

We can check the type of an object by the function `type()`. For example

```
type(1)
<class 'int'>
type(1.0)
<class 'float'>
type("1")
<class 'str'>
```

We can change the type of an object to another type by casting. For example:

```
type("1")
<class 'str'>
type(int("1"))
<class 'int'>
```

The first object `"1"` has the type `string` whereas the second object `int("1")` has the type `int` as the function `int()` converts the argument value to an integer (i.e. a object with type `int`).

What do you expect the outcome of the following code?

```
int("a")
```

Relations Between Type and the Behaviour of the Objects

What we can do on objects depends on their type. For example, when we use the `+` operator on two objects with numeric type (`int` here in the example below):

```
1 + 2
3
```

we get another object with `int` type and value `3`, which is the sum of `1` and `2`.

If we do:

```
1 + "a"
```

we will get the error message: `TypeError: unsupported operand type(s) for +: 'int' and 'str'`, as we are not allow to “add” `int` and `string` together. The type of the objects decides what you can do and what you cannot do on them.

The type of an object also decides the behaviour of the operation. If we apply the summation operator on two `string`s:

```
"hello"+"world"
'helloworld'
```

we are not “summing” the `string`s in the numerical sense but we *concatenate* the `string`s to get a new `string`. The type therefore dictates the behaviour of the objects.

What if we run the same code in R?

```
## R code
"hello"+"world"
```

Does it work?

Variables and Assignment

In the above examples we created an object with type `int` and value `3` by the expression `1+2`. We also created another object with type `string` `helloworld` by concatenating two `string`s together. However, these objects are created and lost. We are not able to use them again (e.g. for further calculation). In order to be able to use them later, we need to assign them to *variable*s. We can do it by using the assignment operator `=`. For example:

```
pi = 3.14
```

which we assign the object with type `float` and value `3.14` to the variable with name `pi`. So now we can retrieve the value by using the variable `pi`:

```
pi
3.14
```

The variable will have the same type as the object that assigned to it:

```
print(type(pi))
<class 'float'>
print(type(3.14))
<class 'float'>
```

Similar to R, we can assign value to a variable using an expression, see the third line of the code:

```
pi = 3.14
radius = 11
area = pi * (radius**2)
area
379.94
```

We can rebind the variable to another object by simply assigning another object to it. For example:

```
pi = "hello"
pi
'hello'
```

Now `pi` is bound to the object with value `"hello"`.

The type of a variable depends on the object assigned to it. Now the type of `pi` is `string`, as it binds to the object `"hello"`.

```
type(pi)
```

```
<class 'str'>
```

Note that this assignment has no effect on the value of the variable `area`, which remains as `3*(11**2) = 379.94`

```
area
```

```
379.94
```

Naming a Variable

Similar to R, variable names can contain upper-case and lower-case letters, digits (but they cannot start with a digit) and underline characters, and variable names are case-sensitive. Unlike R, and you **cannot** use dots as part of the variable name. We will see later that dot is used to get attributes from an object.

In Python, some of the words are reserved for specific purposes (known as *keywords*) and you are not allowed to use them as variable names. Example of the keywords are `if`, `else`, `for`, `def`, etc. If you run the following:

```
if = 3
```

you will get an error with the error message: `SyntaxError: invalid syntax`.

Mutability

In the page “Python and Data Structures” we have seen that we can modify objects like `list`. For objects that can be modified, we say they are *mutable*. Examples of type of mutable objects are `list` and `dict`.

For example, we can modify a `list`:

```
colours = ["red", "blue", "green"]
colours[0] = "orange"
colours
```

```
['orange', 'blue', 'green']
```

Remember in Python the indices start from `0` (in R indices start from `1`).

On the other hand, we cannot modify *immutable* objects like `tuple` or `string`. For example if we try to modify an element in a `tuple`:

```
colours = ("red", "blue", "green")
colours[0] = "orange"
```

It will give the error message `TypeError: 'tuple' object does not support item assignment`.

Aliasing

Consider the following code:

```
colours = ["red", "blue", "green"]
colours_2 = colours
colours_2[0] = "orange"
print(colours_2)
['orange', 'blue', 'green']
print(colours)
['orange', 'blue', 'green']
```

While you probably expected `colours_2` has `"orange"` as the first element, you probably did not expect that`colours` has `"orange"` as the first element too! What is wrong? After all if we write similar code in R:

```
# R code
r_colours <- c("red", "blue", "green")
r_colours_2 <- r_colours
r_colours_2[1] <- "orange"
print(r_colours_2)
print(r_colours)
[1] "orange" "blue"   "green"
[1] "red"    "blue"   "green"
```

`r_colours` would not be changed! Why similar blocks of code behave differently in R and Python?

To understand further, it is worth taking a closer look what assignment of an object to a variable is actually doing behind the scene in Python. Below we use Python Tutor to visualise how the first 3 lines of the Python code bind the object (the `list`) with variable names `colours` and `colours_2`, and change the object through `colours_2`. Please press the `next` button to see what actually is happening for each line of the code.

```
colours = ["red", "blue", "green"]
colours_2 = colours
colours_2[0] = "orange"
```

As we can see in the visualisation above, when we run `colours_2 = colours`, what we are copying is the *reference* to the `list` but not the `list` itself. In other words, what we are doing is assigning the `list` that the variable `colours` referred to to `colours_2`. Effectively now the variables `colours` and `colours_2` are binding to the same `list`. You can think `colours` and `colours_2` are just two names of the same object. The `list` itself is not copied. When we run the third line, we modify the `list` through the variable name `colours_2`. As the variable `colours` refers to the same `list` as `colours_2`, when we retrieve the `list` by the variable name `colours`, we will see the first element is `"orange"` instead of `"red"`.

We can verify that that the variables `colours` and `colours_2` refer to the same object by using the function `id()`:

```
id(colours)
5441935496
id(colours_2)
5441935496
```

The function `id()` returns a unique identify of an object and we can see that `id(colours)` and `id(colours_2)` return the same value.

If we want to actually make a copy of the `list` (the object), we need to do it explicitly by using the `copy()` function:

```
colours_3 = colours.copy()
colours_3[0] = "black"
print(colours_3)
['black', 'blue', 'green']
print(colours)
['orange', 'blue', 'green']
```

When working on Python, remember that a container can be modified by any variables binding to them. If you want to make sure no other variables can modify the container, explicitly create a copy when assigning to another variable, as shown in the code block above.

