Python: Control Flow and I/O

This lecture discusses the basics of control flow in Python, i.e. conditionals, loops and functions. It also discusses how to read and output files.

1 Control Flow

Control flow refers to the tools we have available to control the execution of our script: conditionals (if, else, elif), loops (while, for) and functions.

1.1 Conditionals

The syntax for `if` statements is:

```
if condition:
    # if condition evaluates to True
    # run code in here
else:
    # if condition is False
    # run code in here instead
```

Notice the use of a colon after the condition and after the `else` keyword. Also observe that we have indented the expressions after the `if` and `else` keywords. Contrary to other languages that use brackets or the `end` keyword to delimit a scope, Python uses indentation. The indentation indicates what is inside the `if` block and what is not. Finally, notice that semicolons are not required by Python, you can still use them but they are not required, instead Python uses linebreaks to find the end of the command.

The condition that comes after the `if` keyword is an expression that evaluates to a boolean. If that is true, then the indented code after `if` is executed. If the expression evaluates to false, then the indented code after `else` is executed. For example:

```
variance = -2
if variance < 0:
    print('Your variance is negative, what are you doing?')
else:
    print('Variance is positive, phew!')
```

We can also have an `if` without an `else`:

```
say_hi = True
if say_hi:
    print("Hi!")
```

We can test multiple conditions with `elif`:

```
variance_type = 'RV'
if variance_type == 'RV'
    print('Computing realized variance')
elif variance_type == 'BV':
    print('Computing bipower variance')
else:
    print('Computing truncated variance')
```

If the first two conditions are not executed then the last condition (the default) is.

It is possible to use several `elif` to test more conditions.

```
if x < -1:
    doSomething
elif x >= -1 and x <= 1:
    doSomethingElse
elif x > 2 and x != 3:
    doYetAnotherThing
else:
    # x == 3
    doOtherThing
```

1.2 Loops

In Python, we can write loops with `while` and `for`.

A while loop evaluates an expression, and, if it is `True`, a block of is executed and the loop repeats. The syntax for a `while` is:

```
while condition:
    # if the condition is true
    do something here
```

The code inside the `while` loop will be executed as long as the `condition` evaluates to True.

For example:

```
total = 10
i = 0
print("Beginning while loop:")
while i < total:
    i += 1                      # equivalent to i = i + 1
    print(i)
```

A `for` loop can be written as:

```
for i in sequence:
    do something here
```

A for loop iterates the variable `i` over the values in `sequence`. For each value `i` takes, the code block is executed.

For example:

```
students = ['A', 'B', 'C', 'D', 'E']
for i in [0, 1, 2, 3, 4]:
    print(students[i])
```

Remember that we have two built-in functions that can help here. The function `range` can create a list of numbers like the one in the for loop above. And the function `len` returns the length of an object. So:

```
students = ['A', 'B', 'C', 'D', 'E']
for i in range(len(students)):
    print(students[i])
```

For loops in Python really behave like a "for-each" loop. We can iterate over the elements of `students` directly, without the need to use an index: for each `i` in `students` do something.

```
students = ['A', 'B', 'C', 'D', 'E']
for name in students:
    print(name)
```

The container with the students takes care of giving out each of the elements without explicitly tracking the index of elements. If the container (in this case a list) is ordered, then the elements will be processed in the same order.

If you really need to access the indices then you can use the built-in function `enumerate` to iterate over the indices and the student names at the same time. Let's first look at what `enumerate` does:

```
x = [100, 200, 300]
print(enumerate(x))              # does not reveal much
# Convert the enumerate object to a list, so that we can see what's inside
print(list(enumerate(x)))
```

Each element in the enumerated list is a tuple! The first element of the tuple is the index of the second element of the tuple on the original list. For example, the first tuple is `(0, 100)`, where `100` is the first value in `x`, and `0` is the index of `100` in `x`.

Tuples can be unpacked into many variables at the same time:

```
y = (0, 100)
index, value = y
index, value = (0, 100)
```

This type of assignment is called "tuple unpacking" (or list unpacking), and assigns the value `0` to the name `index`, and the value `100` to the name `value`.

