NPA Data Science:

Computer Programming 2

michaelferrie@edinburghcollege.ac.uk

Part 1: Operators in Python

Python defines several types of operators, operators can be used to evaluate operands in a procedure, consider the following procedure 4+5 in this procedure the operator is the additional symbol and 4 and 5 are the operands which are used as two arguments within the body of the procedure. The procedure should result in 9, given that 4+5=9. Here are some of the other common arithmetic operators available in python:

Operator	Name	Example	Result
+	Addition	4 + 3	7
-	Subtraction	4 - 3	1
*	Multiplication	4 * 3	12
/	Division (Float)	4 / 3	1.33333333333
//	Floor Division	4 // 3	1
%	Modulus	4 % 3	1
**	Exponential	4 ** 3	64

The standard arithmetic operators should be self-explanatory however division is a little more complicated / gives normal division // gives division but removes any value after the floating point. Modulus % gives the remainder after a division so 6 % 4 returns 2 because if you divide 6 by 4 you get 1 and a remainder of 2, the remainder is what the modulus returns.

Python also defines several comparison operators, where you can compare one operand to another - some (not all) are detailed here:

Operator	Name	Example	Result
`==`	Equal	`3 == 4`	False
`!=`	No Equal	`3 != 4`	True
`>`	Greater than	`3 > 4`	False
`<`	Less than	`3 < 4`	True
`>=`	Greater than or equal to	`3 > 4`	False
`<=`	Less than or equal to	`3 < 4`	True

There are some other operators that will be useful to us while studying this course, this table contains the logical, identity and membership operators - these are included here:

Operator	Name	Example	Result
`and`	Returns True if both statements are true	`3 > 4 and 5 > 6`	False
`not`	Reverse the result, returns False if the result is true	`3 > 4 and 5 > 6`	True
`in`	Returns True if a sequence with the specified value is present in the object	`‘h’ in ‘hello’`	True

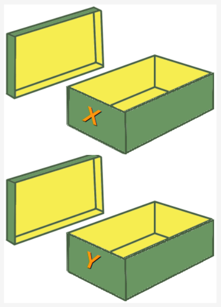
Part 2: Variables and Data Types

Variables

Variables are used to store information to be referenced and manipulated in a computer program. They also provide a way of labeling data with a descriptive name, so our programs can be understood more clearly by the reader and ourselves. It is helpful to think of variables as containers that hold information. Their sole purpose is to label and store data in memory. This data can then be used throughout your program.

When you are naming variables, think hard about the names. Try your best to make sure that the name you assign your variable is accurately descriptive and understandable to another reader. Sometimes that other reader is yourself when you revisit a program that you wrote months or even years earlier. Think of the variable as a box in the computer memory that you can store things in and label.

In python a variable is created using the assignment operator = the name of the variable is specified followed by the value you want to set. Because the equals sign (=) is reserved for variable assignment, to say equals use ==.



```
my_variable = 3

my_second_variable = 4

my_third_variable = “hello”
```

Data Types

In programming data types are an important concept. All variables have a data type, python is a dynamically typed language - which means you can change the type of a variable. Variables can store data of different types, and different types can do different things. Python has the following data types built-in by default, in these categories:

Strings - these are strings of characters, single characters or even full sentences, you set a variable to be a string by enclosing it in quotes the string is abbreviated to str in python.

```
x = "Hello World"   str
```

Numbers are called integers and python abbreviates this to int, to create an integer just type the variable name then a number

```
x = 20              int
```

Another numeric type is a float. This is a number with a decimal point, Americans call the decimal point a floating point so this is where that name comes from.

```
x = 20.5            float
```

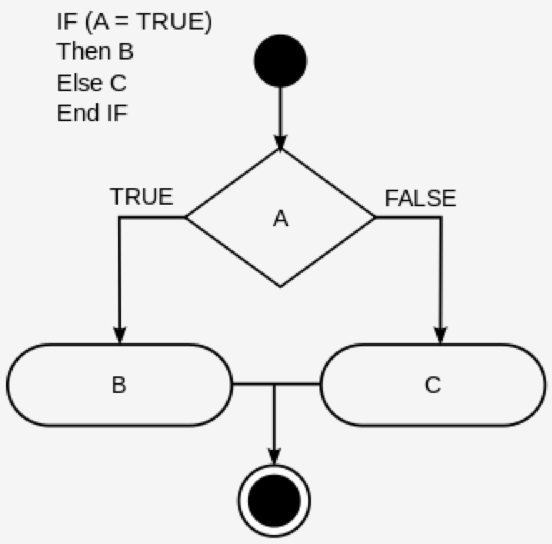
A final numeric type is a boolean, this is a type that is either true or false python writes these as True or False, always with a capital T/F.

```
x = True            bool
```

Part 3: Conditionals and Loops in Python

Conditionals

A conditional is sometimes called a test or an if statement and is made of three parts, the predicate, the consequent and the alternative. Conditionals are very common, we use these to get the computer to stop what it is doing and check a condition. This is sometimes called if then else. Consider the diagram that shows the basic structure of a conditional:



In the diagram we can see the three parts of the conditional, the **predicate** in A, the **consequent** in B and the **alternative** in C.

