# APSC-5984 Lab 3: Python Basics II

Due: 2023-02-06 (Monday) 23:59:59



## 0. Overview

In this lab, you will learn more Python syntax. We will cover the essential concepts of Python:

- Lists
- NumPy Array
- Dictionaries
- Control structures: `For` loops and `while` loops

As usual, you will need to open the `labs/lab_03/assignment.ipynb` file in VS Code and follow the instruction to complete this lab assignment.

(Back to top)

## 1. Lists

### 1.1 Assign values to a list

A list is a collection of items in a particular order. You can make a list that includes variables, numbers, or strings. For example, you can make a list that includes the letters [a, b, c, d, e]:

```
letters = ['a', 'b', 'c', 'd', 'e']
print(letters) # ['a', 'b', 'c', 'd', 'e']
```

Same rules apply to numbers:

```
numbers = [1, 2, 3, 4, 5]
print(numbers) # [1, 2, 3, 4, 5]
```

You can also make a list that includes both strings and numbers:

```
mixed = ['a', 1, 'b', 2, 'c', 3]
print(mixed) # ['a', 1, 'b', 2, 'c', 3]
```

You might notice that there is a case where certain items in a list are related to each other. For example, you might want to store the name and age of a person in a list. In this case, you can use a nested list:

```
person_age_1 = ['John', 20]
person_age_2 = ['Mary', 25]
person_age_3 = ['Michael', 30]
nested_list = [person_age_1, person_age_2, person_age_3]
print(nested_list) # [['John', 20], ['Mary', 25], ['Michael', 30]]
```

Append a new element to a list is easy. We can add a new person to the list `nested_list` by using the `append` method:

```
nested_list.append(['Elizabeth', 35])
print(nested_list) # [['John', 20], ['Mary', 25], ['Michael', 30],
['Elizabeth', 35]]
```

### 1.2 Accessing elements in a list

You can access elements in a list by using the index (i.e., the position) of the element. In Python, the index of a list starts from 0, which means the first element in a list has index 0, the second element has index 1, and so on. For example, you can access the first element in the list `letters` by using `letters[0]`:

```
letters = ['a', 'b', 'c', 'd', 'e']
print(letters[0]) # a
```

The third element in the list `nested_list` can be accessed by using `nested_list[2]`:

```
nested_list = [['John', 20], ['Mary', 25], ['Michael', 30]]
print(nested_list[2]) # ['Michael', 30]
```

In addition, indexing can be concatenated. Following the same list `nested_list`, if you want to obtain the age of the third person in the list `nested_list`, you can do `nested_list[2][1]`:

```
mike_age = nested_list[2][1]
print(mike_age) # 30
```

What if you want to access the last element in a list? You can use negative index to access the last element in a list. For example, you can access the last element in the list `letters` by using `letters[-1]`:

```
letters = ['a', 'b', 'c', 'd', 'e']
print(letters[-1]) # e
```

Alright, now you know how to access elements in a list. What if you want to access a range of elements in a list? You can use the `:` operator to access a range of elements in a list. For example, you can access the first three elements in the list `letters` by using `letters[0:3]`:

```
letters = ['a', 'b', 'c', 'd', 'e']
print(letters[0:3]) # ['a', 'b', 'c']
```

It is noted that the number after the `:` operator is not included in the range. In the above example, the number 3 is not included in the range. If you want to access the first four elements (i.e., 0, 1, 2, and 3) in the list `letters`, you can use `letters[0:4]`:

```
letters = ['a', 'b', 'c', 'd', 'e']
print(letters[0:4]) # ['a', 'b', 'c', 'd']
```

If you have a long list and you do not want to specify the end index, you can use `letters[3:]` to access the elements from the fourth element to the last element:

```
letters = ['a', 'b', 'c', 'd', 'e']
print(letters[3:]) # ['d', 'e']
```

