Intro to Python (Part I)

Kseniya Usovich

Introduction to Python

Table of Contents

Dependencies:

In [ ]:
```
import numpy as np
import pandas as pd
import math
```

Variable Assignment

We should think of a variable as a container or a storage box that allows us to save various types of information we plan to use later. Defining a variable consists of two components: a name it will be called by and information you store under that name. One variable can hold one type of information.

In [ ]:
```
# EXAMPLE

var = "Variable"
var
```

In [ ]:
```
print(var)
```

In the next cell assign the variable hello to a phrase "Hello World".

In [ ]:
```
# EXERCISE

hello  = "..."
hello
```

Data Types

Every programming language has different types of data that it can use. A data type is essentially a way the information can be saved and manipulated. Each data type has its own functions and methods that can modify it. And usually, the functions that can modify one data types, won't work with the other or give a different result. That is why it is important to know your data types.

Strings ¶

A string is a data types that consists of textual information. It will read all types of data as a text, even if you were to use numbers. Strings are defined by the quotation marks (either double or single).

In [ ]:
```
# EXAMPLE

"Kseniya Usovich"
```

Textual info without the quotation marks is read as a variable, and if that variable doesn't contain any info will give you an error.

In [ ]:
```
# EXAMPLE

Kseniya
```

You can save your string into a variable. In the cell below, create a variable name with your name in it.

In [ ]:
```
# EXERCISE

name = "..."
name
```

As was mentioned before, you can use either double or single quotation marks. But be careful, because some of the textual information might already contain the single (more common) or double (less common) quotation marks inside. Like so:

In [ ]:
```
# EXAMPLE

"Joe's"
```

If we were to use single quotation marks on the outside, it would give us an error:

In [ ]:
```
# EXAMPLE

'Joe's'
```

In order to avoid that error, simply put a backslash before the quotation mark, so the computer will read it as part of the string instead of a part of the code itself.

In [ ]:
```
# EXAMPLE

'Joe\'s'
```

Everything inside that quotation marks will be read as a textual information. Compare the outputs in the cell below.

In [ ]:
```
# EXAMPLE

print("2+3")
print(2+3)
```

String Methods

There are a few useful built-in functions and methods we can use for strings. We can either capitalize all the letters or change them all to a lower-case.

In [ ]:
```
# EXAMPLE

print(name.upper())
print(name)
```

As you can see, the name hasn't changed. What should we do to change all the letters to upper-case for good?

In [ ]:
```
#EXERCISE

...
```

The way to change all the letters to lower-case is very similar to the one we have shown above.

In [ ]:
```
# EXAMPLE

print(name.lower())
print(name)
```

To know what a method does, you can use **?** after it to learn what it is doing.

In [ ]:
```
# We use "str" for string

str.lower?
```

String Functions

Another built-in function you will see a lot is **len( )**. It allows us to count the amount of characters (string) or objects (lists, arrays, etc.).

In [ ]:
```
# EXAMPLE


len(name)
```

Before running the code cell below, consider what the output will be.

In [ ]:
```
# EXAMPLE

new_string = "Good Morning"

len(new_string)
```

Did the output match your prediction? Why or why not?

Numerals

Numerals essentially are just numbers. And you can do all kinds of manipulations with them as you would do with numbers. There are two types of numerals: int and float. "Int" is an integer and "float" is a number with a decimal point. In the previous versions of Python they were not "compatible", meaning you were only able to manipulate the numerals of the same kind. But starting with Python3, you can combine both types in the calculation without it causing any error.

In [ ]:
```
# EXAMPLE

2+3
```

In [ ]:
```
# EXAMPLE

2.0+3
```

In [ ]:
```
# EXAMPLE

4/2
```

In [ ]:
```
# EXAMPLE

7//2
```

In [ ]:
```
# EXAMPLE

5*3
```

In [ ]:
```
# EXAMPLE

5**3
```

In [ ]:
```
# EXAMPLE

(1+2)*3
```

You can also save a numeral in a variable. You can later use that variable instead of typing a number in your calculations.