Should We Worry About Aliasing with Immutable Objects?

The short answer is no because *immutable* objects cannot be changed. Consider the following code:

```
pi = 3.14
print(pi)
3.14
pi = "hello"
print(pi)
hello
```

Are we not changing the object in the third line? While we indeed "update" the value associate with the variable `pi` by the second assignment, object created from the first assignment is not changed. What is actually happening in the second assignment is that a *new* object associated with the value `"hello"` is created and is bound to the variable `pi`. We can confirm this by printing the identity of the object bound to `pi`:

```
pi = 3.14
print(id(pi))
5441320784
pi = "hello"
```

```
print(id(pi))
5441941320
```

We can see that the identity has changed, showing that `pi` on line 1 and 3 are binding to different objects. We did not *change* the immutable object, instead we created a new immutable object.

Therefore, we do not worry about the aliasing issue, as shown in the example below:

```
pi = 3.14
a = pi
print(id(pi))
5441320904
print(id(a))
5441320904
pi = "hello"
print(a)
3.14
print(id(pi))
5441416304
print(id(a))
5441320904
```

Why Does the R Code Behave Differently?

The *short* answer is that R uses the *copy-on-modify* strategy. When you want to modify the object that a variable is pointing to, R will make a copy of the object, make the change and assign it to the variable. If there are other variables pointing to the same object, they will remain pointing to the old, unmodified object. In reality it is more complicated than this and we will explain further below.

If you find it difficult to understand even after reading the explanation above, just remember the following: in this course you do not need to worry about the aliasing issue in R. When you modify a variable in R, in this course you can *assume* other variables will not be modified.

Is the R `vector` a Mutable Object?

We can modify `vector` in R in a similar way as `list` in Python:

```
# R code
r_colours <- c("red", "blue", "green")
r_colours[1] <- "orange"
r_colours
[1] "orange" "blue"   "green"
```

It feels like R `vector` is mutable as we can change its element.

Does R Copy the Reference Like Python?

In R, when we assign `r_colours_2 <- r_colours`, similar to Python only the reference is copied. We can verify it using the function `tracemem()`. `tracemem()` shows you the memory that the variable is pointing to, and will give a message when the corresponding object is copied.

```
# R code

r_colours <- c("red", "blue", "green")

tracemem(r_colours)

r_colours_2 <- r_colours

tracemem(r_colours_2)

[1] "<0x7fbb2ee79008>"

[1] "<0x7fbb2ee79008>"
```

Note that no message is shown as no object is copied. Both `r_colours` and `r_colours_2` have the same memory address.

However, when we change `r_colours`,

```
# R code

r_colours[1] <- "orange"

tracemem[0x7fbb2ee79008 -> 0x7fbb2f304308]: eval eval withVisible withCallingHandle
rs handle timing_fn evaluate_call <Anonymous> evaluate in_dir block_exec call_block
process_group.block process_group withCallingHandlers process_file <Anonymous> <Ano
nymous>
```

We can see the message from the `tracemem()` function printing out `tracemem[old_address -> new_address]`, which tells us that `r_colours` is now pointing to a *new* object. When there are more than one variables pointing to an object and we want to modify one of the variables (e.g. `r_colours[1] <- "orange"`), R will make a copy of the object, make a change and then assign to the variable which trigger the change. For other objects, they are still pointing to the old object.

We can verify that `r_colours` is printing to a new object and `r_colours_2` is pointing to the old object by the `tracemem()` function:

```
# R code

tracemem(r_colours)

tracemem(r_colours_2)

[1] "<0x7fbb2f304308>"

[1] "<0x7fbb2ee79008>"
```

Note that the address for `r_colours` has changed, but the address for `r_colours_2` has not changed. It is still pointing the old object.

Therefore, the first element of `r_colours` has changed to `"orange"` but not `r_colours_2`:

```
# R code
print(r_colours)
print(r_colours_2)
[1] "orange" "blue"   "green"
[1] "red"    "blue"   "green"
```

Useful Links and Resources

- Guttag, J.V. (2013). Introduction to computation and programming using Python. Section 2.1.2.
- Python tutor: A tool to visualise your Python code
- Official Python tutorial on `list`

Control Flow Structures in Python

** Note: The code chunks below should be run in the following order **

Conditional Statements

We start with the `if` statement. Its basic form follows:

```
if condition:
  true_action_1
  true_action_2
  ...

if condition:
  true_action_1
  true_action_2
  ...
else:
  false_action_1
  false_action_2
  ...
```

For example:

```
mark = 80
if mark >= 50:
    print("pass")
pass
```

While the control statements between R and Python are quite similar, there are several differences between them:

1. No parentheses are needed to enclose the condition
2. `:` is needed after the `if` and `else` keywords
3. **Indentation is required for the block after the condition statement**. You have to indent following the way demonstrated above. While any number of spaces will do as long as the spacing is consistent, the standard is to use four spaces for one level indentation. The following code will return error with error message `IndentationError: expected an indented block`:

```
mark = 80
if mark >= 50:
print("pass")
```

More on Indentation

In Python, each line of code in the the block must be indented by the same amount. This is in contrast to many other languages (like R) where indentation is optional, and curly brackets `{}` are used to define blocks.

In Python, different indentation may provide different outcome, and some indentation may result in errors. For example:

```
mark = 30
if mark >= 50:
    print("pass")
    print("congrats!")
```

and

```
mark = 30
if mark >= 50:
    print("pass")
print("congrats!")
congrats!
```

provide different results. Why is this the case? Note that the line `print("congrats!")` is not part of the control flow in the second example. Therefore, it will be executed no matter whether the condition `mark >= 50` is satisfied.

The code below will trigger the error message `IndentationError: unexpected indent`, as the `if` statement should not be indented.

```
mark = 80
    if mark >= 50:
print("pass")
```

Similarly, the code below results in the same error message, as the indentation should be the same for both print statements.

```
if mark >= 50:
    print("pass")
      print("congrats")
```

The code below results in the same error message as well, as the `if` and the `else` statement should be at the same level.

```
if mark >= 50:
    print("pass")
    else:
        print("fail")
```

Again, the code below results in the same error message. Why is it the case?

```
if mark >= 50:
    print("pass")
print("congrats!")
else:
    print("fail")
```

Indentation in R vs Python

In R the use of indentation is optional. You have the freedom to choose whether to indent and how to indent your code. Often indentation is used in R to make the code easier to be read. If we rewrite the examples which result in error above in R syntax they will run fine in R:

```
## R code
mark = 80
if (mark >= 50)
print("pass")
[1] "pass"
## R code
    if (mark >= 50)
print("pass")
[1] "pass"
## R code
if (mark >= 50) {
  print("pass")
    print("congrats")
}
[1] "pass"
[1] "congrats"
```

While the above R code is valid, it is not recommended to write the code with non-standard indentation as it makes it more difficult to understand the code and likely to cause confusion.

