Python Basics

This lecture covers the core of the Python language, introducing the main concepts of the language and providing a comprehensive set of tools that can be used to solve several fundamental problems.

1 Interacting with Python

There are two ways to execute Python scripts. First, you write a `.py` file and run it in a terminal with `python3 myfile.py`. Second, you execute the Python commands line by line in the Python REPL (read-eval-print-loop). To open the Python REPL simply type `python3` in the terminal. You can quit Python by running `exit()` or with Control-D. In the Jupyter notebook you are interacting with a REPL, each cell you execute sends the commands to the interpreter, which then executes and prints outputs and then waits for more commands.

2 Built-In Data Types

The main data types that are built-in Python are:

- Numerics: hold different types of numbers (integers, floats and complex numbers);
- Text: hold characters and strings (`string`);
- Boolean: used to test conditions (`bool`, `True`, `False`);
- Sequences: list of elements where order matters (`list`, `tuple` and `range`);
- Sets: list of distinct objects where order is not important (`set`, `frozenset`);
- Mapping: maps keys to values (dictionary, `dict`);
- Objects: represents all data in Python (class)
- Exceptions: represents errors in Python

All of the built-in types are fully described here. We will cover most of them. There are also many built-in functions, which are described here, and we will cover some of them as we go.

2.1 Numerical Data

Numbers are separated into three different types: `int`, `float` and `complex`. Integers can be as big as required, as long as you have free memory available.

Variables in python are not strongly typed. This means the type of the variable is inferred at the time of declaration from its context. Thus, when creating numerical data, the type of variable will be inferred from its definition.

```
# integer
x = 3005
# big integer
y = 10000000000000000000000000000000000000000000000
z = 10**30000
```

We can print these numbers on the screen by typing the name of the variable, like `x`, or by using the function `print`.

```
# print the value of the variables on the screen
x
y
z
# alternatively use print
print(x)
# can print multiple variables
print(x, y, z)
```

We can check the size of these integers (in bytes) with a helper function from the `sys` module (more on modules later), named `getsizeof`. We use functions inside a module using the dot notation: `module_name.name_of_function`. For example, to use `getsizeof` we call `sys.getsizeof`.

```
# import module sys
import sys
# print size of the integer stored in x
print(sys.getsizeof(x))
# print size of the large integers
print(sys.getsizeof(y), sys.getsizeof(z))
```

The number of bytes assigned to the variable changes as the integer increases.

Floating point numbers have a different implementation that varies by operating system, but usually is equivalent to a double type in C (8 bytes). To create a float:

```
# floating point numbers: floats
a = 3.1415
b = 2.718281828
```

To create complex numbers we use `j`:

```
# complex numbers
c = 2+3j
d = 1j
print(c, d)
```

Complex numbers have properties that can be accessed using the dot notation, just like we accessed the function `getsizeof` from the module `sys`.

```
# real part of a complex number
print(c.real)
# imaginary part of a complexn umber
print(c.imag)
```

We will not talk anymore about complex numbers and choose to focus only on integers and floats.

The integers and floats support the usual mathematical operations.

```
x = 11
y = 2
# Addition, subtraction, multiplication and division
print(x + y, x - y, x*y, x/y)
# Integer division
print(x//y)
# Remainder of integer division
print(x%y)
# Absolute value
print(abs(-x-y))
# Exponentiation
print(pow(x, 2))
print(x**2)                    # equivalent to pow
```

Notice the use of two built-ins: `abs` for absolute value and `pow` for exponentiation.

Some of the operations will make implicit conversions from integers to floats or the other way around. We can check the type of a variable with the built-in function `type`.

```
# x and y are integers
print(x, type(x))
print(y, type(y))
# when we divide x and y, they are implicitely converted to floats
# so that the result is also a float
print(x/y, type(x/y))
```

We can convert between integers and floats by calling the built-in functions `int` and `float`:

```
z = 3.2
print(int(z), type(int(z)))
h = 22
print(float(h), type(float(h)))
```

These functions are useful to convert strings into numbers.