Let's create an example conditional to illustrate this, let us say that we want to check if a number is greater than 5 - that is the **predicate**. If the number is greater than or equal to

5, then I would like the computer to print, “Yes it’s higher than or equal to 5,” that is the **consequent**. Finally, if the number does not match the condition of the predicate and is not greater than 5 then we use the **alternative**. If the number is lower than 5 I would like the computer to print, “This number is lower than 5.” To create a conditional in python the syntax is as follows:

```
my_number = 10


if my_number >= 5:
  print("Yes it’s higher than or equal to 5")
else a <= b:
  print("No this is lower than 5")
```

It is worth noting that python does not require an else alternative, we could just write this as follows, and python would not give any errors:

```
if my_number >= 5:
  print("Yes it’s higher than or equal to 5")
```

In this example we have a variable which is an integer with the value of 10, then we create a conditional with the predicate of the number being greater than 10, if this is true we print the consequent, otherwise we print the alternative.

Loops

Python defines **only two** types of loops - for loops and while loops a loop is useful when you have a number of values to check, the computer can follow a sequence using a loop. For loops are used when you have a given sequence that you would like to iterate over, consider the following example which will print each letter in the word apple.

```
for x in "apple":

  print(x)
```

For loops can be combined with the range function to provide a range of values to iterate over:

```
for x in range(5):

  print(x)
```

The second type of loop in python is a while loop. With the while loop we can execute a set of statements as long as a condition is true. In this example we have a variable called my_number which is set to 10. A while loop has a condition that if the variable my_number is less than 10 print out the value of the variable then add one to the variable. This will continue adding 1 to the variable until the variable is 20. It may also be worth noting that the while loop itself contains a conditional.

```
my_number = 10

while my_number < 20:

    print (my_number)
    my_number += 1
```

Part 4: Lists in Python

Introduction to Lists

As well as simple data types like strings, integers and floats, python also offers some more abstract data types, the first of which is a list. Lists are one of the most frequently used and very versatile data types used in Python. A list is created by placing all the items (elements) inside square brackets [], separated by commas. It can have any number of items and they may be of different types (integer, float, string etc.).

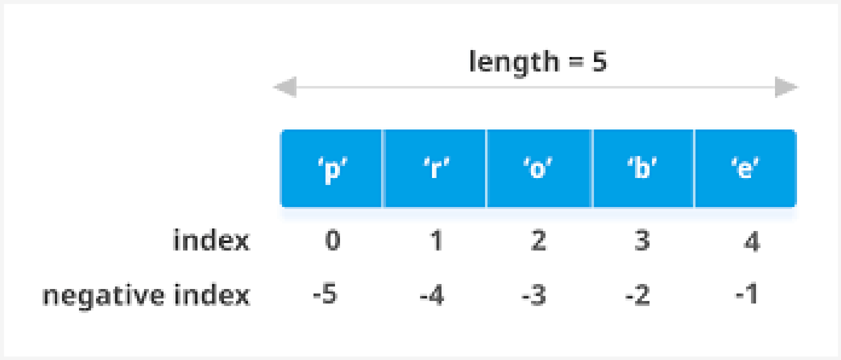
Consider the following examples:

```
# empty list

my_list = []

# list of integers

my_list = [1, 2, 3]

# list with mixed data types
```

```
my_list = [1, "Hello", 3.4]
```

List Indexing

We can use the index operator [] to access an item in a list. In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4.



Trying to access indexes other than these will raise an IndexError. The index must be an integer. We can't use float or other types, this will result in TypeError. Lists also have a negative index, so that you can access data counting from the last item in the list.

```
# List indexing

my_list = [1, 2, 3]

print(my_list[0])

# Output: 1

print(my_list[2])

# Output: 3

print(my_list[-1])

# Output: 3
```

Slicing Lists

We can access a range of items in a list by using the slicing operator :(colon).

```
# elements 3rd to 5th

print(my_list[2:5])
```

```
# elements beginning to 4th

print(my_list[:-5])

# elements 6th to end

print(my_list[5:])
```

Adding and removing from a list

Lists are **mutable (changeable)**, meaning their elements can be changed. We can use the assignment operator = to change an item or a range of items.

```
my_list = [2, 4, 6, 8]

# change the 1st item

my_list[0] = 1

print(my_list)

# change 2nd to 4th items

my__even_list[1:4] = [3, 5, 7]

print(my_list)

# delete one item

del my_list[2]

# delete multiple items

del my_list[1:3]

print(my_list)

# delete entire list

del my_list
```

Useful list methods

Methods that are available with list objects in Python, they are accessed by my_list.method(). Consider the following table of list methods.

List Method	**Description**

`append()`	Add an element to the end of the list
`insert()`	Insert an item at the defined index
`extend()`	Add all elements of a list to the another list
`remove()`	Removes an item from the list
`pop()`	Removes and returns an element at the given index
`copy()`	Returns a shallow copy of the list
`sort()`	Sort items in a list in ascending order
`clear()`	Removes all items from the list
`reverse()`	Reverse the order of items in the list
`count()`	Returns the count of the number of items passed as an argument
`index()`	Returns the index of the first matched item

Test out the following examples, notice how the method can be added to the list using a.

```
my_list = [3, 2, 3]

print (my_list)

print (my_list.count(2))

my_list.sort()

print (my_list)

my_list.append(4)

print (my_list)
```

Looping Over Lists

Another very common technique, arguably one of the most important skills in python is to be able to loop over the items in a list. In previous examples we have done this with a string and the examples are quite similar. Consider the following examples on my_list, again we use a temporary variable by convention. I use **item** for illustration purposes.

```
# Create and print a list

my_list = [3, 2, 3]

print (my_list)


# Iterate over the list and print the values one at a time as
they are placed into the item variable.

for item in my_list:

     print(item)


# Perform the same iteration but this time only if the item meets
a condition

for item in my_list:

if item == 3:

     print(item)


# A more illustrative example of using a while loop and an
accumulator variable to control the iteration count of the list:

while count <3:

     print (my_list)

     count += 1
```