We can combine `enumerate` with tuple unpacking in a for loop!

```
for i, name in enumerate(students):
    print('students[' + str(i) + ']=' + name)
```

The iterator `enumerate(students)` returns more than one value (it returns two values at each iteration). In general, iterators can return a `tuple` with any number of values, and these values are unpacked in the for-loop. In this case, the `enumerate(students)` returns a tuple with two elements, the first is an integer representing the index (starting at 0), and the second is one of the elements of the students list. These two values are unpacked, the first one is assigned to the variable `i`, and the second is assigned to the variable `name`.

We can also use dictionaries in a for-loop.

```
studentsGrades = {'A': 10, 'B': 9, 'C': 9.5, 'D': 8}
for i in studentsGrades:
    print(i)
```

This for loop will iterate over the keys of the dictionary, so that at each iteration `i` receives one of the keys, such as `'A'` or `'B'`.

We can also iterate over keys and values at the same time using unpacking. To do so, we use the `items` method of dictionaries. The method returns a list of tuples, where the first element of the tuple is the key, and the second element of the tuple is the value:

```
dir(grades)
help(grades.items)
for name, grade in studentsGrades.items():
    print(f"{name}'s grade was {grade}")
```

In some other languages a for-loop is written with the following syntax:

```
for (int i = a; i < n; i += s) {
do stuff here
}
```

This type of loop can be written in Python using the function `range`:

```
for i in range(a, n, s):
    # do things here
```

The function `range(a, n, s)` will create a list of numbers starting at the value `a`, stops before the value `n`, and has a step size `s`.

1.3 Functions

Functions can be defined with the `def` keyword and there is no need to specify the type of the return (remember values in Python are inferred when they are assigned). The body of the function starts in the next line and must be indented.

```
def fibonacci(n):
    """
    Writes a list of the first n numbers after the two initial values of
    Fibonacci series.

    Input:
     n (int): number of elements to obtain from the Fibonacci series

    Output:
     series (list): list of elements from the Fibonacci series
    """
    previous, current = 1, 1
    series = [previous, current]
    for i in range(n):
        current, previous = current + previous, current
        series.append(current)
    return series
```

After the `def` keyword comes the name of the function, and then parentheses `()`. Inside the parentheses are the name of the variables, if any, and then comes a colon.

The body of a functin starts with its documentation string (docstring). The triple quotes define a multi-line comment and it is a good practice to add the definition of the function, or at least what it is supposed to do and output.

We can execute this code cell so that the function `fibonacci` is available for use. We can now call this function:

```
fibonacci(10)
```

Notice that calling the function `help` on or just defined function will output its documentation:

```
help(fibonacci)
```

The body of the function defines a local scope. Any variables that are assigned in the body of the function first refer to the local scope, and then to the global scope.

```
message = "Hello World"
def sayHello():
    message = "Hello World!!!!"
    print(message)

print(message)
sayHello())
print(message)                    # prints Hello World, not Hello World!!!!
```

The variable `message` is first defined outside the function, it is a global variable. Inside the `sayHello` function we define a variable with the same name, it is a local variable and does not overwrite the global variable with the same name. This becomes clear when we print the value of `message` outside of the function.

Notice that the `sayHello` function has no `return` keyword and does not return any value. A function that has no `return` keyword implicitly returns a value of `None`. That is, Python impicitly appends a `return None` to the function that has no `return` value. The body of the function ends when the indentation ends, so there is no need to use brackets or an `end` keyword. When the value of `None` is the only one to be returned by a function, the REPL suppress printing it on the screen.

```
sayHello()                        # does not print the return value
print(sayHello())                 # prints the return value, which is None
a = sayHello()
a == None                         # True
```

1.3.1 Functions are First-Class Objects

Functions are first-class objects, which means they are treated the same as any other object in the language. Remember, everything in Python is an object. Functions can be created, destroyed (this is done automatically by Python), passed to other functions, assigned to a variable, returned from another function.

```
type(sayHello)
print(sayHello)                   # prints info, address of function in
    memory
a = sayHello
print(a)                          # same address
```

When a function is defined and loaded by the Python interpreter it is assigned some space in memory. When we ran the code that defined `sayHello`, the function is created and assigned some space in memoy. The name `sayHello` then points to that space in memory. When we assign it to the variable `a`, then `a` starts pointing at the same space in memory.

```
del sayHello
print(sayHello)                    # name not defined
print(a)                           # the function lives!
a()
```