2.2 Text Data

Textual data in Python is stored in `str` objects. There are several ways of creating strings:

```
# Create strings with single quotes '
first_name = 'Guilherme'
# or double quotes "
last_name = "Salome"
print(first_name, type(first_name), sys.getsizeof(first_name))
# A string can also have quotation marks
```

```
more_strings = 'This is a string with "quotation marks"'
print(more_strings)
# To create strings with multiple lines use triple quotes '''
# or triple double quotes """
multiple_lines = '''This is a string
that spans
multiple lines. They can also be created with
triple quotation marks """ """'''
print(multiple_lines)
```

Strings have an implicit join when they are part of a single expression:

```
name = "Guilherme " "Salome"
complete_name = ("Guilherme " "Salome")
```

Strings are a collection of characters. Think of a vector, where each element is a letter. We can access letters in a string using the `[]` notation.

```
# The first letter of a string has the index 0
# we can access it with [0]
name = "Guilherme"
print(name[0])
print(name[1], name[2])
# Last letter
print(name[8])
# Last letter of a string can be recovered with the index -1
print(name[-1])
```

We can use the built-in function `len` to get number of letters in a string.

```
print(len(name))
# Since index starts at 0, the last letter is at position len(name)-1
print(name[len(name) - 1])
```

Strings are immutable:

```
name = "Guilherme"
print(name[0])                    # first letter of string
name[0] = 'A'                     # error
```

We can concatenate strings with the `+` operator.

```
fname = "John"
lname = "Doe"
print(fname + lname)
print(fname + ' ' + lname)
```

Whatever is being concatenated has to have the type `str`, otherwise it needs to be converted to a string by using the built-in function `str`:

```
x = 1
name = "Guilherme"
print('Student ' + x + ': ' + name)  # error since x is not a string
# convert to string with str
print('Student ' + str(x) + ': ' + name)  # error since x is not a string
```

Strings are objects. Objects hold values, like the value of the string, but can also hold other information, like properties and methods. We have seen an object with properties

when we created a complex number. In that case, the complex number object had two properties, one that held the real part of the number, and another that held the complex part of the number. Objects can also have methods. Methods are functions which relate to the object itself. For example, a method might use the objects value and create a new one. In the case of strings, there is a method that takes the string in the variable and returns a new string with all letter capitalized.

```
first_name = 'guilherme'
# Capitalize string
print(first_name.capitalize())
print(first_name)
capitalized = first_name.capitalize()
# All upper case
upper_case = first_name.upper()
print(upper_case)
# All lower case
print(upper_case.lower())
```

Remember that strings are immutable, so string methods all return new strings when used. We will use other methods after we talk about sequences and booleans. A complete list of the string methods is available here.

2.3 Boolean

There are two boolean values: `True` and `False`. The booleans are a subclass of integers, and, in some contexts, the integers 1 and 0 represent the booleans `True` and `False`.

We can obtain booleans via comparisons. There are 8 types of comparisons in Python:

```
print(1 < 1)
print(1 <= 1)
print(1 > 1)
print(1 >= 1)
print(1 == 1)
print(1 != 1)
print(1 is 1)  # compares if two object are the same
print(1 is not 1)
```

Booleans support the operations: `and`, `or` and `not`.

```
print((1 > 0) and (2 > 1))
print((1 > 2) or (1 < 2))
print((1 > 2) or (0 > 1))
print(not True)
```

The built-in function `bool` can convert any value to a boolean.

```
print(bool(0), bool(1))
print(bool('Guilherme'))
print(bool([]))
```

When we use the function `bool` Python evaluates whether the input of the function is associated with False or True. The most common objects that will lead to a `False` value are:

- Constants that are False by definition: `False` and `None`

- Numbers that are zero: 0, 0.0, 0j
- Empty sequences: `''`, `()`, `[]`, `{}`

Other objects that are non-empty will lead to a value of `True`.

2.4 Sequences

Sequences define a sequence of objects where the order of the objects is kept. There are three basic types: `list`, `tuple`, `range`. A `list` can hold any number of elements and types of objects:

```
list_of_numbers = [1, 2, 3, 2.3, 3.1, -1, 0, -5]
print(list_of_numbers)
list_of_strings = ['Guilherme', 'Salome', 'HFFE', 'Lecture 13']
mixed_list = [-1, False, 'Hello', 3.2]
```

Elements of the list can be accessed by index. The index of a list starts at `0`.

```
# first element at index 0
print(list_of_numbers[0])
print(list_of_numbers[1])
```

The built-in function `len` gives the length of a list:

```
print(len(list_of_numbers))
print(len([]))
print(len('HFFE'))
```

Notice that a string is backed up by a list, so it shares some of its methods and returns the number of characters when `len` is called on a string.

