Module 2 Part 1: Introduction to Python

Introduction

This module is designed as an introduction to the Python programming language. It covers the basic syntax of Python, main data types and most used data collections with examples.

This module consists of 3 parts:

- **Part 1** - Introduction to Python.
- **Part 2** - Python Strings and Lists.
- **Part 3** - Python Tuples, Dictionaries, Reading data from a file, Formatting print output.

Each part is in a separate notebook. It is recommended to follow the order of the notebooks from Part 1 to Part 3.

Learning Outcomes

In this module, you will learn and practice:

- Basic Python syntax
- Main data types in Python
- Variables and expressions
- Python modules and built-in functions
- How to design function in Python
- Python strings, lists, tuples and dictionaries
- How to work with files in Python

Readings and Resources

The majority of the notebook content draws from the recommended readings. We invite you to further supplement this notebook with the following texts:

1. Downey, A. (2015). *Think Python. How to Think Like a Computer Scientist*. O'Reilly Media.

> **NOTE:** This book is also available online as a Free Book from Green Tea Press and can be retrieved from http://greenteapress.com/wp/think-python-2e/. Green Tea Press.

2. McKinney, W. (2017). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython* (2nd edition). O'Reilly Media.
3. VanderPlas, Jake (2016). *Python Data Science Handbook. Essential Tools for Working with Data*. O'Reilly Media.

> **NOTE:** This books is also available online on author's GitHub page and can be retrieved from https://jakevdp.github.io/PythonDataScienceHandbook/.

Table of Contents

Defining a Problem and Preparing the Data

This image shows the stages of creating a model from start to finish.

This image shows the stages of creating a model from start to finish. (Course Authors, 2018)

Models are used in a variety of industries to understand business problems and predict potential outcomes. This allows organizations to make data-driven recommendations and decisions. As we saw in the last module, the first step to building a predictive model is to define the problem we are trying to solve. This will aid us in selecting and preparing data for our model.

Defining a Problem

Projects often fail when a problem has not been clearly defined. As a data scientist, you will often help a line of business validate and solve a problem — so it is critical for all stakeholders to be the same page about what the problem is and its potential causes.

When defining the problem, consider some of the following business questions:

1. **What is the outcome you're trying to understand?** e.g.

- Low revenue
- High expenses
- Lack of customer response to a product
- High production error rate

2. **What are the inputs that may lead to the outcome?** e.g.

- Customer profiles
- Store locations
- Raw materials

3. **Which parties are involved?** e.g.

- Front-line employees
- Customers

4. **What is the time frame of the problem?** e.g.

- Over the past year
- Since product inception

Having a clear problem statement will help you build hypothesis to inform your models. You can use the SMART method to build your problem statement. SMART stands for:

1. Specific,
2. Measurable,
3. Action-oriented,
4. Relevant,

5. and Time-bound.

Here is an example problem statement following the SMART method:

> "Credit card sales in Ontario, our largest division, have decreased 35% since January, despite employing traditional sales tactics at the branch. We need to understand which customer-driven attributes are leading to the sale decline so we can adjust our product, marketing and sales strategy."

As you can see, the problem statement is Specific (about credit card sales in Ontario), Measurable (decrease of 35%), Action-oriented (understanding the drivers will help us re-evaluate our strategy), Relevant (this is our largest business), and Time-bound (this has been happening since January).

You can use analytics to determine the root causes leading to your problem. Once you have an understanding of the drivers which cause the problem, you can use predictive modeling techniques to assess how a change in strategy will influence an outcome.

When the problem has been defined, you can select which data points are relevant to your analysis. For example, if trying to understand a customer base, you may choose to look at information such as age, purchase behavior, product selections, city of residence, and method of purchase.

Preparing the Data

The first step of data preparation is exploration. The purpose of this is to understand what data is available from which sources and whether or not this data can help you in solving your problem.

