Lab Task 1 - Python basics

CSIT375 AI for Cybersecurity
SCIT, University of Wollongong

1 Getting started

We will be using **python 3** and **jupyter** extensively in this subject. You need to set up the correct **python** environment first. Before you start this practice, you should go through the "Setting up the Environment" slides. It shows how to install Anaconda, which includes Python 3, Jupyter Notebook, and all the major libraries you will need for this subject. The aim of this lab is to introduce you to jupyter and python basics.

2 Using `jupyter`

2.1 Running Code

Jupyter Notebook is an interactive environment for writing and running code. The notebook is capable of running code in a wide range of languages. However, each notebook is associated with a single kernel. The notebook you just opened is associated with the IPython kernel, therefore runs Python code.

Code cells allow you to enter and run code

Run a code cell using `Shift-Enter` or pressing the "Run" button in the toolbar above:

[1]:
```
a = 10
```

[2]:
```
print(a) # this will output the value of a below this cell
```

```
10
```

Cell menu

The "Cell" menu has a number of menu items for running code in different ways. Try them yourself:

- Run and Select Below
- Run and Insert Below
- Run All
- Run All Above
- Run All Below

2.2 Markdown Cells

Text can be added to Jupyter Notebooks using Markdown cells. You can change the cell type to Markdown by using the Cell menu or the toolbar. Markdown is a popular markup language that

is a superset of HTML. Basic syntax can be found here: https://www.markdownguide.org/cheat-sheet/, try them out in your notebook.

2.3 LaTeX equations

You can also include mathematical expressions both inline $e^{i\pi} + 1 = 0$ and displayed:

$$e^x = \sum_{i=0}^{\infty} \frac{1}{i!} x^i \tag{1}$$

3 Exercise

This is not an assignment and you do not need to submit it

Now try out the following cells and run the python code in your notebook.

3.1 Python as a calculator

[3]:
```
2 + 2
```

[3]:
```
4
```

[4]:
```
50 - 5*6
```

[4]:
```
20
```

[5]:
```
(50 - 5*6) / 4
```

[5]:
```
5.0
```

[6]:
```
8 / 5  # division always returns a floating point number
```

[6]:
```
1.6
```

These are some common operators with example:

Operator	Operation	Example	Answer you would get
+	add	6+2	8
-	subtract	6-2	4
*	multiply	6*2	12
/	divide	6/2	3
**	power	6**2	36
e	multiply by 10 to some power	6e2	600

The priority between operators is the same as you'd expect from mathematics. From highest (i.e. first operation) to lowest (i.e. last operation):

Order	Operator	Operation
1	e	multiply by 10 to some power
2	()	evaluate term in bracket
3	**	calculate power
4	*, /, %	multiply, divide, modulo
5	+, -	add, subtract

3.2 Getting the order right

Variables must be "defined" (assigned a value) before they can be used, or an error will occur.

For example, it can't tell you what $x + 2$ is, unless you first give it a value for x.

Python works from top to bottom. That means that you need to **FIRST** give the variable a value and only afterwards try to use it.

For example, this won't work:

[7]:
```
print(b)
b = 0.1
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[7], line 1
----> 1 print(b)
      2 b = 0.1

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

But this will:

[8]:
```
b = 0.75
print(b)
```

```
0.75
```

Python also works from left to right. You always need to put the ***variable*** (the part you don't know) on the left side of the equals side and the ***value*** (the part you do know) on the right: `variable = value`

For example, this won't work:

[9]:
```
32 = c
print(c)
```

```
  Cell In[9], line 1
    32 = c
    ^
```

```
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of '='
```

But this will:

[10]:
```
d = 32
print(d)
```

```
32
```

3.3 Variable names in Python:

- can include letters, digits, and underscores
- cannot start with a digit
- are case sensitive. This means that, for example:

`weight0` is a valid variable name, whereas `0weight` is not

`weight` and `Weight` are different variables

3.4 Types of data

Python knows various types of data. Three common ones are:

- integer numbers
- floating point numbers
- strings

[11]:
```
weight_int = 10
```

[12]:
```
weight_float = 10.6
```

To create a string, we add single or double quotes around some text. For example, to identify and track a cat, we can assign each cat a unique identifier by storing it in a string:

[13]:
```
cat_id = '001'
```

3.5 Using Variables in Python

Once we have data stored with variable names, we can make use of it in calculations. We may want to store the cat's weight in pounds as well as kilograms:

[14]:
```
weight_lb = 2.2 * weight_int
```

We might decide to add a prefix to the cat identifier:

[15]:
```
cat_id = 'cat_' + cat_id
```

3.6 Built-in Python functions

To carry out common tasks with data and variables in Python, the language provides us with several built-in functions. For example, to display information to the screen, we use the `print` function:

[16]:
```
print(weight_lb)
print(cat_id)
```

```
22.0
cat_001
```

We can also call a function inside of another function call. For example, Python has a built-in function called `type` that tells you a value's data type:

[17]:
```
print(type(10.6))
print(type(cat_id))
```

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

Moreover, we can do arithmetic with variables right inside the `print` function:

[18]:
```
print('weight in pounds:', 2.2 * weight_int)
```

```
weight in pounds: 22.0
```

3.7 Control Flow Tools

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the Fibonacci series as follows:

3.7.1 `while` statement

[19]:
```
# Fibonacci series:
# the sum of two elements defines the next
a, b = 0, 1
while a < 10:
    print(a)
    a, b = b, a+b
```

```
0
1
1
2
3
5
8
```

3.7.2 `if` Statement

There can be zero or more elif parts, and the else part is optional. The keyword 'elif' is short for 'else if', and is useful to avoid excessive indentation. An if … elif … elif … sequence is a substitute for the switch or case statements found in other languages.

[20]:
```
x = int(input("Please enter an integer: "))

if x < 0:
    x = 0
    print('Negative changed to zero')
elif x == 0:
    print('Zero')
elif x == 1:
    print('Single')
else:
    print('More')
```

```
Please enter an integer: 6
More
```

3.7.3 `for` Statements

Python's for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

[21]:
```
# Measure some strings:
words = ['AI', 'for', 'Cybersecurity']
for w in words:
    print(w, len(w))
```

```
AI 2
for 3
Cybersecurity 13
```

3.7.4 The `range()` Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

[22]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

The given end point is never part of the generated sequence; range(10) generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another

number, or to specify a different increment (even negative; sometimes this is called the 'step'):

[23]:
```
list(range(5, 10))
```

[23]:
```
[5, 6, 7, 8, 9]
```

[24]:
```
list(range(0, 10, 3))
```

[24]:
```
[0, 3, 6, 9]
```

[25]:
```
list(range(-10, -100, -30))
```

[25]:
```
[-10, -40, -70]
```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

[26]:
```
a = ['Mary', 'had', 'a', 'little', 'cat']
for i in range(len(a)):
    print(i, a[i])
```

```
0 Mary
1 had
2 a
3 little
4 cat
```

3.8 Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

[27]:
```
def fib(n):    # write Fibonacci series up to n
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

# Now call the function we just defined:
fib(2000)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

4 References

- Jupyter notebook documentation: https://jupyter-notebook.readthedocs.io/en/stable/
- The Python Tutorial: https://docs.python.org/3/tutorial/index.html