Because indices start at 0 and not at 1, the last index of a list is given by the length of such list minus 1. The last element of a list can also be accessed via the index `-1` (equivalent of Matlab's `end`):

```
print(mixed_list[3])
print(len(mixed_list))
print(mixed_list[len(mixed_list)])  # index out of range
print(mixed_list[len(mixed_list)-1])
print(mixed_list[-1])
```

We can recover several elements of a list, also known as slicing, using the `[]` notation.

```
numbers = [-1, 2, 5, 7, 9, 10, 12]
# recover first element
numbers[0]
# recover first three numbers
numbers[0:3]
# equivalent to
numbers[:3]
# recover from the third number to the fifth
numbers[2:5]
# recover from the 2nd number to the last
numbers[1:len(numbers)]
numbers[1:]
# recover the last number
```

```
numbers[-1]
# recover every other number
numbers[0:len(numbers):2]        # start:stop:step
numbers[0::2]
# recover every 3rd number
numbers[0::3]
```

Elements of a list can be modified:

```
list_of_numbers[0] = 1000
print(list_of_numbers)
list_of_numbers[-1] = 'oops'
print(list_of_numbers)
```

A list can be extended:

```
print(list_of_numbers)
list_of_numbers.append(100)     # adds to the end of list
print(list_of_numbers)
list_of_numbers.insert(0, 100)  # inserts at beginning of list
print(list_of_numbers)
list_of_numbers.insert(1, 100)  # inserts at position 1, shifts everythin t
```

It is important to note that adding elements to the beginning of a list or at a random location inside the list is slow. Appending at the end of the list is much faster.

A list can be shrunk.

```
print(list_of_numbers)
list_of_numbers.pop()           # removes last item
print(list_of_numbers)
list_of_numbers.pop(0)          # removes first item (this is slow)
print(list_of_numbers)
del list_of_numbers[1]          # removes element at index 1
print(list_of_numbers)
```

A `tuple` is also a sequence, but it is a sequence that cannot be modified after it is created.

```
tuple_of_numbers = (1,2,3,4,5)
print(tuple_of_numbers)
print(tuple_of_numbers[0], tuple_of_numbers[-1], len(tuple_of_numbers))
tuple_of_numbers[0] = 100       # raises error
tuple_of_numbers.append(100)    # raises error
tuple_of_numbers.pop()          # raises error
del tuple_of_numbers[0]         # raises error
```

Tuples are more efficient that lists since they occupy less bytes. Tuples are used when you want to fix a sequence of values that should not be changed.

A `range` constructs a sequence of numbers and is often used for looping a specific number of times in a loop.

```
print(list(range(0, 10)))
print(list(range(10)))
print(list(range(0, 10, 2)))
```

It is important to know that all functions that return a list of numbers in Python exclude the last number from the list. For example `range(5)` returns a list with the numbers

`0,1,2,3,4`. This is because lists start in the index 0, so the last index of a list with 5 elements is the index 4. This contrasts with lists in Matlab which start at index 1, but is a small change that is more natural to programming and is the default in most of the programming languages.

2.5 Sets

A `set` is a list of distinct objects without a specific order (unordered collection). To create a set we pass a list of elements to the `set` function:

```
students = set(['A', 'B', 'C', 'D'])
print(students)
{'A', 'B', 'C', 'D'}            # also creates the same set
set()                           # empty set
{}                              # NOT an empty set, but an empty dictionary
print(type(set()), type({}))
```

Notice that the elements in a set are all unique. Indeed, `set` is an implementation of a mathematical set:

```
print({'A', 'A', 'A', 'B', 'B', 'C'})
```

A `set` can be extended and its elements can be removed:

```
print(students)
students.add('E')
print(students)                 # notice the order of elements might change
students.discard('A')           # removes 'A' if it is in the set
students.discard('A')           # if not in the set, do nothing
students.remove('B')            # removes 'B' it it is in the set
students.remove('B')            # if not in the set, raise error
```

We can find how many elements are in a set with `len`.

```
print(len(students))
```

As the implementation of a mathematical set, we can take unions, intersections and differences. We can also check if an element is in a set.

```
students = {'A', 'B', 'C', 'D', 'E'}
teacher = {'G'}
assistant = {'I'}
everyone = students.union(teacher, assistant)  # new set that is union of a
everyone_but_A = students.difference({'A'})
print({'A', 'B', 'C'}.intersection({'B', 'C'}))
print('A' in students)
```

Other operations are described here.