Nested `if`

As in R, we can nest an `if` statement within an `if` statement. In Python we have the `elif` keyword which is short for 'else if'. The following two codes will give the same result:

```
mark = 65
if mark >= 70:
    print("distinction")
elif mark >= 60:
    print("merit")
else:
    print("pass")
merit
if mark >= 70:
    print("distinction")
else:
  if mark >= 60:
      print("merit")
  else:
      print("pass")
merit
```

Iterations

There are two main types of iterations in Python, the `for` and `while` loops.

`for` Loop

The `for` loops are used to iterate a collection of objects. They have the following basic form:

```
for item in container:
    perform_action
```

Can you see how the syntax in Python is different from R?

For each `item` in a `container`, `perform_action` is called once. For example the following code prints all the numbers between `1` and `5`:

```
for i in [1,2,3,4,5]:
  print(i)
1
2
3
4
5
```

Similar to R, it is possible to also terminate a `for` loop early. There are two ways to do it:

- `continue` exits the current iteration.
- `break` exits the entire for loop.

```
for i in [1,2,3,4,5]:
    if i < 2:
        continue
    # for i < 2 the code below will not be executed
    print(i)
    if i >= 4:
        break # the loop stops here
2
3
4
```

`while` Loop

The `for` loops are useful if you know in advance the set of values that you want to iterate over. If you don't know, you can use `while` loop:

```
while condition:
  action
```

For example, it is more appropriate to use `while` loop for the following example:

```
word = input("Give me a 4-letter word:")
while len(word) != 4:
    print("Wrong input!")
    word = input("Give me a 4-letter word:")
print(word)
```

As we do not know how many times we need to wait until the user gives us a required input.

Useful Links and Resources

- Guttag, J.V. (2013). Introduction to computation and programming using Python. Section 3.2.
- Official Python tutorial on control flow

Function and Scope in Python

** Note: The code chunks below should be run in the following order **

Writing a Function

A function can be defined with the following code:

```
def func_nam(arg1, arg2, arg3):
    '''
    explanation of the function
    '''
    line_1
    line_2
    ...
    return value
```

The keyword `def` tells Python that the following block of code defines a function. Again indentation is important. The explanation in the beginning of the block tells users what the function does. The `return` statement can be omitted if the function does not return any value.

```
def circle_area(radius):
    '''
    This function calculates the area of a cirle given the radius
    input: radius
    output: area = pi * radius ^2
    '''
    pi = 3.14
    return (radius**2)*pi
```

We can call the function by using the name of the function with `()`:

```
circle_area(2)
12.56
```

In contrast to R, if you want to return a value from a function, you have to use the `return`. If you write the function `circle_area` in the following way:

```
def circle_area(radius):
    '''
    This function calculates the area of a cirle given the radius
    input: radius
    output: area = pi * radius ^2
    '''
    pi = 3.14
    (radius**2)*pi
```

When you call the function nothing is returned or, in other words, the value `None` is returned.

```
circle_area(2)
```

Scope

In Python, variables are only available within the corresponding region it is created. Such region is called a *scope*.

Global and Local Scope

A variable created within the main body in Python is in the *global scope*. Their values are available anywhere. Variables created in the function belong to the *local scope*. They are only available within the corresponding function but not elsewhere. Note the local scope is created when the function is *called* but not when it is defined.

Variables From the Global Scope is Available Everywhere

Consider the following code (Example 1):

```
# Example 1
def add_a(x):
    x = x+a
    return x


a = 8
z = add_a(10)
print(z)
18
```

Does the following code work, despite not defining `a` in the function `add_a`? Yes it works because `a` is global and can be used anywhere (including inside a function).

Press `next` to visualise the code and understand what is happening when we execute each line of the code.

As we can see, we first have the function `add_a()` defined in the *global frame*. By running the fifth line we have `a = 8` in the global frame. When we call the function `add_a()` in the sixth line, we created a *local frame* `add_a`. Note that it is created when the function `add_a()` is *called* but *not* when it is defined. The object `x` is in this local frame with value `10` and it then is updated to `18` after executing the line `x = x + a`. `a` is available within the function as `a` is global so it is available everywhere. The function `add_a` hits the return statement with the return value `18`. The return value is then assigned to `z` in the global frame and the local frame `add_a` is gone.

Variables From the Local Scope is Not Available Elsewhere

Consider the following code (Example 2):

```
# Example 2
def add_b(x):
    b = 8
    x = x + b
    return x

z = add_b(10)
print(b)
```

If you run the code above, you will get the error `NameError: name 'b' is not defined`. Why is it the case? Press `next` to visualise the code and understand what is happening when we execute each line of the code.

Like the previous example, we first have the function `add_b()` defined in the *global frame*. When we call the function `add_b()` in the sixth line, we created a *local* frame `add_b`. This time the object `b` is in this *local frame* with value `8`. The function `add_b` hits the return statement. The local frame `add_b` is gone and so is the variable `b`. Therefore, we get an error when we try to `print(b)` in the main body.

LEGB Rule

Previous, we saw that if we are trying to get a variable in the local frame but it is not defined there, the global one will be used if it is available. What if the same variable name is available for the local and global scope? Consider the following code (Example 3):

```
# Example 3
def add_1(x):
    x = x + 1
    return x

x = 10
z = add_1(12)
print(z)
13
```

What is the value of `z`? What is the value of `x`? Let us look at the visualisation of the above code. Please press `next` to see how each line works behind the scene.

As we can see, we first have the function `add_1` defined in the *global frame*. By running the fifth line we have `x = 10` in the global frame. When we call the function `add_1()` in the sixth line, we first created a *local* frame `add_1`. The `x` in this local frame with value `12` is different from the `x` with value `10` in the global frame. They are two different objects. Therefore when we do `x = x+1` in the function `add_1`, we see that `x` in the `add_1` frame has changed to `13` but `x` in the global frame stays as `10`. The function `add_1` hits the return statement with the return value `13`. The return value is then assigned to `z` in the global frame and the local frame `add_1` (together with the local `x`) is gone.