In this course, we will be using the programming language Python for our data analysis and model development. Before working with real data, in the following sections you will learn the language structure and basic syntax for common functions. This will help you apply the appropriate functions depending on different data types, for example numeric vs. alphanumeric data. In part 3 of this module, you will also learn to read and write data to and from a file. This is an important phase of model development, as it will help you identify potential data inefficiencies early in your journey.

Why Python?

Python is one of the most widely used programming languages and is very popular among data scientists. It is generally considered a first choice for social media data mining. It is widely used in the analysis of sociological data and its use in the financial industry has been increasing rapidly since 2005.

Increases in Python usage were led largely by maturation of analytics libraries, such as NumPy and Pandas to allow working with DataFrames, SciPy, and Scikit-learn to provide statistics and machine learning algorithms.

Python is well-suited as an interactive analysis environment. It can also enable the development of robust systems in a fraction of the time it would have taken in Java or C++. It supports a mixture of procedural, object-oriented, functional, and imperative styles. Python has a reputation for being relatively easy to learn.

Data types, variables and assignments

Python is an interpreted programming language. There are two ways to use the interpreter: **interactive** and **script** modes.
In interactive mode, we can type a line of Python code and the interpretor processes it immediately and displays the result.

In [1]:
```
1 + 2
```

Out[1]:
```
3
```

Alternatively, the code can be stored in a file, which sometimes is called a script. The content of the file can be executed by the interpreter. By convention, Python scripts have filenames that end with `.py`. One of the benefits of working with an interpreted language is that you can test bits of code in interactive mode before it is put in a script.

In the code, a variable can be created and assigned to a value. In Python, a variable name can be arbitrarily long and can contain both letters and numbers, but must start with a letter. Both uppercase and lowercase letters can be used, but it is often recommended to start with lowercase. The underscore sign can appear in a name and a name may start with underscore. Quite often, the underscore sign is used between words in long names. This style is sometimes referred to as **snake case**.

It is important to remember that Python 3 has 35 reserved keywords that cannot be used as variable names. Below is the list of Python 3.6 reserved keywords in their exact spelling (McKinney, 2017):

```
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
```

The main data types in Python are **integer**, **float**, **string**, and **boolean**. For example,

- `1` is an integer
- `1.2` is a float
- `"Hello, world!"` is a string
- `"1.2"` is also a string.

Any string of characters enclosed in either single or double quotation marks is a value of type `string`.

Python is a dynamically-typed language. The type of a variable is set when a value is assigned to it. There is no need to declare a variable's type. Variable assignment to a value is easy and concise. This also allows us to change the type of a variable farther down in the code. The built-in function **type** returns the type of an object.

Below are some examples of assignments.

In [2]:
```
radius = 8
pi_number = 3.14

area_of_circle = pi_number * radius**2

print(area_of_circle)
print(type(area_of_circle))
```

```
200.96
<class 'float'>
```

In [3]:
```
my_string = "Hello, world!"

print(my_string)
print(type(my_string))
```

```
Hello, world!
<class 'str'>
```

In [4]:
```
a = True

print(a)
print(type(a))
```

```
True
<class 'bool'>
```

In Python, assignment may take more complex form. Below is an example of a **chained** assignment:

In [1]:
```
a = b = c = 10

print(a)
print(b)
print(c)
```

```
10
10
10
```

Generally, the chained assignment has the form:

`x0 = x1 = ... = xN = value`.

Another form of assignment is `y0, y1 = value_1, value_2`. In this case, the variable `y0` is assigned to `value_1` and the variable `y1` is assigned to `value_2`.

In [6]:
```
x, y, z = 10, 'plus', 20

print(x)
print(y)
print(z)
```

```
10
plus
20
```

Statements and expressions

A **statement** is a unit of code that the Python interpreter can execute. For example, an assignment is a statement. An **expression** is a combination of values, variables, and operators. A value all by itself is considered an expression. Technically, an expression is also a statement, but it is probably simpler to think of them as different things. The important difference is that an expression has a value; a statement does not.

Python supports the following operators on numbers: addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`). Python also supports integer division (`//` , remainder or modulo (`%`), and exponentiation (`**`).

