CSEL 562: PYTHON PROGRAMMING FOR DATA ANALYTICS

Module - I: Python Concepts, Data Structures, Classes

Introduction

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

Why python?

- Python can be used on a server to create web applications.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

The most recent major version of Python is Python 3 (3.11)

What is Python interpreter?

Python is an interpreter language. It means it executes the code line by line. Python provides a Python Shell, which is used to execute a single Python command and display the result.

Python Interpreters

- CPython.
- IronPython.
- Jython.
- PyPy.
- PythonNet.
- Stackless Python

Python Indentation

Indentation refers to the spaces at the beginning of a code line.

Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.

Python uses indentation to indicate a block of code.

```
if 5 > 2:
        print("Five is greater than two!")
```

Python Variables

In Python, variables are created when you assign a value to it:

Variables in Python:

```
x = 5
y = "Hello, World!"
```

Python has no command for declaring a variable.

Comment statement

```
#This is a comment.
print("Hello, World!")
```

Python Data Types

In programming, data type is an important concept.

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:

Text Type:	str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	dict
Set Types:	set, frozenset
Boolean Type:	bool
Binary Types:	bytes, bytearray, memoryview

Example datatypes

x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview

Get the Type

You can get the data type of a variable with the type() function.

Example

```
x = 5
y = "John"
print(type(x))
print(type(y))
```

Python Numbers

There are three numeric types in Python:

- int
- float
- complex

Int

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

Example

Integers:

```
x = 1
y = 35656222554887711
z = -3255522
print(type(x))
print(type(y))
print(type(z))
```

Float

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

Example

```
x = 1.10

y = 1.0

z = -35.59


print(type(x))

print(type(y))

print(type(z))
```

Complex

Complex numbers are written with a "j" as the imaginary part:

```
x = 3+5j
y = 5j
z = -5j

print(type(x))
print(type(y))
print(type(z))
```

Python statements

Python Conditions and If statements

Python supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

Example:

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

The while Loop

With the while loop we can execute a set of statements as long as a condition is true.

Example

```
i = 1
sum=0
n= int(input(“enter the number”))
while i < n:
  sum=sum+i
  i += 1
print(n)
```

The break Statement

With the break statement we can stop the loop even if the while condition is true:

Example

Exit the loop when i is 3:

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

Python For Loops

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

With the break statement we can stop the loop before it has looped through all the items:

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

The continue Statement

With the continue statement we can stop the current iteration of the loop, and continue with the next:

Example

Do not print banana:

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

The range() Function

To loop through a set of code a specified number of times, we can use the range() function,

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number. Example

```
for x in range(6):
  print(x)  # displays values from 0 to 5
```

range(2, 30), which means values from 2 to 30 (but not including 30). And increment value is 1

```
for x in range(2, 30, 3):
  print(x)
```

Output: 2
5
8
11
14
17
20
23
26
29

Python Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Example

```
def my_function():
  print("Hello from a function")

my_function()
```

Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
def my_function(fname):
  print(fname + " Refsnes")
my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Keyword Arguments

You can also send arguments with the key = value syntax.

This way the order of the arguments does not matter.

Example

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

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

Python Strings

Strings in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

You can display a string literal with the print() function:

Example

```
print("Hello")
print('Hello')
```

Strings are Arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

However, Python does not have a character data type, a single character is simply a string with a length of 1.

Square brackets can be used to access elements of the string.

Example

Get the character at position 1 (remember that the first character has the position 0):

```
a = "Hello, World!"
print(a[1])
```

Looping Through a String

Since strings are arrays, we can loop through the characters in a string, with a for loop.

Example

Loop through the letters in the word "banana":

```
for x in "banana":
  print(x)
```

Output:

```
b
a
n
a
n
a
```

String Length

To get the length of a string, use the len() function.

Example

The len() function returns the length of a string:

```
a = "Hello, World!"
print(len(a))
```

Output: 13

Check String

To check if a certain phrase or character is present in a string, we can use the keyword in.

Example

Check if "free" is present in the following text:

```
txt = "The best things in life are free!"
print("free" in txt)
```

Output: True

Use it in an if statement:

Print only if "free" is present:

```
txt = "The best things in life are free!"
if "free" in txt:
  print("Yes, 'free' is present.")
```

Output: Yes, 'free' is present.

Python - Slicing Strings

Slicing

You can return a range of characters by using the slice syntax.

Specify the start index and the end index, separated by a colon, to return a part of the string.

Get the characters from position 2 to position 5 (not included):

```
b = "Hello, World!"
print(b[2:5])
```

Output: llo

Slice From the Start

```
b = "Hello, World!"
print(b[:5])
```

Output: Hello

Slice To the End

```
b = "Hello, World!"
print(b[2:])
```

Output: llo, World!

