Introduction to Python

Outline

So far, we have become familiar with the fundamentals of Python. We have learned about basic variable types, conditional statements (if and else), and loops.
In this lecture, we will cover the following topics:

- Data Types: We will delve deeper into different data types in Python, specifically those used for storing multiple items in a single variable, namely **lists** and **dictionaries**.
- Functions: We will explore how to define and use functions in Python to organize and reuse code effectively.
- Object-Oriented Programming: Finally, we will examine the principles of object-oriented programming in Python, focusing on classes.

Python Lists

Lists are one of the 4 built-in data types in Python used to store collections of data. The other 3 are Tuples, Sets, and Dictionaries, all with different qualities and usage.

Here are a few points about lists in Python:

- Lists are created using square brackets.
- Lists are ordered, changeable, expandable, and allow duplicate values.
- List items are indexed, with the first item having index [0].

- If we add new items to a list, they will be placed at the end of the list.
- List items can be of any data type and they can be of different types.
- We use the ***len()*** function to determine the number of items in a list.

```
#list items can be of any type
#list items can be of different types
thislist = ["apple", "banana", 2]
print(thislist)

#indexed
print(thislist[0])

#changeable
thislist[0] = "orange"
print(thislist)

#using the len() function
print(len(thislist))
```

```
['apple', 'banana', 2]
apple
['orange', 'banana', 2]
3
```

Python Lists

We can specify a range of indices by specifying where to start and where to end the range.

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi"]
print(thislist[2:4])

#This will return the items from index 2 to 4(4 is excluded).
```

```
['cherry', 'orange']
```

We can add items to the end of a list by using the ***append()*** function.

```
thislist = ["apple", "banana", "cherry"]

thislist.append("orange")
print(thislist)
```

```
['apple', 'banana', 'cherry', 'orange']
```

By using the ***insert()*** function, we can add a new item at a specified index.

```
thislist = ["apple", "banana", "cherry"]

thislist.insert(1, "mango")
print(thislist)
```

```
['apple', 'mango', 'banana', 'cherry']
```

Python Lists

We can remove items from a list using the ***remove()*** function. If there are more than one item with the specified value, this method removes the first occurrence.

```
thislist = ["apple", "banana", "cherry", "banana", "kiwi"]
thislist.remove("banana")
print(thislist)
```

```
['apple', 'cherry', 'banana', 'kiwi']
```

By using the ***clear()*** function, we can empty a list.

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

```
[]
```

By using the ***pop()*** or the ***del()*** function, we can remove an item with a specified index. If the index is not specified, the *pop()* method removes the last item.

```
thislist = ["apple", "banana", "cherry", "mango"]
thislist.pop(2)
print(thislist)

del thislist[1]
print(thislist)

thislist.pop()
print(thislist)
```

```
['apple', 'banana', 'mango']
['apple', 'mango']
['apple']
```

Python Lists

We can loop through the items of a list in various ways. We can use a for loop, a while loop or loop through the index numbers.

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```

```
apple
banana
cherry
```

```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
  print(thislist[i])
  i = i + 1
```

```
apple
banana
cherry
```

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
  print(thislist[i])
```

```
apple
banana
cherry
```

We can also use a short hand for loop that will print all items in a list

```
thislist = ["apple", "banana", "cherry"]
[print(x) for x in thislist]
```

```
apple
banana
cherry
```

Python Dictionaries

Dictionaries are used to store data values in *key: value* pairs.
Here are a few points about dictionaries in Python:

- Dictionaries are created using curly brackets.
- Dictionaries are, changeable, expandable, and do not allow duplicate keys.
- Items can be referred to by using the key name.
- Although items in a dictionary have a defined order, and that order will not change, we cannot refer to an item by using an index.

```
#values can be of any type
#values can be of different types
thisdict =  {"brand": "Ford", "model": "Mustang", "year": 1964}

print(thisdict)
print(thisdict["brand"])

#changeable
thisdict["year"] = 1968
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
Ford
{'brand': 'Ford', 'model': 'Mustang', 'year': 1968}
```

Python Dictionaries

- We can access an item in a dictionary by referring to its key name, inside square brackets. We can also use the ***get()*** function for the same result.
- We can see the list of keys of a dictionary by using the ***keys()*** function.
- We can see the list of values of a dictionary by using the ***values()*** function.
- We can see the items of a dictionary in *key: value* pairs by using the ***items()*** function.

```
thisdict =  {"brand": "Ford", "model": "Mustang", "year": 1964}

x = thisdict["model"]
print(x)

y = thisdict.get("brand")
print(y)

z = thisdict.keys()
print(z)

w = thisdict.values()
print(w)

v = thisdict.items()
print(v)
```

```
Mustang
Ford
dict_keys(['brand', 'model', 'year'])
dict_values(['Ford', 'Mustang', 1964])
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
```

Python Dictionaries

By using the ***update()*** method, we can update the dictionary with the items from the given argument. Note that when using the this method, If the key does not exist, the item will be added.

```
thisdict =  {"brand": "Ford", "model": "Mustang", "year": 1964}

thisdict.update({"year": 2024})
print(thisdict)

thisdict.update({"color": "red"})
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2024}
{'brand': 'Ford', 'model': 'Mustang', 'year': 2024, 'color': 'red'}
```