Same rules apply to negative index. If you want to access the last three elements in the list `letters`, you can use `letters[-3:]`:

```
letters = ['a', 'b', 'c', 'd', 'e']
print(letters[-3:]) # ['c', 'd', 'e']
```

You can mix the positive index and negative index. For example, you can access elements from the second position to the last second position in the list `letters` by using `letters[1:-1]`:

```
letters = ['a', 'b', 'c', 'd', 'e']
print(letters[1:-1]) # ['b', 'c', 'd']
```

(Back to top)

## 2. NumPy Array

Numpy is one of the mostly used Python library in data sciences. It provides an versatile interface for users to manipulate multi-dimensional arrays, particularlly for image data. Practically, we use `np` as an aliase name for the NumPy library. But the aliase can also be other arbitrary names.

```
import numpy as np
np.__version__
# '1.23.1'
```

### 2.1 Create a 1D array

Put a list into `np.array()` to create a 1D numpy array.
(https://numpy.org/doc/stable/reference/generated/numpy.array.html)

```
ls_1d = [1, 2, 3]
np_1d = np.array(ls_1d)

print(ls_1d)
# [1, 2, 3]
print(np_1d)
# array([1 2 3])
```

All-zero or all-one array can be created by `np.zeros()` and `np.ones()`.
(https://numpy.org/doc/stable/reference/generated/numpy.zeros.html)
(https://numpy.org/doc/stable/reference/generated/numpy.ones.html)

```
np.zeros(3)
# array([0., 0., 0.])
np.ones(3)
# array([1., 1., 1.])
```

Create an arithmetic sequence: `np.arange()` creates a sequence defined by an interval.
(https://numpy.org/doc/stable/reference/generated/numpy.arange.html)

`np.linespace()` creates a sequence defined by a size.
(https://numpy.org/doc/stable/reference/generated/numpy.linspace.html)

```
np.arange(1, 10, 2)
# array([1, 3, 5, 7, 9])
np.linspace(1, 10, 5)
# array([ 1.  ,  3.25,  5.5 ,  7.75, 10.  ])
```

Challenge: Create an array with the same elements `[5, 10, 15, 20]` using two different functions: `np.linspace()` and `np.arange()`

```
array_A = np.arange(5, 21, 5)
array_B = np.linspace(5, 20, 4)
print(array_A)
# array([ 5, 10, 15, 20])
print(array_B)
# array([ 5., 10., 15., 20.])
```

## 2.2 Multi-dimensional matrix

Create a 2D array by putting a nested list into `np.array()`.

```
nested_list = [[1, 2, 3], [4, 5, 6]]
np_2d = np.array(nested_list)
print(np_2d)
# array([[1, 2, 3],
#        [4, 5, 6]])
```

You can check the dimension of an array by `np.shape()`.
(https://numpy.org/doc/stable/reference/generated/numpy.shape.html)

In this example, the matrix has two rows and three columns:

```
np.shape(np_2d)
# (2, 3)
```

Same rule applies to all-zero or all-one matrix:

```
np.zeros((2, 3))
# array([[0., 0., 0.],
```

```
#        [0., 0., 0.]])
np.ones((2, 3))
# array([[1., 1., 1.],
#        [1., 1., 1.]])
```

Reshape an array by `np.reshape()` and `np.flatten()`.
(https://numpy.org/doc/stable/reference/generated/numpy.reshape.html)
(https://numpy.org/doc/stable/reference/generated/numpy.flatten.html)

```
np_1d = np.arange(1, 7)
print(np_1d)
# array([1, 2, 3, 4, 5, 6])
np_2d = np_1d.reshape(2, 3)
print(np_2d)
# array([[1, 2, 3],
#        [4, 5, 6]])
print(np_2d.flatten())
# array([1, 2, 3, 4, 5, 6])
```