In [7]:
```
'''Addition:'''

4 + 7
```

Out[7]:
```
11
```

In [8]:
```
'''Subtraction:'''

4 - 7
```

Out[8]:
```
-3
```

In [9]:
```
'''Multiplication:'''

4 * 7
```

Out[9]:
```
28
```

In [10]:
```
'''Division:'''

4 / 7
```

Out[10]:
```
0.5714285714285714
```

In [11]:
```
'''Integer division:'''

4 // 7
```

Out[11]:
```
0
```

In [12]:
```
'''Modulo - divides 2 numbers and returns the remainder'''

4 % 7
```

Out[12]:
```
4
```

Below is an example of using the integer division and modulo operators.

In [13]:
```
'''Example: The run time of a movie is 135 minutes.
How long the movie runs, in hours and minutes?
'''



minutes = 135
print ("The movie runs for {} hour(s) and {} minutes".format(minutes//60, minutes%6
```

```
The movie runs for 2 hour(s) and 15 minutes
```

In [14]:
```
'''Exponentiation:'''

4 ** 7
```

Out[14]:
```
16384
```

The order precedence for these operators is the same as the order of operations in mathematics:

1. Parentheses
2. Exponentiation
3. Multiplication/Division
4. Addition/Subtraction

Let's demonstrate:

In [15]:
```
x = 3

x ** x ** x
```

Out[15]:
```
7625597484987
```

```
In [16]: (x ** x) ** x
```

```
Out[16]: 19683
```

```
In [17]: x ** x * x
```

```
Out[17]: 81
```

```
In [18]: x ** x * x - x
```

```
Out[18]: 78
```

Checking if a variable is of a certain type can be achieved by using the **`isinstance()`** Boolean built-in function.

```
In [19]: x = 1
         isinstance(x, int)
```

```
Out[19]: True
```

Type conversion

Python offers several built-in functions for type conversion of values: `str()` , `int()` , `float()` .

A string that contains a number can be converted to integer or float type by using the functions `int()` or `float()` respectively. The function `str(arg)` converts passed arguments into a string.

`int(arg)` converts floating-point values to integers by chopping its fractional part and can also be used to convert strings into integers.

```
In [20]: int(15.9)
```

```
Out[20]: 15
```

```
In [21]: int('1')
```

```
Out[21]: 1
```

The function `str()` converts the passed argument to a string: str(8) -> '8', str(15.8) -> '15.8'

`bool()` converts any non-zero numerical value or non-empty string to Boolean `True` and 0 or empty string to `False`

```
In [22]: str(8)
```

```
Out[22]: '8'
```

```
In [23]: str(15.8)
```

```
Out[23]: '15.8'
```

```
In [24]: bool(' ')
```

```
Out[24]: True
```

Functions

We have already seen several built-in functions: `type()`, `print()`, `str()`, `int()`, `float()`.

A **function** is a named sequence of statements that perform a computation. Functions allow us to create a block of statements and make a program smaller by eliminating repetitive code. A function is defined by specifying a function name and a sequence of statements. The result of the function is called the **return value**. Some functions yield results. Others perform an action (like printing) and do not return a value — these are referred to as **void functions**.

Once defined, a function can be called by its name — known as a **function call** — any number of times throughout the program. Expressions `type(a)` and `print(type(a))` are examples of function calls that we have seen before.

The definition of a new function must start with the keyword **`def`** followed by the function's name and a list of parameters in brackets. This first row of code, a function definition, ends with a colon `(:)`. If the function returns a value, the body of the function ends with a **`return`** statement.

The body of the function contains statements that will be executed every time the function is called. The scope of the function is specified by indenting the code. The standard indent is 4 spaces.

The general structure of a function definition looks like this:

```
def function_name(param_1, param_2, ...):
    do something with parameters
    ...
    return final_result
```

Conditionals

Conditional execution

Conditional execution allows for the execution of specific code depending on the outcome of a logical expression. The most common conditional statements are: `if` and `else`. The `if` statement will execute the associated code under the `if` branch when the condition evaluates to `True` (at the time of evaluation), otherwise the `else` branch will be executed.