In [ ]:
```
# EXAMPLE

g = 9.8
```

In [ ]:
```
# EXAMPLE

g*4
```

Booleans

Booleans can only take `True` or `False` values. They represent the "truth" values in a logical expression, and can help us answer questions that require a binary responses (yes or no) whenever a given condition is or is not met.

- "boolean values" have a **bool** data type.

Let's use the function type to check what type of data type are is True and False.

In [ ]:
```
type(True)
```

```
In [ ]: type(False)
```

Truth Values

All of the following evaluate to False.

```
In [ ]: False == False
```

```
In [ ]: False == 0
```

```
In [ ]: False == 0.0
```

Note: There are also a couple of other **False** values that we will cover later, such as **None**, **''**, **()**, **{}**, **[]**. On other words, empty sequences and empty mappings evaluate to False.

Booleans also have operations. These are **or** and **and**. Let's look at the table below.

Boolean Operations

These are ordered by ascending priority.

operation	Result
x **or** y	if x is false, then y, else x
x **and** y	if x is false, then x, else y
not x	if x is false, then True, else False

Data Structures

Lists

List is a data type that allows you to save multiple objects. In comparison with strings, lists can contain different data types.

In [ ]:
```
# here's how you can create an empty list
# you'll learn where to use it later in the workshop

empty_lst = []
empty_lst
```

In [ ]:
```
# EXAMPLE

new_lst = [2, "name", 3, [5, 6, 7], "berkeley"]
new_lst
```

Make sure you don't use the built-in functions as names for your variables. If you were to call your list "list", then the function with the same name will not work in your notebook anymore. It is easy to tell (in Jupyter Notebooks) whether something is a function/method. When you enter it, it be in green in your code cell.

In [ ]:
```
list
```

Let's see how we can use this function on a string.

In [ ]:
```
new_name = list(name)
new_name
```

List Iteration and Slicing `[ : ]`

To iterate through the list, you can use square brackets with an index of an object you are interested in. Like so:

In [ ]:
```
# remember that Python, like many programming languages, starts cou
nting from 0.

new_lst[1]
```

Notice also that each component of the list separated by a comma is a single object. So the list inside of the list will be a single object. If you want to iterate through a list inside of the list, you will need to use the square brackets twice. Like we did here:

In [ ]:
```
# EXAMPLE

new_lst[3][0]
```

You can also replace the objects in the list by using iteration.

In [ ]:
```
# EXAMPLE

new_lst[0] = 6
new_lst
```

Mutability

There are a few ways you can copy your list over into a new variable. But these copies will behave differently.

Shallow Copy

In [ ]:
```
# before running this cell, think about the output of lst_2 & new_l
st

lst_2 = new_lst
lst_2[2] = 7

print(new_lst)
print(lst_2)
```

In [ ]:
```
# EXAMPLE

shallow_lst = new_lst
print('This is a copy of new_lst:', shallow_lst)

new_lst[0] = "New Value"
print("Changed first value in new_lst:", new_lst)
print("Values in shallow_lst", shallow_lst, " are same as new_lst")
```

Deep Copy

In [ ]:
```
# compare it to this way of copying

lst_3 = new_lst[:]
lst_3[2] = 8

print(new_lst)
print(lst_3)
```

Slicing creates copies

In [ ]:
```
# EXAMPLE

shallow_lst = new_lst[:3]
print('First three values:', shallow_lst)

new_lst[1] = "New Value"
print("Changed new_lst:", new_lst)
print("First three values", shallow_lst, "SAME!")
```

List Methods

Just like the data types we studied before, lists have built-in functions and methods.

Adding Elements

To add elements to a list, we can use a few fucntions. The ones we will show you today are built-in functions **insert** and **append**.