We can pass functions as inputs to other functions, and even return a function from another function:

```
def greetingFactory(name):
    def greet():
        print(f"Welcome, {name}!")
    return greet
```

The function defined above takes a name as an argument, and creates a function that prints a message using the name. It then returns the newly defined function. This function that was returned remembers the value of `name` that was used in its construction, this is called a 'closure'. A closure is a function that remembers the values of the variables in the enclosing scope (the scope of `greetingFactory`).

```
name = 'Guilherme'
greetMyself = greetingFactory(name)
greetMyself()
greetingFactory('Doe')()
```

1.3.2 Default Values

Functions can take arguments with default values! (This is not the case in Matlab, where default values is not straightforward)

```
import math


def estimate_variance(returns, type_='RV'):
    if type_ == 'RV':
        rv = 0
        for r in returns:
            rv += r**2
        return rv
    elif type_ == 'BV':
        bv = 0
        for i in range(2, len(returns)):
            bv += abs(returns[i-1]*returns[i])
        return (math.pi/2)*bv
```

This function can be called in several different ways:

```
returns = [0.2, 0.3, -.01, -0.5]

estimate_variance(returns)
estimate_variance(returns, 'RV')
estimate_variance(returns, 'BV')
```

You can even specify which variable you are assigning the value to by using name and value pairs:

```
estimate_variance(returns, type_='BV')
estimate_variance(returns=returns, type_='BV')
estimate_variance(type_='BV', returns=returns)
```

You can use other variables to hold the default values for inputs in a function. However, the default values are captured (evaluated) when the function is first defined. That is, even if the value of the variable changes, the default value will not change:

```
i = 10
def f(arg=i):
    print(arg)
i = 20
f()                                  # prints 10
```

2 Input and Output

We now turn to loading and saving files with Python. To begin, let's create a new file and save some data in it. Open a file for writing with `open`:

```
f = open('file.txt', 'w')
# f is an object representing the file
# f has a method for writing to the file: f.write
# f.write takes a string and outputs it to the file
f.write('column1,column2,column3\n')
# \n is a special character that represents a new line
f.write('1.23,2.34,0.3,4.5')
# Close the file so that it is saved
f.close()
```

The file was created in the working directory Python is using. If you are in jupyter notebook, the working directory will be the folder where you launched jupyter notebook from using the terminal. You can use the built-in package `os` to find the working directory:

```
import os
os.getcwd()
```

You can change the path to elsewhere with `os.chdir`. We can inspect the file outside Python by opening it with a text editor.

Let's open the file for reading now:

```
f = open('file.txt', 'r')        # r for reading only
# We can go through the file in different ways
help(f.read)
# f.read: Reads until EOF (end of file) and returns a string with
    everything
data = f.read()
print(data)
# Print string without interpreting the special character \n as a new line
print(repr(data))
```

Python uses a cursor to go through the file. When we called f.read, this cursor moved through all characters of the file and stopped at the end of the file. Thus, if we call f.read again, Python will start reading the file from where the cursor is at. Since the cursor is already at the end of the file we will get an empty string, since there is nothing to read:

```
print(f.read())
print(repr(f.read()))
# To re-read the file, we can either close it and open it again
f.close()
print(f)
```

```
f.read()                          # file is closed
f = open('file.txt', 'r')
# Now, the cursor is back at the beginning of the file
print(f.read())
# The cursor is back at the end of the file
# Move the cursor back to the beginning with f.seek
help(f.seek)
f.seek(0)
print(f.read())
f.seek(0)
# We can read line by line with f.readline
print(f.readline())
# reads until it finds \n, and outputs the result
# Read the next line
print(f.readline())
# Read the next line, which does not exist
print(f.readline())               # empty string
```

The file object is also an iterator. It can be used in a for-loop in a way similar to a list of numbers. But, in this case, we have a list of lines:

```
f.seek(0)
for line in f:
    print(line)
# After we are done with the file, we can close it
f.close()
```

Notice we had to open and close the file. We can use something called a context manager to facilitate the job of opening and then automatically closing the file for us. To do so, we use the `with` keyword:

```
with open('file.txt', 'r') as f:
    # Inside this block, f represents the file.
    # We can now read the file however we prefer.
    data = f.read()
# When we exit the "with" block, it will take care of closing
# the file for us. The file will be closed even if there is an
# error which interrupts the execution of the code.
print(data)
```