Example:

In [25]:
```
score = 92

if (score > 50):
    print("You passed!")
else:
    print("You failed.")
```

```
You passed!
```

The expression after `if` should be a boolean expression, i.e. an expression that evaluates to `True` or `False`. Here is a list of the comparison operators useful for constructing boolean expressions:

```
x == y   Equality
x != y   Inequality (e.g 5 != 6 is True)
x > y    Greater than
x < y    Less than
x >= y   Greater than or equal
x <= y   Less than or equal
x is y   x is the "same" as y
```

NOTE: The `is` operator may appear to be the same as the equality (==) operator. However, the `is` operator checks if two variables **point to the same object**, whereas the `==` operator checks if the values for the two variables are the same.

Often, more than one comparison is required. Comparison expressions can be combined with logical operators: `and`, `or`, & `not`.

Examples:

- `grade > 60 and grade < 80` returns `True` if the value is in that range
- `not(grade > 60)` negates the expression in brackets, and returns `True` if the grade is **less than or equal** to 60

Operator `and` returns `True` only if both boolean expressions are True. Otherwise, `False` will be returned.

- `(True and True)` -> `True`
- `(True and False)` -> `False`
- `(False and True)` -> `False`
- `(False and False)` -> `False`

Operator `or` returns `True` if at least one of the boolean expressions is `True`.

- `(True or True)` -> `True`
- `(True or False)` -> `True`
- `(False or True)` -> `True`
- `(False or False)` -> `False`

In [26]:
```
'''Operator `or` returns True if either x or y is True'''

x = True
y = False
x or y
```

Out[26]:
```
True
```

Chained conditionals

`elif` (else if) statements may be used to create additional conditions which will be evaluated in order. The first condition in the sequence that evaluates to `True` will have its associated code executed. If none of the conditions evaluate to `True` then the `else` branch will be executed. There is no limit to the number of `elif` statements, but there can only be one `if` branch and one `else` branch.

In [1]:
```
a = 9
if (a > 10):
    print ("a is greater than 10")
elif (a < 10):
    print ("a is less than 10")
else:
    print ("a is equal to 10")
```

```
a is less than 10
```

Nested conditionals

Within a block of code associated with an `if...else` statement, you can nest additional `if...else` statements creating more complex code execution logic.

Example:

In [28]:
```
age = 31
if (age >= 30):
    if (age < 40):
        print("He is in his 30s")
    else:
        print ("He is not in his 30s")
else:
    print("He is not in his 30s")
```

```
He is in his 30s
```

Iteration

Variables may be assigned values multiple times. The value of the variable will be updated each time a new value is assigned.

In [29]:
```
x = 5
print(x)
x = 10
print(x)
```

```
5
10
```

while statement

The `while` statement allows for efficient repetition of code. It first evaluates whether a given condition is `True` :

- If `True` : Executes the associated code and jumps back to the condition to be re-evaluated.
- If `False` : Ends the loop and continues to the next statement after the loop in the program.

In [30]:
```
def counter():
    n=0
    while (n < 5):
        print("n is equal to " + str(n))
        n += 1
```

> **NOTE:** The `+=` operator was used in the last example. This operator provides a short way to update a variable. The augmented assignment statement `n += 1` is equivalent to `n = n + 1` , and is the combination, in a single statement, of a binary operation and an assignment. Other examples of augmented assignment statements are `-=` , `*=` , `/=` , `//=` , `%=` , `**=` .

In [31]:
```
counter
```

Out[31]:
```
<function __main__.counter()>
```

In [32]:
```
counter()
```

```
n is equal to 0
n is equal to 1
n is equal to 2
n is equal to 3
n is equal to 4
```

Infinite loops

Infinite loops occur when a loop is written so that the condition can never evaluate to `False` and thus **continues endlessly**.

Below is an example of such a loop. Without incrementing the value of `n` at the end of the loop, the `while` statement will always evaluate to `True`. If the program goes into an infinite loop it will continue to execute the same code indefinitely without progressing.