In [ ]:
```
# EXAMPLE

trees = ["Sequoia", "Palm Tree", "Joshua Tree"]

# to add an element to the end of the list, we can use "append"

trees.append("Pine Tree")
trees
```

We use **insert** when we want to put something at a certain position. But be careful with it, you actually need to specify the position at which you are inserting an element.

In [ ]:
```
# EXAMPLE

trees.insert(2, "Redwood")
trees
```

In the cell below, try adding one tree of your liking to the position 1 and then another tree at the end of the list.

In [ ]:
```
# EXERCISE


...
print(trees)


...
print(trees)
```

Deleting Elements

Python has 3 built-in functions and methods for deleting elements from a list.

In [ ]:
```
random_list = ['Zoom', 1, 7, 9, "Python", [3,8], "Berkeley"]
```

Method **del** works with indeces and allows for the deletion of parts of the list through slicing.

In [ ]:
```
# EXAMPLE

del random_list[0:2]

random_list
```

Method **.pop( )** works with indeces. It is the last symbol/character/object by default. But you can add a specific index.

In [ ]:
```
# EXAMPLE

random_list.pop()
```

In [ ]:
```
# EXAMPLE

random_list.pop(0)
```

Notice that **.pop( )** also outputs the thing it has removed from a list.

In [ ]:
```
# EXAMPLE

state = ["California", "is", "on the", "West Coast"]
last = state.pop()

print(state)
print(last)
```

Another method you can use for lists is **.remove**. This method uses the exact values (case matters too).

In [ ]:
```
# EXAMPLE

random_list.remove("Python")
```

In [ ]:
```
random_list
```

Now you try removing any three elements from this list three different ways.

In [ ]:
```
# EXERCISE

# We created a new list with 7 elements in it

california = ["Los Angeles", "San Francisco", 1, 5, 80, "San Diego"
, ["49ers", "Lakers"]]

# Use 3 different methods of deleting elements from the list

... california[...]
california ...
california ...

print(california)
```

Length of a list

Built-in function **len( )** can also be used with lists.

In [ ]:
```
# EXAMPLE
# Before running this cell, think what the output will be.

countries = ["USA", "Belarus", "Mexico", "Poland"]
len(countries)
```

How is the **len( )** different for strings and lists?

Arrays

Arrays are another data type that is similar to lists in a way, but they only allow for one type of information to be saved in them. Where a list can have multiple data types in it, each array can only have objects of one type (either numerals, or strings, or lists, etc.). They also use different methods and functions, that is why it is important to know your data types.

In [ ]:
```
# EXAMPLE

# here we create two variables with numbers from 0 to 6, one is an array,
# and the other is a list

x_arr = np.arange(7)
x_list = [0, 1, 2, 3, 4, 5, 6]
```

In [ ]:
```
# EXAMPLE

x_arr*3
```

In [ ]:
```
x_list*3
```

In [ ]:
```
# EXAMPLE

x_arr[2] = 5
x_arr
```

Arrays are most commonly used with tables, which we will not interact with in this workshop. Arrays are most useful when you want to have a set of the same kind of data that you can easily modify together.

Tuples

Tuples look and behave similarly to arrays, but they **cannot** be modified. It can be used with geospatial data or other data that shouldn't be modified under any circumstances.

In [ ]:
```
# EXAMPLE

tup = tuple([2, 3, 4])
tup
```

In [ ]:
```
# EXAMPLE

tup[1]
```

In [ ]:
```
# EXAMPLE

tup[1] = 5
```

Dictionaries

Another powerful data structure is the dictionary. Dictionaries help us store and manipulate data relations.

What are dictionaries?

- A dictionary is a collection of unordered, mutable and indexable data types. They are structured as key-value pairs.

About:

- built-in
- built using curly brackets - {key: value}
- you can access the value using the key (like a real dictionary)
- keys tend to be strings (or text), since typically they represent the title or name of the thing(s) you want to represent.