A `frozenset` is similar to a `set` but is an immutable collection.

```
students = frozenset(['A', 'B', 'C', 'D', 'E'])
# can do everything a set does, but cannot be modified
students.add('F')               # error
```

A `frozenset` is useful when you want to name a set of values that will have some use in your program. This makes your code more readable and easier to identify key parameters.

2.6 Mapping (Dictionaries)

A mapping is an object that associates names (keys) to values (any object). The standard implementation of a mapping in Python is a dictionary: `dict`. A dictionary can be created by giving the `dict` constructor a list of `key:value` pairs:

```
grades = {'A': 10, 'B': 8, 'C': 7, 'D': 8.5, 'E': 9.8}
print(type(grades))
print(len(grades))
grades['A']                    # returns value for grade of A
grades['C'] = 7.5
print(grades)
```

Dictionaries can be modified, its elements can be removed, new elements can be added, and existing elements can be updated.

```
print(grades)
grades.update({'A': 0})        # A was cheating
print(grades)
del grades['B']                # remove grade for B
print(grades)
grades['F'] = 10               # adds a new pair to the dictionary
print(grades)
```

We can recover (as a list) all of the keys of a dictionary, all of its values, and all of the `key:value` pairs:

```
list(grades.keys())            # a list of all dict keys
list(grades.values())          # a list of all dict values
list(grades.items())           # a list of the key:value pairs
```

Other methods are described here.

2.7 Objects

Objects are the core of everything in Python. In fact, every data in Python is represented by an object. All of the basic types we studied so far are types. That is why when we call `type(1)`, for example, we get back `<class 'int'>`. This means the number 1 is an object of the class int.

Every object in Python has: an identity, a type and a value. This means every single variable we declare in our scripts will have an identity, a type and a value, since everything in Python is an object, no exceptions.

The identity of an object is a value that never changes after it was created, it is like the address of an object in the computer's memory. The comparison operator `is` compares the identity of two objects.

```
a = []
b = []
print(a == b)                          # the lists have the same values
print(a is b)                          # but they are not the same objects
a.append(1)
b.append(1)
print(a == b)  # both lists have the same values, a 1
# but the 1 that lives in a is not the same that lives in b
# I can change the value of b[0] and it does not affect a[0]
```

```
print(a is b)
```

We can get the identity of an object with `id` function (a function is also an object in Python):

```
print("Identity of 'a': " + str(id(a)))  # id returns an integer representi
print("Identity of 'b': " + str(id(b)))  # id returns an integer representi
```

The type of an object determines what functions it supports, it represents the object's definition. The type of an object cannot be changed. As we have seen before, we can discover the type of an object with the `type` function.

```
  print(type(a))
  print(type(type))                    # type is an object, and has a type
```

The value of an object may be mutable or immutable, like lists and tuples.

The built-in functions `dir` and `help` can be used to inspect the methods of an object, and to get help on those methods:

```
a = [1, 2, 3]
print(type(a))                    # a is a list
print(dir(a))                     # these are all of its methods
# we can understand each method by using the help function
help(a.reverse)
help(a.sort)
```

For now we will skip the topic of how to define new objects, but we will talk about it after we cover loops, conditionals and functions.

2.8 Exceptions

Exceptions are objects that hold the information about errors that occur in Python during the run-time of a script. There are many built-in exceptions that are raised depending on the situation. For example, trying to access an element of a list with an index that is not correct will raise an `IndexError`:

```
students = ['A', 'B', 'C']
students[4]
```

For example, trying to access a value stored in a dictionary with a non-existing key will raise a `KeyError`:

```
grades = {'A': 10, 'B': 8}
grades['C']
```

For example, calling a method on an object that does not actually have that method will raise an `AttributeError`:

```
students = {'A', 'B', 'C', 'D', 'E'}
students.append('F')              # there is no append method in a set
```

A complete list of the exceptions that are built-in Python is available here.