The *LEGB rule* can be thought of as a name look up procedure. LEGB stands for:

- L: Local scope
- E: Enclosing scope, this exists for nested functions. The outer function is the enclosing scope for the inner function.
- G: Global scope
- B: Built-in scope, a scope that is available and automatically loaded whenever you run Python. It contains names of built-in functions, keywords, exceptions, etc. Examples are:
    - Keywords: `if`, `def`, etc.
    - Functions: `print`, `len`, etc.

Python code will find the name sequentially in the local scope ("L"), the enclosing scope ("E"), the global scope ("G"), the built-in scope ("B") based on where the executed line is in. If nothing can be found after searching all these four scopes, Python will throw an error. We can use the LEGB rule to explain the Examples 1 to 3 above:

- Example 1: We search for the variable `a` when we are in the local scope. As the variable name `a` is only available in the global scope, the `a` in the global scope is used.
- Example 2: We search for the variable `b` when we are in global scope. As the variable name `b` is not available (`b` was in the local scope but it is not available in global scope and was not retrievable after the function has returned), Python cannot find the variable `b`.
- Example 3: The variable `x` is available in both local and global scope. Following the LEGB rule, when running the line `x = x + 1`, we first search it in the local scope for `x`. As `x` is found in the local scope, the value of `x` in the global scope will not be used inside the function.

In R, a similar rule applies. See the same examples above implemented in R:

```
# Example 1
add_a <- function(x) {
  x <- x+a
  return (x)
}

a <- 8
z <- add_a(10)
print(z)
[1] 18
# Example 2
add_b <- function(x){
  b <- 8
  x <- x + b
  return (x)
}

z <- add_b(10)
print(b)
Error in print(b): object 'b' not found
# Example 3
add_1 <- function(x) {
  x <- x + 1
  return (x)
}

x <- 10
z <- add_1(12)
print(z)
[1] 13
```

Functions With Mutable Object

Consider the following code:

```
def add_one_number(seq, num):
    seq.append(num)
    return (seq)

nums = [1,3,2,4]
new_nums = add_one_number(nums, 5)
print(new_nums)
[1, 3, 2, 4, 5]
print(nums)
[1, 3, 2, 4, 5]
```

Given what you have learned about mutable objects, aliasing and scoping, are you surprised by the result? If you are confused, take a look of the following visualisation:

When dealing with mutable objects in a function, you need to be careful if there are any unintentional side-effects raised. For the example above, if you do not wish to change the original `list`, the function should be written as:

```
def add_one_number(seq, num):
    new_seq = seq.copy()
    new_seq.append(num)
    return (new_seq)

nums = [1,3,2,4]
new_nums = add_one_number(nums, 5)
print(new_nums)
[1, 3, 2, 4, 5]
print(nums)
[1, 3, 2, 4]
```

Useful Links and Resources

- Guttag, J.V. (2013). Introduction to computation and programming using Python. Chapter 4.
- Python Tutor: A tool to visualise your Python code
- An official Python tutorial on functions

Exceptions and Error Handling in Python

** Note: The code chunks below should be run in the following order **

Exceptions in Python

Until now, you probably have seen different error messages when you write a Python program. Reading the error messages is often the first step to figuring out what is wrong in your code. There are two main types of error:

- `SyntaxError`: When your syntax is not correct and Python cannot parse your program. `IndentationError` is one example of `SyntaxError`. It occurs when you do not indent blocks of code appropriately.
- `Exceptions`: Something else is wrong even your syntax is correct. There are many different types of exceptions and below we list some of the common ones:
    - `NameError`: Attempt to retrieve a variable that is not defined in the local or global scope.
    - `TypeError`: Providing arguments with some inappropriate types.
    - `ValueError`: Type of the argument is correct but the value is not.
    - `IndexError`: When you try to access an element with the index that is out of range.

There are many more types of exceptions, and you can learn more from here.

The following blocks of code trigger different error messages in Python:

```
# SyntaxError
print "hello"
```

which gives the error message `SyntaxError: Missing parentheses in call to 'print'. Did you mean print("hello")?`.

```
# IndentationError
def print_hello():
print("hello")
```

which gives the error message `IndentationError: expected an indented block`.

```
# NameError
hello
Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'hello' is not defined


Detailed traceback:
```

```
  File "<string>", line 1, in <module>

# TypeError

1+'2'

Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: unsupported o
perand type(s) for +: 'int' and 'str'


Detailed traceback:

  File "<string>", line 1, in <module>

# ValueError

int('a')

Error in py_call_impl(callable, dots$args, dots$keywords): ValueError: invalid lite
ral for int() with base 10: 'a'


Detailed traceback:

  File "<string>", line 1, in <module>

# IndexError

num = [1,2,3]

num[3]

Error in py_call_impl(callable, dots$args, dots$keywords): IndexError: list index o
ut of range


Detailed traceback:

  File "<string>", line 1, in <module>
```

Handling Exceptions

Until now, whenever we encounter an exception, we let the program crash and we go back to the code to try out the problem and to fix it. It is not ideal and for some situations, it is not even a feasible strategy. For example, consider the following example:

```
nomin_str = input("Enter an integer: ")

nomin = int(nomin_str)

denom_str = input("Enter another integer: ")

denom = int(denom_str)

num = nomin / denom

print(f"Dividing {nomin} by {denom} gives {num}.")
```

As the input is from the users, we cannot guarantee that they will always provide a valid input. If they accidentally give `"a"` as the input, then you will get the error `ValueError: invalid literal for int() with base 10: 'a'` and the program will crash. Even if numbers are provided as the input, if the second number provided is `0`, then we will get a `ZeroDivisionError`.

Using `try-except` to Handle Exceptions

A better approach is to handle the exception by the `try` and `except` blocks:

```
try:
    nomin_str = input("Enter a number: ")
    nomin = int(nomin_str)
    denom_str = input("Enter another number: ")
    denom = int(denom_str)
    num = nomin / denom
    print(f"Dividing {nomin} by {denom} gives {num}.")
except:
    print("Wrong input.")
```

If an exception is raised in the `try` clause during execution, the rest of the `try` clause will be skipped. The exception will be handled by the `except` statement and the `except` clause will be executed. Now the program does not crash even if the user provides a wrong input, only the printout `"Wrong input."` is given.