Negative Indexing

Use negative indexes to start the slice from the end of the string:

Get the characters:

From: "o" in "World!" (position -5)

To, but not included: "d" in "World!" (position -2):

```
b = "Hello, World!"
print(b[-5:-2])
```

Output: orl

```
a = "Hello, World!"
print(a.upper())

a = "Hello, World!"
print(a.lower())
```

The **strip()** method removes any whitespace from the beginning or the end:

```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

The **replace()** method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

Output: Jello, World!

The **split()** method returns a list where the text between the specified separator becomes the list items.

The split() method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

String Concatenation

```
a = "Hello"
b = "World"
c = a + b
print(c)
```

we cannot combine strings and numbers like this:

Example

```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

Use the **format()** method to insert numbers into strings:

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

Example 2:

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

Output: I want to pay 49.95 dollars for 3 pieces of item 567

Escape Character

To insert characters that are illegal in a string, use an escape character.

An escape character is a backslash \ followed by the character you want to insert.

An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

txt = "We are the so-called "Vikings" from the north.“

To fix this problem, use the escape character \":

The escape character allows you to use double quotes when you normally would not be allowed:

txt = "We are the so-called \"Vikings\" from the north."

Other string methods

The first character is converted to upper case, and the rest are converted to lower case:

```
txt = "python is FUN!"
x = txt.capitalize()
print (x)  #output: Python is fun!
```

Check if the string ends with the phrase "my world.":

```
txt = "Hello, welcome to my world."
x = txt.endswith("my world.")
print(x)  #output: True
```

Return the number of times the value "apple" appears in the string:

```
txt = "I love apples, apple are my favorite fruit"
x = txt.count("apple")
print(x)   #output: 2
```

Where in the text is the word "welcome"?:

```
txt = "Hello, welcome to my world."
x = txt.find("welcome")
print(x)  #output: 7
```

Where in the text is the first occurrence of the letter "e" when you only search between position 5 and 10?:

```
txt = "Hello, welcome to my world."
x = txt.find("e", 5, 10)
print(x)  #output: 8
```

If the value is not found, the find() method returns -1, but the index() method will raise an exception

Replace all occurrence of the word "one":

```
txt = "one one was a race horse, two two was one too."
x = txt.replace("one", "three")
print(x)
```

Replace the two first occurrence of the word "one":

```
txt = "one one was a race horse, two two was one too."
x = txt.replace("one", "three", 2)
print(x)
```

Input statement in Python

```
name = input("Enter your name: ")  # String Input
age = int(input("Enter your age: ")) # Integer Input
marks = float(input("Enter your marks: ")) # Float Input
print("The name is:", name)
print("The age is:", age)
print("The marks is:", marks)
```

The **input()** function automatically converts the user input into string. We need to explicitly convert the input using the type casting.

Python program for addition of two numbers

```
a=int(input("enter the first integer value "))
b=int(input("enter the second integer value "))
c=a+b
print("sum of {0} and {1} numbers is {2}".format(a,b,c))
```

Python program to find squareroot

```
a=float(input("enter the number "))
b=a**0.5
print('square root of %f is %.3f'%(a,b))
```

Python program to find the area of triangle

```
a = float(input('Enter first side: '))
b = float(input('Enter second side: '))
c = float(input('Enter third side: '))

# calculate the semi-perimeter
s = (a + b + c) / 2

# calculate the area
area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
print('The area of the triangle is %0.2f' %area)
```

Python Lists

Lists are used to store multiple items in a single variable.

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.

Lists are created using square brackets:

Create a List:

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

print(thislist)

List Items

List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index [0], the second item has index [1] etc.

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

List Length

To determine how many items a list has, use the len() function:

Print the number of items in the list:

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))  #output: 3
```

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
print(list1)
print(list2)
print(list3)
#output: ['apple', 'banana', 'cherry']
[1, 5, 7, 9, 3]
[True, False, False]
```

```
list1 = ["abc", 34, True, 40, "male"]
print(list1) #output: ['abc', 34, True, 40, 'male']
```

List items are indexed and you can access them by referring to the index number:
Print the second item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1]) #output: banana
```

Negative indexing means start from the end
-1 refers to the last item, -2 refers to the second last item etc.
Example
Print the last item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])   #output: cherry
```

Return the third, fourth, and fifth item:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])
```

Append Items

To add an item to the end of the list, use the append() method:

Using the append() method to append an item:

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

Insert Items

To insert a list item at a specified index, use the insert() method.

The insert() method inserts an item at the specified index:

Insert an item as the second position:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

Extend List

To append elements from another list to the current list, use the extend() method.

Add the elements of tropical to thislist:

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)
```

Python - Remove List Items

Remove Specified Item

The **remove()** method removes the specified item. Remove "banana":

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

Remove Specified Index

The **pop()** method removes the specified index. Remove the second item:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```

If you do not specify the index, the pop() method removes the last item.

Clear the List

The clear() method empties the list. The list still remains, but it has no content.

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

Sort List Alphanumerically

List objects have a sort() method that will sort the list alphanumerically, ascending, by default:

Sort the list alphabetically:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
```

Sort Descending

To sort descending, use the keyword argument reverse = True:

Sort the list descending:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)
```

Python - Copy Lists

Make a copy of a list with the copy() method:

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

Make a copy of a list with the list() method:

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

Python - Join Lists

There are several ways to join, or concatenate, two or more lists in Python.