The philosophy of programming in Python when it comes to dealing with errors is known as `EAFP`: Easier to Ask for Forgiveness than Permission. It means that when we are coding, we should assume that the values/methods exist and try to do the computations, but if something fails, then we will fix it. This is opposed to the style known as LBYL: Look Before You Leap. Where you check if everything is as should be before doing the

computations. This distinction is just a philosophical one, and, in the case of Python, the EAFP increases the speed of developing software.

In a LYBL style of coding we would write software as such:

```
if is_everything_right_for_computing():
    # if everything is right (variables have the values we expect, ...)
    do_the_computations()
else:
    handle_the_error_case()
```

In a EAFP style of coding we would write the same software as:

```
try:
    do_the_computations()
except SomeErrorHappened:
    handle_the_error_case()
```

The EAFP approach is what we will use when handling errors.

Handling errors is an extremely important topic, but requires understanding conditionals, so we will defer studying exceptions to a later section.

3 Control Flow Tools

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.

3.1 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 `eslse` is executed.

For example:

```
value = 100
if value < 100:
    print('Value is smaller than 100')
else:
    print('Value is at least 100')
```

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

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

We can test multiple conditions:

```
name = 'Guilherme'
if name.startswith('A'):
    print('Starts with letter A')
elif name.startswith('B'):
    print('Starts with letter B')
else:
    print('Starts with some other letter')
```

If the first two conditions are not executed then the last condition (the default) is. Notice the use of the `str` method `startswith` to check whether the first letter of the string `name` matches some other letter.

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
```

3.2 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 iterator:
    do something here
```

A for loop iterates the variable `i` over the values in `iterator`. 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.

```
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 the container is not ordered, like in a `set`, then the loop would process each elements in an arbitrary order, but all of the elements would be processed.

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:

```
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 iterate over keys and values of a dictionary. To do so, we use the `items` method of dictionaries:

```
studentsGrades = {'A': 10, 'B': 9, 'C': 9.5, 'D': 8}
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`.

3.3 Break and Else Statements in For Loops

The `break` keyword can be used inside a loop (either for or while) to break out of it. Specifically, the `break` keyword breaks out of the most inner loop.

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Another example:

```
j = 0
while True:
    for i in range(1000):
        print(i, j)
        if i == 5:
            break
    j += 1
    if j == 2:
        break
```

Loops also have an `else` clause that is executed after the loop finishes (after the iterator is exhausted) and only if the loop finishes without the use of a `break`.

```
checkIfPrime = range(3, 10)      # what number is prime from 3 to 9
for number in checkIfPrime:
    for divisor in range(2, number):
        if number % divisor == 0:  # number can be written as a*b with a,b
            print(f'{number} is not prime')
            print(f'{number}={divisor}*{number//divisor}')
            break
    else:
        # no break occurred, this means no divisor was found
        print(f'{number} is prime!')
```

Notice that the `else` clause does not belong to the `if`, but actually to the `for`, the indentations are different.

3.4 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 writeFibonacciSeries(n):
    """
    Writes a list of the first n numbers from the Fibonacci series.

```

```
    Input:
     n: integer, number of elements to obtain from the Fibonacci series
        n >= 2

    Output:
     series: list of integers
    """
    series = [1, 1]             # first 2 numbers of the series
    total = 0
    while total < n - 2:
        series.append(sum(series[-2:]))
        total += 1
    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 `writeFobinacciSeries` is available for use. We can now call this function:

```
writeFibonacciSeries(10)
```

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

```
help(writeFibonacciSeries)
```

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)                  # 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
```

3.4.1 Functions are Objects, like everything else

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 memor
a = sayHello
print(a)                        # same address
# the variable a points to the same function
id(a) == id(sayHello)
```

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. We can in fact delete the name `sayHello` and the function will still be accessible via the name `a`:

```
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')()
```

3.4.2 Default Values

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