We can use separate `except` clauses to handle different types of exceptions:

```
try:
    nomin_str = input("Enter an integer: ")
    nomin = int(nomin_str)
    denom_str = input("Enter another integer: ")
    denom = int(denom_str)
    num = nomin / denom
    print(f"Dividing {nomin} by {denom} gives {num}.")
except ValueError:
    print("Input provided cannot be converted into a number.")
except ZeroDivisionError:
    print("Zero cannot be the denominator.")
except:
    print("Some other error.")
```

The `ValueError` handles the case when users provide non-numerical input. `ZeroDivisionError` handles when the second input is zero. The last `except` handles all other possible exceptions. Try to run the code above with different inputs, and see what printout you will get.

We can also use the following with the `try` and `except` clauses:

- `else`: The clause will be executed if *no* exception is raised
- `finally`: The clause will *always* be executed no matter there is an exception or not. It will be executed even if there is a `break`, `continue` or `return`.

Try to run the following code and test your understanding on how `except`, `else` and `finally` work.

```
count = 0
while True:
    try:
        nomin_str = input("Enter an integer: ")
        nomin = int(nomin_str)
        denom_str = input("Enter another integer: ")
        denom = int(denom_str)
        num = nomin / denom
    except ValueError:
        print("Input provided cannot be converted into an integer.")
    except ZeroDivisionError:
        print("Zero cannot be the denominator.")
    except:
        print("Some other error.")
    else:
        print(f"Dividing {nomin} by {denom} gives {num}.")
        break
    finally:
        count += 1
        print(f"Attempt {count}.")
```

`try-except` or `if-else` Statement?

The code above can be rewritten by `if-else` instead of the `try-except`:

```
nomin_str = input("Enter an integer: ")
denom_str = input("Enter another integer: ")
if nomin_str.isdigit() and nomin_str.isdigit()
    nomin = int(nomin_str)
    denom = int(denom_str)
    if denom != 0:
        num = nomin / denom
        print(f"Dividing {nomin} by {denom} gives {num}.")
    else:
        print("Zero cannot be the denominator.")
else:
    print("Input provided cannot be converted into a number.")
```

Which is a better way? In Python, EAFP (easier to ask for forgiveness than permission) is the common coding style. We first do what we expect to do by assuming the input is fine, and if the assumption proves to be wrong we catch the exceptions by the `try-except` clauses. This contrasts to "LBYL" (look before you leap) style commonly used in many other languages (e.g. C), which try to avoid the exceptions (by the `if-else` statement) instead of catching them. Therefore in Python `try-except` is better to be used instead of `if-else` when something might be wrong. If you compare both codes, the `try-except` version is probably easier to be read.

Raise Exceptions

Above we have shown that we can avoid raising exceptions by catching them. Sometimes we want to *raise* our own exceptions to give more precise information about what is wrong.

Consider the following code:

```
try:
    nomin_str = input("Enter an integer: ")
    nomin = int(nomin_str)
    denom_str = input("Enter another integer: ")
    denom = int(denom_str)
    num = nomin / denom
    print(f"Dividing {nomin} by {denom} gives {num}.")
except ValueError:
    raise ValueError("Input provided cannot be converted into a number.")
except ZeroDivisionError:
    raise ZeroDivisionError("Zero cannot be the denominator.")
except:
    raise ValueError("Wrong input.")
```

If you give `"a"` as the first input, you will get the error `ValueError: Input provided cannot be converted into a number.` This error message gives additional information to the generic error message `ValueError: invalid literal for int() with base 10: 'a'`.

Useful Links and Resources

- Guttag, J.V. (2013). Introduction to computation and programming using Python. Chapter 7.
- Official Python tutorial on errors and exceptions

Debugging in Python

** Note: The code chunks below should be run in the following order **

Debugging Tools in Python

Below are the main debugging tools in Python:

- Spyder
- Python Tutor
- Put `print()` into where you suspect there may be a bug in the program

Below we will illustrate how to use Python Tutor and function `print()` for debugging. For Spyder, we will show it in the live section. Let us consider the following code, with the function `get_sorted_numbers()` aims to return a sorted list. The list returned from the function does not seem to be sorted:

```
def get_sorted_numbers(nums):
    '''
    input: a list of numbers
    output: a list of sorted numbers
    '''
    sorted(nums)
    return nums


nums = [1,3,2,1]
sorted_num = get_sorted_numbers(nums)
print(sorted_num)
```

```
[1, 3, 2, 1]
```

What is wrong? Let us try to examine the code using Python Tutor:

We can see visually from Python Tutor that the function `sorted()` does not change the order of the list `num`. Such visualisation power is very helpful for us to understand each line of the code and see if any of the lines does not work as we have expected.

Next, we use the `print()` function to see if `nums` is actually sorted by the function `sorted()`. We put the `print()` function before and after the line `sorted(nums)` to see if the line has actually changed the list `nums`:

```
def get_sorted_numbers(nums):
    '''
    input: a list of numbers
    output: a list of sorted numbers
    '''
    print("before sort", nums)
    sorted(nums)
    print("after sort", nums)
    return nums


nums = [1,3,2,1]
sorted_num = get_sorted_numbers(nums)
before sort [1, 3, 2, 1]
after sort [1, 3, 2, 1]
print(sorted_num)
[1, 3, 2, 1]
```

We can see that the list `nums` is the same before and after `sorted()`, showing us that something must be wrong at the line `sorted(nums)`. Now we can proceed and correct the problem using something like:

```
def get_sorted_numbers(nums):
    '''
    input: a list of numbers
    output: a list of sorted numbers
    '''
    return sorted(nums)
nums = [1,3,2,1]
sorted_num = get_sorted_numbers(nums)
print(sorted_num)
[1, 1, 2, 3]
```

Assertions

Previously in “Exceptions, Error Handling and Debugging in R”, we talked about the “defensive programming” strategy. One of the ways to implement this strategy is to use `assert` statement to check if the program is running as expected. If it is not, `AssertionError` exception is raised. Assertions can be used to:

- Halt the program as soon as some unexpected conditions occur. This makes it easier to discover and locate the bugs.
- Check if the output is appropriate before returning the bad value and causing unexpected consequences.