One of the easiest ways are by using the + operator.

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list3 = list1 + list2
print(list3)
```

Another way to join two lists is by appending all the items from list2 into list1, one by one:

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
for x in list2:
  list1.append(x)
print(list1)
```

Passing a List as an Argument

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

Python Tuples

Tuples are used to store multiple items in a single variable.

Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.

A tuple is a collection which is ordered and unchangeable.

Tuples are written with round brackets.

List is a collection which is ordered and changeable. Allows duplicate members.

Tuple is a collection which is ordered and unchangeable. Allows duplicate members.

Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

Tuple items are ordered, unchangeable, and allow duplicate values.

Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

Ordered

When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

Unchangeable

Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

Allow Duplicates

Since tuples are indexed, they can have items with the same value.

Print the number of items in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

One item tuple, remember the comma:

```
thistuple = ("apple",)
print(type(thistuple))
```

#NOT a tuple

```
thistuple = ("apple")
print(type(thistuple))
```

A tuple can contain different data types:

A tuple with strings, integers and boolean values:

```
tuple1 = ("abc", 34, True, 40, "male")
```

Access Tuple Items

You can access tuple items by referring to the index number, inside square brackets:

Print the second item in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Negative Indexing

Negative indexing means start from the end.

-1 refers to the last item, -2 refers to the second last item etc.

Print the last item of the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new tuple with the specified items.

Return the third, fourth, and fifth item:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")

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

Python - Update Tuples

Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created.
Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Convert the tuple into a list to be able to change it:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

Add Items

Since tuples are immutable, they do not have a build-in append() method, but there are other ways to add items to a tuple.

1. Convert into a list: Just like the workaround for changing a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

Convert the tuple into a list, add "orange", and convert it back into a tuple:

```
thistuple = ("apple", "banana", "cherry")

y = list(thistuple)

y.append("orange")

thistuple = tuple(y)
```

Add tuple to a tuple. You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:

Create a new tuple with the value "orange", and add that tuple:

```
thistuple = ("apple", "banana", "cherry")

y = ("orange",)

thistuple = thistuple+y


print(thistuple)
```

Unpacking a Tuple

When we create a tuple, we normally assign values to it. This is called "packing" a tuple:

Packing a tuple:

```
fruits = ("apple", "banana", "cherry")
```

But, in Python, we are also allowed to extract the values back into variables. This is called "unpacking":

Unpacking a tuple:

```
fruits = ("apple", "banana", "cherry")
(green, yellow, red) = fruits
print(green)
print(yellow)
print(red)
```

Python - Loop Tuples

Iterate through the items and print the values:

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

Print all items by referring to their index number:

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

Join Two Tuples

To join two or more tuples you can use the + operator:

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)
```

Multiply Tuples

If you want to multiply the content of a tuple a given number of times, you can use the * operator:

Multiply the fruits tuple by 2:

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2
print(mytuple)
```

Python Classes and Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword class:

Create a class named MyClass, with a property named x:

Create an object named p1, and print the value of x:

```
class MyClass:
  x = 5

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

The __init__() Function

To understand the meaning of classes we have to understand the built-in __init__() function.

All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Example

Create a class named Person, use the __init__() function to assign values for name and age:

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

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

Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.
Parent class is the class being inherited from, also called base class.
Child class is the class that inherits from another class, also called derived class.
Create a Parent Class
Any class can be a parent class, so the syntax is the same as creating any other class:
Example
Create a class named Person, with firstname and lastname properties, and a printname method:

```
class Person:
        def __init__(self, fname, lname):
                self.firstname = fname
                self.lastname = lname

        def printname(self):
                print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:
x = Person("John", "Doe")
x.printname()
```

Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

Example

Create a class named Student, which will inherit the properties and methods from the Person class:

```
class Student(Person):
  pass
```

Use the pass keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

Example

Use the Student class to create an object, and then execute the printname method:

```
x = Student("Mike", "Olsen")
x.printname()
```

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  pass

x = Student("Mike", "Olsen")
x.printname()
```

So far we have created a child class that inherits the properties and methods from its parent.

We want to add the __init__() function to the child class (instead of the pass keyword).

```
class Student(Person):
  def __init__(self, fname, lname):
    #add properties etc.
```

To keep the inheritance of the parent's __init__() function, add a call to the parent's __init__() function:

Example

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

Python also has a super() function that will make the child class inherit all the methods and properties from its parent:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

Add a property called graduationyear to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

Add a year parameter, and pass the correct year when creating objects:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
```

Add a method called welcome to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

```
class Person:
        def __init__(self, fname, lname):
                self.firstname = fname
                 self.lastname = lname

        def printname(self):
                print(self.firstname, self.lastname)

class Student(Person):
        def __init__(self, fname, lname, year):
                 super().__init__(fname, lname)
                 self.graduationyear = year

        def welcome(self):
                print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)

x = Student("Mike", "Olsen", 2019)
x.welcome()
```