```
def installVirus(prompt, retries=4, reminder='Please try again!'):
    while True:
        response = input(prompt)  # asks user for input
        if response in ('y', 'Y', 'yes', 'Yes'):
            return True
```

```
        if response in ('n', 'N', 'no', 'No'):
            return False
        retries = retries - 1   # retries starts at 4 by default
        if retries <= 0:
            break
    print(reminder)
```

Notice the use of the membership operator `in` to check whether the `response` variable is a member of a set.

This function can be called in several different ways:

```
installVirus('Do you want to Install this software?')
installVirus('Install?', 1)
installVirus('Delete your computer?', 10, 'Try again!')
```

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

```
installVirus('Delete file?', reminder='Did not work, try again.')
```

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
```

The inputs of a function are positional: the order they are defined in the function definition is the order in which they should be received. However, is possible to change the order by specifying the name of the input. In this case, we pass `name=value` pair, also known as keyword arguments. For example:

```
def foo(x, y=0, z=0):
    print(x, y, z)

# x is a positional input, it has no default value so must always be suppli
foo()                           # error
# missing a positional argument
foo(1)
# y and z are keyword arguments
# we can pass them as positional arguments
foo(1, 10, 20)
# or by specifying name=value pairs
foo(1, y=10, z=20)
# when specifying name=value pairs the order can be changed
foo(1, z=20, y=10)
# but positional arguments must always come first
foo(z=20, y=10, 1)              # error
```

3.4.3 Functions with *args and **kwargs

You will often find functions where some of its inputs are called `*args` and `**kwards`:

```
def myFunction(variable1, *args, **kwargs):
      pass
```

Here the `variable1` is just the first argument that the function receives, it is named 'variable1' and is available for use within the function body. The keywords `*args` and `**kwargs` allow the function to accept optional arguments, as many as you want. Thus, the function above requires at least one argument, called "variable1", and can accept extra arguments.

The keyword `*args` collects additional positional arguments, that is, arguments that do not have a keyword associated with them. These arguments are collected into a tuple with the name 'args'.

```
def f(a, *args):
    print(a)
    print(args)
f(1, 2, 3, 4, 5, 6, range(10))
```

The keyword `**kwargs` collects values that are associated with names. These names and values are collected into a dictionary, which is available in the function body via the name 'kwargs':

```
def f(a, *args, **kwargs):
      print(a)
      print(args)
      print(kwargs)
f(1, 1, 2, 3, range(10), name='Guilherme', lastname='Salome', age='29')
```

If no extra arguments are passed, then "args" and "kwargs" will be empty:

```
f(1)
```

3.4.4 Anonymous Functions: lambdas

We can declare short inline functions using the `lambda` keyword.

```
add = lambda x, y: x + y
```

After the keyword `lambda` comes the arguments, in this case x and y. After the colon comes whatever the function will return, in this case `x+y`. The function here is assigned to the name 'add'.

Lambdas are useful in some contexts. Suppose we have a list that we want to sort:

```
grades = [('A', 9), ('B', 10), ('C', 7), ('D', 9.5)]
dir(grades)
help(grades.sort)
```

The list method `sort` sorts a list in place. It takes an optional argument called `key`. This argument is a function that takes an element of the list and returns a value. The value that is returned is used for sorting the list.

```
print(grades)
grades.sort()
print(grades)
```

Notice that the tuples inside this list were sorted by their first value, a string. We can sort the list of students and grades based on the grades. We can passa a lambda function to the `key` argument that will return the grade value for sorting:

```
print(grades)
grades.sort(key=lambda x: x[1])
print(grades)
```

If we want to sort in a decreasing order:

```
print(grades)
grades.sort(key=lambda x: x[1], reverse=True)
print(grades)
```

4 Handling Exceptions

It is possible to write a Python script that can handle exceptions as they occur during execution. In fact, many of Python's own functions use exceptions as a form of control flow. Let's see how we can handle errors:

```
while True:
    try:
        number = int(input("Please enter an integer: "))
        break
    except ValueError:
        print("Invalid input. Not a number, please try again.")
print(f'Your number converted to integer = {number}')
```

The `input` will wait for the user to type something in the standard input and hit enter. Whatever was inputed will come in as a string, and the `int` function will attempt to convert it to an integer. If the string does not contain an integer, then a `ValueError` exception will be raised.

```
int(2.35)                       # no error, converting float to int works
int('2')                        # works, there is an integer in the string
int('2.35')                     # error, no integer in the string
```

Python will execute whatever is in the `try` block. If no exception occurs inside the block, then the `except` statement is skipped and the `try` statement is finished. However, if an exception occur inside the `try` block, then the rest of the `try` block is skipped. Then, if the exception type matches with the keyword that comes after `except`, the `except` block is executed. After the `except` block is executed, the code continues on.