The following function aims to sort the list of numbers:

```
def get_sorted_numbers(nums):

    sorted(nums)

    assert(len(nums)<2 or all([nums[i] <= nums[i+1] for i in range(len(nums)-1)])),
"The list " + str(nums) + " is not sorted"

    return nums
```

The `assert` statement check if the next number is at larger or equal to the previous number. If the condition is not satisfied, the `AssertionError` is raised and the error message `The list ... is not sorted` is shown. This tells us that there must be a bug in this function.

Useful Links and Resources

- Guttag, J.V. (2013). Introduction to computation and programming using Python. Section 6.2 and Chapter 7.

Classes and Programming Paradigms

** Note: The code chunks below should be run in the following order **

Classes

In the previous sections, we have defined our own functions. Here we will define our own *type* of objects by creating a new *class*. Classes allow us to bundle data and functionality together (*encapsulation*).

Problem-Solving Using Functions

Consider that we have data in the form of two dimension points, e.g. `(4, 3)`, `(1.1, 2.42)` etc. We want to store the data and be able to do some calculations on them (e.g. calculate the distance between two points). One possibility is to use tuples to store the data points (x and y coordinate value) and functions to do calculation, as shown below:

```
def point_dist(point_1, point_2):
    '''
    return the distance of the two given points.
    point_1 and point_2 should be in the form of (x1, y1), (x2, y2) with x1,x2,y1,y
2 are numbers.
    '''
    assert len(point_1) == 2 and len(point_2) == 2, "points must be 2 dimensional"
    try:
        return ((point_1[0] - point_2[0]) ** 2 + (point_1[1] - point_2[1]) ** 2) **
(1/2)
    except:
        raise ValueError("Inputs are not in the form of (x1, y1), (x2, y2) with x1,
x2,y1,y2 are numbers.")


def dist_from_origin(point):
    '''
    return the distance of a point from the origin.
    point should be in the form of (x, y) with x, y are numbers.
    '''
    assert len(point) == 2, "points must be 2 dimensional"
    try:
        return (point[0] ** 2 + point[1] ** 2) ** (1/2)
    except:
```

```
        raise ValueError("Input is not in the form of (x, y), with x,y are numbers.
")


from numbers import Number as numeric

def is_same_point(point_1, point_2):
    '''
    check if the given two points are the same.
    point_1 and point_2 should be in the form of (x1, y1), (x2, y2) with x1,x2,y1,y
2 are numbers.
    '''
    assert len(point_1) == 2 and len(point_2) == 2, "points must be 2 dimensional"
    assert all(isinstance(element, numeric) for element in point_1), "point_1 non-n
umeric"
    assert all(isinstance(element, numeric) for element in point_2), "point_2 non-n
umeric"
    try:
        return point_1[0] == point_2[0] and point_1[1] == point_2[1]
    except:
        raise ValueError("Inputs are not in the form of (x1, y1), (x2, y2) with x1,
x2,y1,y2 are numbers.")


def mid_point(point_1, point_2):
    '''
    return a mid point from the given two points.
  point_1 and point_2 should be in the form of (x1, y1), (x2, y2) with x1,x2,y1,y2
are numbers.
    '''
    assert len(point_1) == 2 and len(point_2) == 2, "points must be 2 dimensional"
    try:
        x = (point_1[0] + point_2[0]) / 2
        y = (point_1[1] + point_2[1]) / 2
        return (x, y)
    except:
        raise ValueError("Inputs are not in the form of (x1, y1), (x2, y2) with x1,
x2,y1,y2 are numbers.")
```

The above code chunks only defined the functions required for the calculations. We have to *call* the functions to execute them, for example:

```
point = (3,4)
dist_from_origin(point)
5.0
```

With the implementation above, the calculations and data are separated. Also note that any other objects not in the form of a point can also use the functions, although it is likely that an error will be raised. This implementation is in the “style” of *functional programming*, for which we solve a problem by functions:

```
point_1->[dist_from_origin]-> distance
```

Also, the function does not have any side effect (e.g. does not mutate the input, modify the global variables or have any print out), and the output of the function only depends on the input.

This is very similar to how we solve the mathematics problem \(y = f(x)\). For the example above, we have `x = (3,4)` (a point), `f = dist_from_origin` (function to calculate the distance from the origin) and `y = 5` (distance).

Problem-Solving Using Classes

Alternatively, we can define our own class to do achieve what we have done above. Have a look at the code below before we explain what it is doing.

```
class Point:
    '''
    The Point class represents a point in 2 dimensions.
    '''
    def __init__(self, x, y):
        if isinstance(x, numeric) & isinstance(y, numeric):
            self.x = x
            self.y = y
        else:
            raise ValueError('x and y must be numbers')


    def dist(self, pt):
        '''
        return the distance between the point and a given point.
        '''
        assert isinstance(pt, Point), "the argument must be a point"
        return ((self.x - pt.x) ** 2 + (self.y - pt.y) ** 2) ** (1/2)


    def dist_from_origin(self):
        '''
        return the distance of the point from the origin.
        '''
        return (self.x ** 2 + self.y ** 2) ** (1/2)


    def mid_point(self, pt):
        '''
```

```
        return a mid point from the point and a given point.
        '''
        assert isinstance(pt, Point), "the argument must be a point"
        x = (self.x + pt.x) / 2
        y = (self.y + pt.y) / 2
        return (x, y)

    def is_same(self, pt):
        '''
        check if the point and the given point is the same.
        '''
        assert isinstance(pt, Point), "the argument must be a point"
        return self.x == pt.x and self.y == pt.y
```

The class `Point` defined above bundles the data (x, y coordinates) and the functionalities (calculate the distance between two points, find the midpoint, etc.) of points together:

- Data: `self.x` and `self.y` store the x and y coordinate information of a given point. The values are assigned in the `__init__()` method.
- Functionalities: four different methods are defined. The method `distance_from_origin`, for example, calculate the distance of the point from the origin, using the point data stored in `self.x` and `self.y`.

The code in the above code chunk is written in a *object-oriented programming* (OOP) style, for which we solve the problem by creating classes.

Creating an Object With Type `Point`

As we have seen with functions, the above code chunk only *defines* the class. To use it, we need to create an instance of the object with the type `Point`, for example:

```
pt = Point(3,4)
```

This create an object with type `Point` representing a point with x and y coordinates 3 and 4.