Practice with a 3D-array:

```
np_3d = np.arange(1, 13).reshape(2, 2, 3)
print(np_3d)
# array([[[ 1,  2,  3],
#         [ 4,  5,  6]],
#        [[ 7,  8,  9],
#         [10, 11, 12]]])
np_1d = np_3d.flatten()
print(np_1d)
# array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

## 2.3 Indexing and slicing

Indexing and slicing in `numpy` are the same as in `list`. Let's create 2D array first:

```
np_2d = np.linspace(1, 30, 30).reshape((5, 6))
print(np_2d.shape)
# (5, 6)
print(np_2d)
# array([[ 1.,  2.,  3.,  4.,  5.,  6.],
#        [ 7.,  8.,  9., 10., 11., 12.],
#        [13., 14., 15., 16., 17., 18.],
#        [19., 20., 21., 22., 23., 24.],
#        [25., 26., 27., 28., 29., 30.]])
```

Get the first column:

```
np_2d[:, 0]
# array([ 1.,  7., 13., 19., 25.])
```

Get the second and third rows:

```
np_2d[1:3, :]
# array([[ 7.,  8.,  9., 10., 11., 12.],
#        [13., 14., 15., 16., 17., 18.]])
np_2d[[1, 2], :]
# array([[ 7.,  8.,  9., 10., 11., 12.],
#        [13., 14., 15., 16., 17., 18.]])
```

Negative indexing is also aupported:

```
np_2d[-2:, :]
# array([[19., 20., 21., 22., 23., 24.],
#        [25., 26., 27., 28., 29., 30.]])
```

Get one every two columns:

```
np_2d[:, ::2]
# array([[ 1.,  3.,  5.],
#        [ 7.,  9., 11.],
#        [13., 15., 17.],
#        [19., 21., 23.],
#        [25., 27., 29.]])
```

## 2.4 Basic statistics

`numpy` provides a lot of useful functions for basic statistics. Here are some examples:

```
np_2d = np.linspace(1, 30, 30).reshape((5, 6))
print(np_2d)
# array([[ 1.,  2.,  3.,  4.,  5.,  6.],
#        [ 7.,  8.,  9., 10., 11., 12.],
#        [13., 14., 15., 16., 17., 18.],
#        [19., 20., 21., 22., 23., 24.],
#        [25., 26., 27., 28., 29., 30.]])
print(np.mean(np_2d))
# 15.5
print(np.median(np_2d))
# 15.5
print(np.std(np_2d))
# 8.65544144839919
```

```
print(np.var(np_2d))
# 74.91666666666667
print(np.min(np_2d))
# 1.0
print(np.max(np_2d))
# 30.0
print(np.sum(np_2d))
# 465.0
```

### 2.5 Axis-wise operations

Sometimes, we want to perform operations on a specific axis, such as the mean of each column. We can use `axis` argument to specify the axis. `axis=0` means the first axis (rows), `axis=1` means the second axis (columns), and so on. Here is an example:

```
np_3d = np.linspace(1, 30, 30).reshape((3, 2, 5))
print(np_3d)
# array([[[ 1.,  2.,  3.,  4.,  5.],
#         [ 6.,  7.,  8.,  9., 10.]],
#        [[11., 12., 13., 14., 15.],
#         [16., 17., 18., 19., 20.]],
#        [[21., 22., 23., 24., 25.],
#         [26., 27., 28., 29., 30.]]])
print(np.mean(np_3d, axis=0))
# array([[11., 12., 13., 14., 15.],
#        [16., 17., 18., 19., 20.]])
print(np.mean(np_3d, axis=1))
# array([[ 3.5,  4.5,  5.5,  6.5,  7.5],
#        [13.5, 14.5, 15.5, 16.5, 17.5],
#        [23.5, 24.5, 25.5, 26.5, 27.5]])
print(np.mean(np_3d, axis=2))
# array([[ 3.,  8.],
#        [13., 18.],
#        [23., 28.]])
```

Multiple axes can be specified at the same time:

```
print(np.mean(np_3d, axis=(0, 2)))
# array([12., 17., 22., 27.])
```

(Back to top)

## 3. Dictionaries

### 3.1 Creating a dictionary

A dictionary is a collection of key-value pairs. Unlike a list, where items are accessed by their position, items in a dictionary are accessed via keys. A key's value can be a number, a string, a list, or even another dictionary. In Python, a dictionary is defined by curly brackets `{}`. Here is an example:

```
person_age = {'John': 20, 'Mary': 25, 'Michael': 30, 'Elizabeth': 35}
print(person_age) # {'John': 20, 'Mary': 25, 'Michael': 30, 'Elizabeth':
35}
```

In this case, the keys are `John`, `Mary`, `Michael`, and `Elizabeth`, and the values are 20, 25, 30, and 35, respectively. To add a new key-value pair to a dictionary, you have two options. First, you can use the `update` method. For example, you can add a new key-value pair to the dictionary `person_age` by using `person_age.update({'David': 40})`:

```
person_age.update({'David': 40})
print(person_age) # {'John': 20, 'Mary': 25, 'Michael': 30, 'Elizabeth':
35, 'David': 40}
```

Second, you can use the `[]` operator. For example, you can add a new key-value pair to the dictionary `person_age` by using `person_age['David'] = 40`:

```
person_age['David'] = 40
print(person_age) # {'John': 20, 'Mary': 25, 'Michael': 30, 'Elizabeth':
35, 'David': 40}
```

You can also use integers as keys:

```
dict_int = {3: "three", 10: "ten", 2: "two"}
print(dict_int) # {3: 'three', 10: 'ten', 2: 'two'}
```

### 3.2 Accessing elements in a dictionary

You can access the value of a key by using the key name. For example, you can access the age of John by using `person_age['John']` or `person_age.get('John')`:

```
john_age = person_age['John']
print(john_age) # 20\
john_age = person_age.get('John')
print(john_age) # 20
```

What if you want to access to the value of the last key in a dictionary? Although you cannot use negative index like you did in a list, you can still access the value in a dictionary. You can use the `keys()` method to

get a list of keys in a dictionary, and this function should return a list of keys. With this list, you can access the last key in the dictionary and get the value of the last key. Here is an example:

```
# step 1: get a list of keys in a dictionary
keys = person_age.keys()
print(keys) # dict_keys(['John', 'Mary', 'Michael', 'Elizabeth', 'David'])

# step 2: access the last key in the dictionary
last_key = list(keys)[-1]
print(last_key) # David

# step 3: access the value of the last key
last_value = person_age[last_key]
print(last_value) # 40
```

On the other hand, you can use the `values()` method to get a list of values in a dictionary. For example, you can get a list of ages in the dictionary `person_age` by using `person_age.values()`:

```
values = person_age.values()
print(values) # dict_values([20, 25, 30, 35, 40])
```

It is also possible to query whether a key exists in a dictionary. You can use the `in` operator to check whether a key exists in a dictionary. For example, you can check whether the key `David` exists in the dictionary `person_age` by using `if 'David' in person_age:`:

```
if 'David' in person_age:
    print("David exists in the dictionary person_age")
else:
    print("David does not exist in the dictionary person_age")
```

(Back to top)

## 4. Loops

It is quite a hassle to repeatly access elements in a list or a dictionary. In this section, we will introduce how to use `for` loops and `while` loops in Python. These two types of loops are very useful to interact with these data structures.

### 4.1 `for` loops

A `for` loop is used to iterate over a sequence (e.g., a list, a dictionary, or a string). There are two components in a `for` statement: `variable` and `iterators`. Let's take a look at an example:

