COMPUTER VISION

LAB 1 :Programming in Python

MUNADI SIAL

SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY

Lab Overview

In this Lab, you will learn the following:

- **Python:** Programming language used widely in Computer Visio
- **IDE:VS Code or PyCharm** programming environment
- **OpenCV:** Library for image processing and computer vision

Python Language

- Python is an interpreted, open-source language with an emphasis on code readability and has a very large community support
- Consider the following comparison between C++ and Python codes:

Python	**C++**

`a = 10`
`b = "M"`
`c = 7.6`
`print(a,b,c)`
	`int a = 10;`
	`char b = "M";`
	`float c = 7.6;`
	`cout << a << " " << b << " " << c << endl;`

- The output of both programs is the same:

```
>> 10 M 7.6
```

- No semicolons are needed to end each line in Python
- It is not necessary to declare the data type of variable in Python
- The print() statement creates spaces and new line automatically

Python Scripts

- Python scripts are files that end in **.py** format and contain the code
- To write a script, we will use **SublimeText** which is a text editor that allows syntax highlighting for python

- Let's write a simple python script and execute it via the terminal:

```
a = 10
b = "M"
c = 7.6
print(a,b,c)
```

- Once you have typed the above code, save the script as **test.py**
- Now open the terminal, go the directory (with cd) where you saved the script and execute it with the following command:

```
python test.py
```

Variables in Python

- To add a comment in code, use # in Python
- The data type of a variable can change in Python:

```
a = 10          # this will work
a = 5.5         # this will work
a = "house"     # this will work
```

- The equivalent code in C++ will give an error:

```
int a = 10;     // this will work
a = 5.5;        // error
a = "house";    // error
```

Data Types in Python

- Python contains several data types, some of which are given below:

```
x_string = "Hello World"
x_integer = 25
x_float = 25.1
x_boolean = True
x_complex = 2j
x_list = [1, 2, 3, 4, 5]
x_tuple = (1, 2, 3, 4, 5)
x_set = {1, 2, 3, 4, 5}
x_dictionary = {"name":"Ali", "age":31}
```

Arithmetic Operators

- The following code shows the arithmetic operators in Python:

```
a = 10
b = 3

add = a + b
sub = a - b
mul = a*b
div = a/b
quotient = a//b
remainder = a%b
power = a**b

print(add, sub, mul, div, quotient, remainder, power)
 >> 13 7 30 3.3333333333333335 3 1 1000
```

Logical Operators

- The following code shows the logical operators in Python:

```
a = True
b = False

print(a and b)
>> False

print(a or b)
>> True

print(not a)
>> False

print(not(a and b))
>> True

print(not(a or b))
>> False

print(not b)
>> True
```

Strings

- Python supports the string data type which is an array of characters

```
h = "Manipulator"
print(h)
```

Output:

```
Manipulator
```

- You can get individual characters with square brackets

```
print(h[0])
print(h[1])
print(h[8])
```

Output:

```
M
a
p
```

- You can get the number of characters with the len() function

```
g = len(h)
print(g)
```

Output:

```
11
```

Strings

- You can concatenate strings easily in python

```
c = "Computer"
d = "Vision"
e = c + d
print(e)
```

Output:

```
ComputerVision
```

```
f = c + " " + d
print(f)
```

Output:

```
Computer Vision
```

- You can check if a character is present in the string with the "in" keyword

```
print("t" in c)
print("s" in c)
```

Output:

```
True
False
```

User Input

- To get input from user, we use the input() function

```
v1 = input("Enter the first number: ")
print(v1)

v2 = input("Enter the second number: ")
print(v2)
```

Output:

```
Enter the first number: 200
200
Enter the second number: 100
100
```

User Input

- The input() function returns a string data which is stored in the variable
- If we add the previous two numbers, they will concatenate because they are strings, not numbers

```
print(v1 + v2)
```

Output: `200100`

- To compute the numeric sum, we need to convert the string data type to an int or float data type

```
v1 = int(v1)
v2 = int(v2)
print(v1 + v2)
```

Output: `300`

If…Else

- Python supports the IF…ELSE conditional statements which choose to execute statements depending if a condition is true or false
- The syntax for a IF statement is given as:

```
if <condition>:
   <statement_1>
   <statement_2>
```

- An example is given below:

```
a = 4
b = 3
if a > b:
   print("a is greater than b")
```

- The <condition> can be any of the following:

```
a==b     a!=b     a < b     a > b     a <= b     a >= b
```

If…Else

- The **else** keyword contains statements that execute when the condition is not met

```
a = 4
b = 3
if a > b:
   print("a is greater than b")
else:
   print("b is greater than a")
```

- Note that the colon symbol is part of the syntax
- To contain statements, they are indented with spaces
- The indents are mandatory in Python to define the block of statements

If…Else

- The **elif** keyword nests an if command inside an else statement
- It is the equivalent of **else if**

```
a = 4
b = 3
if a > b:
   print("a is greater than b")
elif (a < b):
   print("a is less than b")
else
   print("a is equal to b")
```

- Only one of the three print statements will be executed

Loops

- Python has two types of loops: WHILE and FOR
- The **while** loop has the following syntax:

```
while <condition>:
    <statement_1>
    <statement_2>
```

- The while loop checks the condition. If the condition is true, the statements are executed. After executing, the condition is rechecked. If it is true, it is executed again. This continues until the condition becomes false at which point the loop ends
- An example is given below

```
a = 0
while a < 5:
    print(a)
    a = a + 1
```