In [33]:
```
def counter():
    n=0
    while (n < 1):
        print("n is equal to " + str(n))
```

The **`break`** statement allows for a loop to be exited prematurely. This statement can be useful when the programmer doesn't know the number of iterations the loop must make.

In [34]:
```
def counter(n):
    while (n < 1000):
        if ((n * 11) % 7 == 0):
            print (n)
            break
        else:
            n += 1

counter(15)
```

```
21
```

The `break` statement allows us to exit a loop, and the **`continue`** statement can be used to prematurely stop the current iteration and continue with the next.

An illustration of the difference between the `break` and `continue` statements:

> Let's imagine a student reading a book on the Python programming language chapter by chapter, and each chapter is an iteration.
>
> While reading chapter 2, the student realizes they are familiar with the content of the chapter, so they jump to the next chapter (3) — this is an execution of the `continue` statement. However, if something happens that forces the student to stop reading the book altogether (e.g. realizing the book covers Python 2 instead of Python 3) this is an execution of the `break` statement.

Let's consider Newton's method for computing square roots as an example of an algorithm where the number of iterations is not known in advance. To find the square root of number `n` one can start with almost any estimate `x` and then repeatedly improve the estimate with the following formula `(x + n/x)/2`. The number of steps required to get the right answer is unknown, and a `break` statement can be used when the estimate stops changing.

In [35]:
```python
'''The implementation below assumes that
the number used as an argument is a positive number, x > 0.'''


def squareroot(x):
    est = x/2
    while True:
        print(est)
        y = (est + x/est)/2
        if y == est:
            break
        est = y

squareroot(256)
```

```
128.0
65.0
34.46923076923077
20.94807220229001
16.584383571973717
16.010295955761958
16.000003310579185
16.00000000000034
16.0
```

Recursion

A function can call another function. It can even call itself — this is referred to as a **recursive function**.

As an example of a recursive function, let's consider the factorial function:

In [2]:
```python
def factorial(n):
    if n == 0:
        return 1
    else:
        result = n * factorial(n-1)
        return result

factorial(10)
```

Out[2]:
```
3628800
```

If the argument is 0, the function returns 1 since the factorial of 0 is 1 (0! = 1).

If any other integer number is passed, that number `n` will be multiplied by `factorial(n - 1)` or n * (n-1)!. Next, `factorial(n-1)` will call itself and the calculation will be expanded to `n * (n-1) * factorial(n-2)` and so on. The function will keep calling itself until the argument is 0.

In the end, the function will result in the value of the following expression: `n * n-1 * n-2 * ... * 2 * 1`.

EXERCISE 1: Fibonacci Sequence

Write a function to compute the n-th element of the Fibonacci sequence recursively.

If you need to familiarize yourself with the Fibonacci sequence, please refer to https://en.wikipedia.org/wiki/Fibonacci_number (Fibonacci number, n.d.). The function should take an integer number `n` as an argument and return n-th element of the Fibonacci sequence. The first two numbers in the sequence are 0 and 1, and each subsequent number is the sum of the previous two.

In [4]:

```
'''Type your code here:'''



def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1)+fib(n-2)

fib(10)
```

Out[4]:

```
55
```

In [38]:

```
'''Exercise 1 Solution.

The first two numbers are 0 and 1; so that if n is 0 or 1,
the numbers in Fibonacci sequence are 0 and 1, correspondingly.
For other n we take sum of two preceding numbers in the sequence.'''


def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

End of Part 1

This notebook makes up one part of this module. Now that you have completed this part, please proceed to the next notebook in this module.

If you have any questions, please reach out to your peers using the discussion boards. If you and your peers are unable to come to a suitable conclusion, do not hesitate to reach out to your instructor on the designated discussion board.

References

Hashemi, M. (2014). https://www.paypal-engineering.com/2014/12/10/10-myths-of-enterprise-python/

McKinney, W. (2017). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython* (2nd edition). O'Reilly Media.

Fibonacci number (n.d.). https://en.wikipedia.org/wiki/Fibonacci_number.