What does Python do when we call `pt = Point(3,4)`? Python first creates an *instance* of the class Point (which is `self`) and the special initialising function `__init__()` is called automatically when an object of the class is created. `__init__()` allows the class to initialise the attributes of the instance `self`, and in our case we assign the value 3 to `self.x` and 4 to `self.y`. The instance of `Point` created is then bound to the variable name `pt`. As the instance of `Point` has the attributes `x` and `y`, we can retrieve them by using the dot notation:

```
print(pt.x)
3
print(pt.y)
4
```

Calling the Methods in `Point`

Methods are functions of a class. Every method (including `__init__()`) requires `self` to be the first argument. To use a method, we again use the dot notation. For example:

```
pt.dist_from_origin()
```

```
5.0
```

Note that while the method `dist_from_origin()` requires an argument `self`, we do not need to provide such argument when calling it, as `self` is automatically provided to the method. In terms of the result, calling `pt.dist_from_origin()` is the same as calling:

```
Point.dist_from_origin(pt)
```

```
5.0
```

And `pt` is given as `self` for the method. Note that you should always use `object.method()` (e.g. `pt.dist_from_origin()`) instead of `Class.method(object)` (e.g. `Point.dist_from_origin(pt)`), although they should in principle provide the same result.

Encapsulation

Encapsulation With Classes

Encapsulation is a concept of bundling data and methods together and restricting the direct access to data of the object. We have seen that the use of classes allows us to bundle data and methods together. We now show how we can restrict the direct access of the data with the example below:

```
class BankAccount:
    '''
    BankAccount represents the bank account, with methods to deposit and withdraw m
oney from it
    '''
    def __init__(self, balance = 0):
        assert balance >= 0, "balance has to be non-negative."
        self._balance = balance


    def deposit(self, amount):
        '''
        Add the deposit amount to the account balance.
        Deposit amount has to be non-negative.
        No return value.
        '''
        if amount < 0:
            print("Deposit fail: deposit amount has to be non-negative.")
```

```
            return 0
        else:
            self._balance += amount
            return amount

    def withdraw(self, amount):
        '''
        Deduct the withdraw amount from the account balance.
        Withdraw amount has to be non-negative and not greater than the balance.
        Return the value of the withdrawn amount
        '''
        if amount < 0:
            print("Withdraw fail: withdraw amount has to be non-negative.")
            return 0
        elif self._balance >= amount:
            self._balance -= amount
            return amount
        else:
            print("Withdraw fail: withdraw amount is more than the balance.")
            return 0

    def get_balance(self):
        '''
        Return current balance
        '''
        return self._balance
```

One thing you may have noticed is that we defined the class attribute for the account balance as `_balance`. Any variable name with one underscore (`_`) indicates that such variable is intended for internal use. Pressing `tab` after typing `object.` in a Python terminal lists all the methods and data that are visible to users. For example for the `Point` object `pt`, we can see all the methods and data that we have defined:

```
>>> pt = Point(3,4)
>>> pt.
pt.dist(              pt.dist_from_origin(  pt.is_same(          pt.mid_point(
pt.x                  pt.y
```

However, we do not see `_balance` for an instance of `BankAccount`:

```
>>> account_a = BankAccount(100)
>>> account_a.
account_a.balance(   account_a.deposit(   account_a.withdraw(
```

Why do we want to hide the account balance? It is because we do not want users to work on the *internal* data directly. For example, users may change the balance to negative, which is not a valid account balance. Instead, we provide a *getter* method `get_balance()` to allow users to see the account balance. If they want to change the amount, they should use the `withdraw()` or `deposit()` functions, as the functions check the withdraw and deposit amount to make sure the balance is always non-negative. The code below illustrates how the balance amount is updated with `withdraw()` and `deposit()`:

```
account_a = BankAccount(100)
account_a.deposit(20)
20
print(account_a.get_balance())
120
money = account_a.withdraw(200)
Withdraw fail: withdraw amount is more than the balance.
print(account_a.get_balance())
120
money = account_a.withdraw(50)
print(account_a.get_balance())
70
```

By encapsulation and data hiding with class, we prevent users from manipulating the data in an arbitrary way and ensure data integrity.

One thing to notice is that while `_balance` is marked for internal use, Python still allows you to change the value:

```
account_a._balance = -100
print(account_a.get_balance())
-100
```

You *should not* do it in this way (even you are able to do it).

Functions

If we want to rewrite the solution with functions instead of creating a new class, we can do it in this way:

```
def deposit(balance, deposit_amount):
    '''
    Return the balance with the deposit amount added to the account balance.
    Deposit amount has to be non-negative.
    '''
    if deposit_amount < 0:
        print("Deposit fail: deposit amount has to be non-negative.")
        return balance
```

```
    else:
        return balance + deposit_amount

def withdraw(balance, withdraw_amount):
    '''
    Return the balance with the withdraw amount deducted from the account balance.
    Deposit amount has to be non-negative and smaller than the balance
    '''
    if withdraw_amount < 0:
        print("Withdraw fail: withdraw amount has to be non-negative.")
        return balance
    elif balance >= withdraw_amount:
        return balance - withdraw_amount
    else:
        print("Withdraw fail: withdraw amount is more than the balance.")
        return balance

balance = 100
balance = deposit(balance, 20)
print(balance)
120
balance = withdraw(balance, 200)
Withdraw fail: withdraw amount is more than the balance.
print(balance)
120
balance = withdraw(balance, 50)
print(balance)
70
```

Note data can be directly accessed, and functions and data are separated.

Inheritance and Polymorphism

Classes

Inheritance occurs when we define a new class (call the *child* class) that is based on an existing class (the *parent* class). Let us have a look of the following code:

```
class OverdraftAccount(BankAccount):
    '''
    OverdraftAccount represents a bank account with overdraft limit
```

```
    '''
    def __init__(self, balance, overdraft_limit):
        assert overdraft_limit > 0, "overdraft limit has to be non-negative."
        assert balance > -overdraft_limit, "balance exceeds overdraft limit"
        self._balance = balance
        self._overdraft_limit = overdraft_limit

    def withdraw(self, amount):
        '''
        Deduct the withdraw amount from the account balance.
        Withdraw amount has to be non-negative and not greater than the balance wit
h overdraft limit.
        Return the value of the withdrawn amount
        '''
        if amount < 0:
            print("Withdraw fail: withdraw amount has to be non-negative.")
            return 0
        elif self._balance + self._overdraft_limit >= amount:
            self._balance -= amount
            return amount
        else:
            print("Withdraw fail: overdraft limit does not allow this withdrawal")
            return 0

    def get_overdraft_limit(self):
        '''
        Return the overdraft limit
        '''
        return self._overdraft_limit
```

The first line of the code tells Python to define a new class `OverdraftAccount` based on the `BankAccount`. It inherits *all* the methods from `BankAccount`, and that is why we do not need to define `deposit()` or `get_balance()`.