If another exception occur with a type different than `ValueError`, then it is an unhandled exception, and the execution of the script will be stopped and an error message will be shown. Try executing the `while` loop above and instead of entering an integer, hit Ctrl-C Ctrl-C, which will raise a `KeyboardInterrupt` error. Since this error is not handled, then the script will be completely interrupted. This is an unhandled error.

It is possible to handle more than one type of exception:

```
while True:
    try:
        number = int(input("Please enter an integer:"))
    except ValueError:
        print("Invalid input. Not an integer, please try again.")
    except KeyboardInterrupt:
        print("\n\n")
        print("Input interrupted by the keyboard.")
```

```
        break                    # break the while loop so the script stops
    else:
        print(f'Integer = {number}')
        break
```

If we hit Ctrl-C Ctrl-C, then the script will raise a `KeyboardInterrupt` exception. On Jupyter this is equivalent to going on the menu `Kernel` and clicking the button `Interrupt`. Which the second `except` will catch and handle. It hanldes the exception by printing a message and breaking out of the loop, causing the script to end. Notice that we have an `else` clause at the end. That clause will only be executed if no exception is raised. That happens if the user enters an integer number, in that case the number is printed and the loop is broken.

If the `except` keyword is used without specifying an exception type, then it will catch all exceptions. This is not a good practice since it will catch any exception that you may or may not expect. Only hanlde exceptions you know how to handle.

Lastly, there is a `finally` keyword that can be used in a `try` block:

```
try:
    raise KeyboardInterrupt
except ValueError:
    print('A value error occurred.')
finally:
    print("Goodbye")
```

The above will `raise` an exception which is not handled. Since the exception is not handled it will stop the execution of the script. However, before doing so, the code in the `finally` clause is executed.

The real world use of the `finally` clause is to guarantee that resources are correctly closed, like closing a file or a network connection even if an error in the script occurred.

5 Assertions

Exceptions and the `try` blocks are used to tackle errors that you can handle. Assertions are used to alert you of an error that cannot be recovered (that should stop the execution of the script).

```
def discountPrice(price, discount):
    assert 0 <= discount <= 1, "Discount greater than actual price."
    return price*(1-discount)
```

The `assert` statement will check if the variable `discount` is between 0 and 1, and, if it is not, it will raise an `AssertionError` with the message "Discount greater than actual price".

If the `discountPrice` function sits in the middle of a payment system of some company, and at some point it is invoked with a discount that is greater than 100\We found a bug! If this assertion error occurred, then we need to stop the program and fix it. An assertion error should only be raised if there is a bug in your program.

6 Assignment

All assignments should be submitted to the Github repository you have been assigned to. The deadline is August 16th by 11 PM. You should write a report in Latex with the solutions to the problems below. If the problem requires you to code, then the code should also be included in the report. For a quick guide on how to add Matlab code to your Latex files, refer to the Section "Adding Matlab Code" on this tutorial.

Go to Codewars and sign in with your Github account. Solve the problems listed below. Your code must pass the submission test on the Codewars website.

6.1 Introductory Problems (start here)

- Opposite Number
- Even or Odd
- Sum of Positive Numbers
- Repeat Strings
- Strip First and Last Character
- Remove White Space
- Counting Sheeps
- Summation
- Basic Calculator
- Counting Numbers in an Array
- Summation with Exclusion

6.2 More Problems

- Count Vowels
- Middle Characters
- Numbers from a String
- Shortest String
- Reverse Digits
- DNA
- x and o
- Squares
- Square Digits
- Create Phone Number

6.3 Challenges

- Multiplicative Persistence
- Tribonacci Sequence
- Equal Sides of an Array
- Build a Tower