In [ ]:
```
# EXAMPLE

# favorite movies for a group of 4 students

movies = {"Karla" : "Blade Runner", "Andy" : "Toy Story", "Sam" : "
It", "Jennifer" : "Harry Potter and the Sorcerer's Stone"}
movies
```

In here, the keys are the names of the students - `Karla, Andy, Sam, and Jennifer`, and values are the names of the favorite movie for each student - `Blade runner, Toy story, It, and Harry Potter and the Sorcerer's Stone` respectively.

In [ ]:
```
# EXAMPLE

# Features of my pet. We want to know what to form a relationship
# between my pet and it's gestation type, breed, gestation
# period (in days), and average lifespan (in years).

my_pet = {"name" : "Guido", "type" : "golden retriever", "gestation
_period" : 63, "life_expectancy" : 11}
my_pet
```

Notice, that we are able to store different data types in the value filed. For `name` and `type` we have strings, and for `gestation_period` and `life_expectancy` we have numerals.

In [ ]:
```
# EXAMPLE

# Now, we want to collect the same information as above
# but for different pets, but now we will associate it the
# name of the owner. For this exampe, we will say that  Karla
# has a dog, Sam has a cat, and Alan has hamster.

our_pets = {"Karla" : {"name" : "Guido", "type" : "golden retriever
", "gestation_period" : 63, "life_expectancy" : 11},
            "Sam" : {"name" : "Frati", "type" : "siamese", "gestati
on_period" : 60, "life_expectancy" : 17},
            "Alan" : {"name" : "Boots", "type" : "syrian", "gestati
on_period" : 17, "life_expectancy" : 2.5} }
our_pets
```

Wow! We just created a dictionary of dictionaries! There are different ways to arrange the information that we stored above. For instance, we could have used lists instead.

In [ ]:
```
# EXAMPLE

our_pets_2 = {"Owner":["Karla", "Sam","Alan"], "Name":["Guido","Fra
ti","Boots"], "type":["golden retriever","siamese","syrian"],"gesta
tion_period":[63,60,17],"life_expectancy":[11,17,2.5]}
our_pets_2
```

If this is a bit hard to understand, do not worry! All you need to remember is that they are key-value pairs that store data. Now, let's see how we can access and manipulate that data.

Accessing Elements

The elements in a dictionary you can use the bracket notation we had learned before. It will return the value if the "key" exists, otherwise it will return a KeyError.

Let's use the movies example from above.

In [ ]:
```
# Run this cell to see the dictionary

movies
```

In [ ]:
```
# EXAMPLE

movies['Sam']
```

In [ ]:
```
# EXERCISE

# What movie does Andy like?
```

In the example above, "Sam" is the key, and "It" is value. Now, let's try to access a key that is NOT on the dictionary.

In [ ]:
```
# EXAMPLE

movies['Jen']
```

Inserting and Changing Elements

Changing Elements

Let's begin by changing the Sam's favorite movie to "Wizard of Oz".

In [ ]:
```
# EXAMPLE

movies['Sam']= "Wizard of Oz"

movies
```

In [ ]:
```
# EXERCISE

# Change the movie that Andy likes to your favorite movie
```

Adding Elements

We can add and insert elements in a dictionary the same way.

In [ ]:
```
# EXAMPLE

movies['Karina'] = "Coco"

movies
```

In [ ]:
```
# EXERCISE

# Add new person and their favorite movie to the dictionary
```

Removing Elements

To remove an element, we have to use the function `del`.

In [ ]:
```
# EXAMPLE

del movies['Karla']
movies
```

In [ ]:
```
# EXERCISE

# Remove the entry you added on the previous exercise
```

Notebook developed by: Kseniya Usovich & Karla Palos

Cal NERDS GitHub: https://github.com/Cal-NERDS (https://github.com/Cal-NERDS)