Note we do define other methods that are already available in `BankAccount` like `withdraw()`. This is because we want the `withdraw()` method for `OverdraftAccount` to behave differently to the `withdraw()` method for `BankAccount` to reflect that the `OverdraftAccount` allows negative balance as long as it is under the overdraft limit. This is called *polymorphism*, for which a single interface to entities of different types.

Let us rerun the code above except we change the account type from `BankAccount` to `OverdraftAccount`:

```
account_b = OverdraftAccount(100, 300)
account_b.deposit(20)
```

```
20
print(account_b.get_balance())
120
money = account_b.withdraw(200)
print(account_b.get_balance())
-80
money = account_b.withdraw(50)
print(account_b.get_balance())
-130
```

Now we allow withdrawal that results in a negative balance.

There are two more methods defined for `OverdraftAccount`. One is `__init__()`, which we redefine so that we can initialise the overdraft limit, and check the `balance` argument differently (to reflect the balance can be negative). Another one is `get_overdraft_limit()`, which is not available in `BankAccount`. While `OverdraftAccount` has all the methods that `BankAccount` has, it can have *more* methods that the parent class.

Functions

For the function approach, we *modify* the function `withdraw()` in order to accommodate overdraft limit.

```
def withdraw(balance, withdraw_amount, overdraft_limit = 0):
    '''
    Return the balance with the withdraw amount deducted from the account balance.
    Deposit amount has to be non-negative and smaller than the balance
    '''
    if withdraw_amount < 0:
        print("Withdraw fail: withdraw amount has to be non-negative.")
        return balance
    elif balance + overdraft_limit >= withdraw_amount:
        return balance - withdraw_amount
    else:
        print("Withdraw fail: withdraw amount is more than the balance.")
        return balance


balance = 100
overdraft_amount = 300
balance = deposit(balance, 20)
print(balance)
120
balance = withdraw(balance, 200, overdraft_amount)
print(balance)
```

```
-80
balance = withdraw(balance, 50, overdraft_amount)
print(balance)
-130
```

When using `class` for this question, we define a new child class `OverdraftAccount` to accommodate the other type of bank account that has overdraft limit. We *do not* need to change the original implementation of `BankAccount`. However, with the use of `function`, we do need to write a new function but we need to modify the original implementation of `BankAccount`.

Programming Paradigms

Programming paradigm is a way or style to program. The programs we have written so far can be described as:

- *Procedural*: Programs are lists of instructions that tell the computer what to do with the program's input.
- *Functional*: Programming decomposes a problem into a set of functions, and avoids change of states as much as possible
- *Object oriented*: Programming decomposes a problem into a set of objects.

All programs we have written before this week were mostly procedural with functional *style*. We solve the problems by writing functions or a list of procedures.

The code we have written so far in Python is not actually *functional*. For functional programming, functions should only take inputs and generate outputs. It should not have any side effect (like `print()` or mutate a `list`). It also should not have any internal state that affects the output produced for a given input. While we have highlighted the danger of mutating the input and using global variables in a function, we do not prohibit the use of them. Also, we often have some internal states in a function (e.g. `x = (point_1[0] + point_2[0]) / 2` in the function `mid_point()`). Nevertheless, we have written our code in the functional *style*, in the sense that we solve the problem by a flow of functions. For the rest of the note, the word *functional programming* refers to the *style* of decomposing the problem into a series of functions, and avoid the side effect if possible.

This week we introduced how to write our own class, and we can solve problems by writing different objects. Above we have seen that in Python we can solve the same problem with a more procedural/functional programming approach or a more object oriented approach. Python is a *multiple programming paradigm* language, and it supports all three programming paradigms mentioned above.

Which paradigm to use depends on the situation:

- How the problem can be naturally represented: If the problem can be easily / naturally to be decomposed into objects, then object-oriented programming (OOP) is likely to be more suitable. If the problem can be easily/naturally be decomposed into functions, then functional programming style is likely to be more suitable.
- Readiness to use: OOP tends to require more code that takes longer to write compared to functional programming style. So if you want something fast, functional programming style may be more suitable
- Future update: If you foresee that you will have some more new types of objects with the same interface (e.g. to have more different types of bank accounts in the future with the same `withdraw()`, `deposit()`) interface, then OOP is more suitable as what you need to do is to create a new child class. If you foresee that you will have the same type of objects but likely to have new functionalities (e.g. data exploration when you are unsure

what analysis to do on a piece of data), then functional programming style is more suitable as what you need is to create a new function.
- Data hiding: It is easier to do data hiding in OOP than functional programming style.

Remember different programming paradigms are different tools for you to solve a problem. You do not have to stick to one paradigm, and it is likely that you will use a mixture of paradigms for a program.

Object-Oriented Programming in R

R provides multiple systems for OOP (e.g. `S3`, `S4`, `R6`). `S3` and `S4` are provided by base R, while `R6` is provided by the `R6` R package. Generally in R, functional programming is much more important than OOP because it is easier to solve complex problems by breaking them down into simpler functions than simple objects. See Chapter 10 of the Advanced R textbook for more details.

The example below illustrates how polymorphism is provided by the default `S3` OOP system in R:

```
x <- rnorm(100)
y <- x + rnorm(100)
mod <- lm(y~x) # fit a linear model

print(class(x))
[1] "numeric"
print(class(mod))
[1] "lm"
summary(x)
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
-2.290494 -0.734283 -0.008068 -0.050195  0.743667  2.429998
summary(mod)

Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-3.3788 -0.7515 -0.0380  0.8287  2.3064

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 0.007164   0.106501   0.067    0.947
x           1.006858   0.105160   9.574 1.02e-15 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 1.064 on 98 degrees of freedom
Multiple R-squared:  0.4833,    Adjusted R-squared:  0.478
F-statistic: 91.67 on 1 and 98 DF,  p-value: 1.015e-15
```

Note here the function `summary()` behaves differently depending on the type of objects passed to it.

Useful Links and Resources

- Guttag, J.V. (2013). Introduction to computation and programming using Python. Chapter 8.
- Official Python tutorial on classes
- Functional programming how to
- Chapter 10 from “Advanced R”