```
for i in [0, 1, 2]:
    print(i)
```

```
# 0
# 1
# 2
```

In this example, `i` is the variable, and `[0, 1, 2]`, a list, is the iterator. The `for` loop will iterate over the iterator and assign the value of each element to the variable. In this case, the value of `i` will be 0, 1, and 2, respectively. After the value of `i` is assigned, the code block `print(i)` will be executed. The variable name can be anything you want. Conventionally, we use `i` as the variable name.

In Python, `range(3)` is a built-in function that returns a list of integers from 0 to 2. You can use `range(3)` to replace `[0, 1, 2]` in the above example:

```
for i in range(3):
    print(i)
# 0
# 1
# 2
```

As we mentioned above, the iterators can be a list, a dictionary. Let's see what happens if we use the dictionary `person_age` as the iterator:

```
for i in person_age:
    print(i)
# John
# Mary
# Michael
# Elizabeth
# David
```

You would see that the variable `i` is assigned to the keys (not the values) in the dictionary `person_age`. In this case, the variable `i` is assigned to `John`, `Mary`, `Michael`, `Elizabeth`, and `David`, respectively. If you want to access the values in the dictionary `person_age`, you can use the `[]` operator:

```
for key in person_age:
    print(person_age[key])
# 20
# 25
# 30
# 35
# 40
```

The above example is equivalent to the following example:

```

```

```
print(person_age['John'])
print(person_age['Mary'])
print(person_age['Michael'])
print(person_age['Elizabeth'])
print(person_age['David'])
# 20
# 25
# 30
# 35
# 40
```

There is a handy alternative to get both the keys and values in a dictionary is to use the `items()` method:

```
for key, value in person_age.items():
    print(key, value)
# John 20
# Mary 25
# Michael 30
# Elizabeth 35
# David 40
```

### 4.2 `while` loops

A `while` loop is used to repeat a block of code until a condition is met. Again, let's take a look at an example:

```
i = 0
while i < 3:
    print(i)
    i = i + 1
# 0
# 1
# 2
```

In this example, The value of `i` was assigned to 0 before the loop starts. And the `while` loop will check the condition `i < 3` before each iteration. If the condition is met, the code block will be executed. Otherwise, the `while` loop will be terminated. In each iteration, the value of `i` will be increased by 1. After the value of `i` is increased, the condition `i < 3` will be checked again. If the condition is met, the iteration will continue.

The above example is equivalent to the following code:

```
i = 0
if i < 3:
    print(i)
    i = i + 1
    if i < 3:
```

```
        print(i)
        i = i + 1
        if i < 3:
            print(i)
            i = i + 1
            # ... and so on
        else:
            pass
    else:
        pass
else:
    pass
```

A quick summary: In practice, there is no strict rule to decide whether to use `for` loops or `while` loops. If you need to iterate over a sequence, a `for` loop is usually the better choice. If you need to repeat a block of code until a condition is met, a while `loop` could.

### 4.3 `break` and `continue`

In the above cases, the `for` loop and `while` loop will iterate over the entire sequence or continue to execute the code block until the condition is met. However, there are cases that you want to stop the iteration or skip the current iteration. In this case, you can use the `break` and `continue` statements.

The `break` statement is used to stop the iteration. For example, you can use `break` to stop the iteration when the value of `i` is 2:

```
for i in range(5):
    if i == 2:
        break
    print(i)
# 0
# 1
```

When `i` is 2, the `break` statement will stop the iteration and the code block `print(i)` will not be executed. It is noted that once the `break` statement is executed, the `for` loop will be terminated immediately; the value of `i` will not be printed.

The `continue` statement is used to skip the current iteration. For example, you can use `continue` to skip the iteration when the value of `i` is 2:

```
for i in range(5):
    if i == 2:
        continue
    print(i)
# 0
# 1
# 3
# 4
```

When `i` is 2, the `continue` statement will skip the current iteration and the code block `print(i)` will not be executed. The value of `i` will be increased by 1 and the condition `i < 5` will be checked again. If the condition is met, the iteration will continue.

(Back to top)