We can also use a new index key and assign a value to it, in order to add a new item to a dictionary.

```
thisdict =  {"brand": "Ford", "model": "Mustang", "year": 1964}

thisdict["color"] = "red"
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}
```

Python Dictionaries

We can remove an item with a specified key from a dictionary, using the ***pop()*** or the ***del()*** function.

```
thisdict =  {"brand": "Ford", "model": "Mustang", "year": 1964}

thisdict.pop("brand")
print(thisdict)

del thisdict["model"]
print(thisdict)
```

```
{'model': 'Mustang', 'year': 1964}
{'year': 1964}
```

By using the ***clear()*** function, we can empty a dictionary.

```
thisdict =  {"brand": "Ford", "model": "Mustang", "year": 1964}

thisdict.clear()
print(thisdict)
```

```
{}
```

Python Dictionaries

We can loop through the items of a dictionary in various ways.
We can print the keys of a dictionary in two ways.

```
thisdict =  {"brand": "Ford", "model": "Mustang", "year": 1964}

for x in thisdict:
  print(x)

print("\n")

for x in thisdict.keys():
  print(x)
```

```
brand
model
year


brand
model
year
```

We can print the values in a dictionary in two ways.

```
thisdict =  {"brand": "Ford", "model": "Mustang", "year": 1964}

for x in thisdict:
  print(thisdict[x])

print("\n")

for x in thisdict.values():
  print(x)
```

```
Ford
Mustang
1964


Ford
Mustang
1964
```

We can loop through both keys and values using the ***items()*** method.

```
thisdict =  {"brand": "Ford", "model": "Mustang", "year": 1964}

for x, y in thisdict.items():
  print(x, y)
```

```
brand Ford
model Mustang
year 1964
```

Python Functions

A function is a block of code which only runs when it is called. We can pass data as arguments, known as parameters, into a function. A function can return data as a result. In Python, a function is defined using the ***def*** keyword; and a function is called, using the name of the function, followed by parenthesis.
Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parenthesis. The number of arguments is not limited, and they are separated with commas.

```
def my_function():
  print("This is a function")

my_function()
```

```
This is a function
```

Note that a function must be called with the correct number of arguments. This means that the number of arguments given to a function when calling it, should be the same number of arguments we specified when defining the function.

```
def my_function(name):
  print("My name is " + name)

my_function("Mary")
```

```
My name is Mary
```

Python Functions

We can also send arguments with the *key = value* syntax. This way, the order of the arguments does not matter.

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

```
The youngest child is Linus
```

We can have default values for parameters. If we call the function without argument, it uses the default value.

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function()
```

```
I am from Sweden
I am from Norway
```

A function can return a value, using the ***return*** keyword.

```
def my_function(x):
  return 5 * x

print(my_function(3))
```

```
15
```

Function definitions cannot be empty, but if we want to have a function definition with no content, we must put in the ***pass*** statement.

```
def myfunction():
  pass
```

Python Classes

A class is a user-defined blueprint or prototype from which objects are created.
There are two main components in a class:

- **Attributes**: Variables that belong to a class. They hold the state of an object. For instance, in a class representing a dog, attributes might include *breed* and *age.*
- **Methods**: Functions defined within a class that describe the behaviors of an object. For example, a method could be *bark()* for a dog class.

We use the class keyword to create a class in Python.

```
class MyClass:
  x = 5

print(MyClass)
```

```
<class '__main__.MyClass'>
```

Now, we can create object from the class.

```
class MyClass:
  x = 5

p1 = MyClass()
print(p1.x)
```

```
5
```

Python Classes

All classes have a function called ***__init__()***, which is always automatically executed when the class is being initiated. It is used to initialize the attributes of the class.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("Mary", 26)

print(p1.name)
print(p1.age)
```

```
Mary
26
```

The ***__str__()*** function controls what should be returned when the class object is represented as a string. If it is not set, the string representation of the object is returned.

Here is an example in which the *__str__()* function is defined.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def __str__(self):
    return f"{self.name}({self.age})"

p1 = Person("John", 36)

print(p1)
```

```
John(36)
```

Python Classes

Methods in objects are functions that belong to the object. The ***self*** parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class. It does not have to be named *self* (this parameter can have any name); however, it has to be the first parameter of any function in the class.
Now, let us create a method in the Person class.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("My name is " + self.name)

p1 = Person("Mary", 26)
p1.myfunc()
```

```
My name is Mary
```

Python Classes

It is possible to modify properties on objects.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)


p1 = Person("Mary", 26)
p1.age = 28
print(p1.age)
```

28

We can delete properties on objects by using the ***del*** keyword.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)


p1 = Person("John", 36)
del p1.age
print(p1.age)
```

```
Traceback (most recent call last):
  File "demo_class7.py", line 13, in <module>
    print(p1.age)
AttributeError: 'Person' object has no attribute 'age'
```

Python Classes

We can delete objects by using the ***del*** keyword.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)


p1 = Person("John", 36)
del p1
print(p1)
```

```
Traceback (most recent call last):
  File "demo_class8.py", line 13, in <module>
    print(p1)
NameError: 'p1' is not defined
```

Class definitions cannot be empty, but if we want to have a class definition with no content, we must put in the pass statement to avoid getting an error.

```
class Person:
  pass
```