Output:

```
0
1
2
3
4
```

Loops

- The **break** keyword terminates the loop in which it is placed:

```
a = 0
b = 3
while a < 5:
    print(a)
    a = a + 1
    if a == b:
        break
```

Output:

```
0
1
2
```

Loops

- The **continue** keyword skips the current iteration of the loop:

```
a = 0
b = 3
while a < 5:
    a = a + 1
    if a == b:
        continue
    print(a)
```

Output:

```
1
2
4
5
```

Loops

- The **for** loop goes through a sequence of items (iterable object)
- We use the range() to create a sequence of numbers

```
for i in range(0,5):
    print(i)
```

Output:

```
0
1
2
3
4
```

```
for items in range(0,10,2):
    print(items)
```

Output:

```
0
2
4
6
8
```

Loops

- The **for** loop can also go through a sequence of characters
- The iterable object is a string variable in this case

```
for i in "PYTHON":
    print(i)
```

Output:

```
P
Y
T
H
O
N
```

Functions

- We can define and call functions in python

```
def my_function(a, b):                    # Function Definition
    out = a + b
    print(a, "+", b, "=", out)
    return out

value = my_function(3,4)                  # Function Call
print("value =", value)
```

Output:

```
3 + 4 = 7
value = 7
```

Data Structures

Python offers 4 built-in data types for storing collections of data:

1. **Lists:** A collection of data which is ordered, changeable and allows duplicate members

2. **Tuples:** A collection of data which is ordered, unchangeable and allows duplicate members

3. **Sets:** A collection of data which is unordered, unchangeable and does not allow duplicate members

4. **Dictionaries:** A collection of data which is stored in *key:value pairs* and is ordered, changeable and does not allow duplicate members (Dictionaries were unordered before Python 3.7)

Data Structures

Examples of the collection data are given below. Notice the bracket types.

```
x_list = [1, 2, 3, 4, 5]

x_tuple = (1, 2, 3, 4, 5)

x_set = {1, 2, 3, 4, 5}

x_dictionary = {"name" : "Ali"
                "age" : 31
              "result" : False}
```

Lists

- A List is a collection of data that is
  - Ordered
  - Changeable (Mutable)
  - Allows Duplicate Members

- A List is a sequence of values (called items or elements)
- Whereas a string is a sequence of characters, a list can be a sequence of any data type

- Lists are among the commonly used data types in python
- Lists are used extensively in ROS such as in reading laser data
- A List works similar to the arrays in C++

Lists - Creation

- To create a list, the simplest way is to use square brackets to enclose the items and use commas to separate the items

```
x = [1, 2, 3, 4, 5]

print(x)
```

Output:

```
[1, 2, 3, 4, 5]
```

Lists

- Lists can hold different types of data:

```
a = [11, 2, 93, 401, 560]
b = [1.5, 6.6, 7.3, 8.9]
c = ["apple", "banana", "cherry"]
d = [True, True, False, True, False]
e = []
f = list("ROBOT")

print(a)
print(b)
print(c)
print(d)
print(e)
print(f)
```

Output:

```
[11, 2, 93, 401, 560]
[1.5, 6.6, 7.3, 8.9]
['apple', 'banana', 'cherry']
[True, True, False, True, False]
[]
['R', 'O', 'B', 'O', 'T']
```

Lists

- The same list can also hold different types of data:

```
g = [31.5, "Robotics", 4, True, 9]

print(g)
```

Output:

```
[31.5, 'Robotics', 4, True, 9]
```

Lists

- Lists can nest other lists:

```
h = ["wheels", 35, 5.2, [10, 20], 50]

print(h)
print(h[3])
print(h[3][0])
print(h[3][1])
```

Output:

```
['wheels', 35, 5.2, [10, 20], 50]
[10, 20]
10
20
```

Lists

- Lists allow duplicates of data:

```
j = ["apple", "banana", "cherry", "apple", "watermelon"]

print(j)
```

Output:

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

Accessing Items in a List

- To access individual items in a list, square brackets are used which contain the index number of the item
- The index of the first item starts at zero
- The index of the last item is (number of items – 1)

0	1	2	3	4	5	6	7

```
my_list = [14, 20, 93, 41, 56, 77, 38, 62]

print(my_list[0])
print(my_list[1])
print(my_list[5])
print(my_list[7])
```

Output:

```
14
20
77
62
```

Changing Items in a List

- The value of an item in a list can be changed by using its index

0	1	2	3	4	5	6	7

```
my_list = [14, 20, 93, 41, 56, 77, 38, 62]
print(my_list)

my_list[2] = 37
print(my_list)
```

Output:

```
[14, 20, 93, 41, 56, 77, 38, 62]
[14, 20, 37, 41, 56, 77, 38, 62]
```

Accessing Items with Negative Index

- Items can also be indexed from the end. This is done by using negative numbers for the index
- The negative indexing starts from -1 (not zero)

	0	1	2	3	4	5	6	7
my_list =	14	20	93	41	56	77	38	62
	-8	-7	-6	-5	-4	-3	-2	--1

```
print(my_list[-1])
print(my_list[-2])
print(my_list[-3])
print(my_list[-8])
```

Output:

```
62
38
77
14
```

Changing Items with Negative Index

- Negative indexing can also be used to change the value of items

-8	-7	-6	-5	-4	-3	-2	--1

```
my_list = [14, 20, 93, 41, 56, 77, 38, 62]
print(my_list)

my_list[-5] = 500
print(my_list)
```

Output:

```
[14, 20, 93, 41, 56, 77, 38, 62]
[14, 20, 93, 500, 56, 77, 38, 62]
```

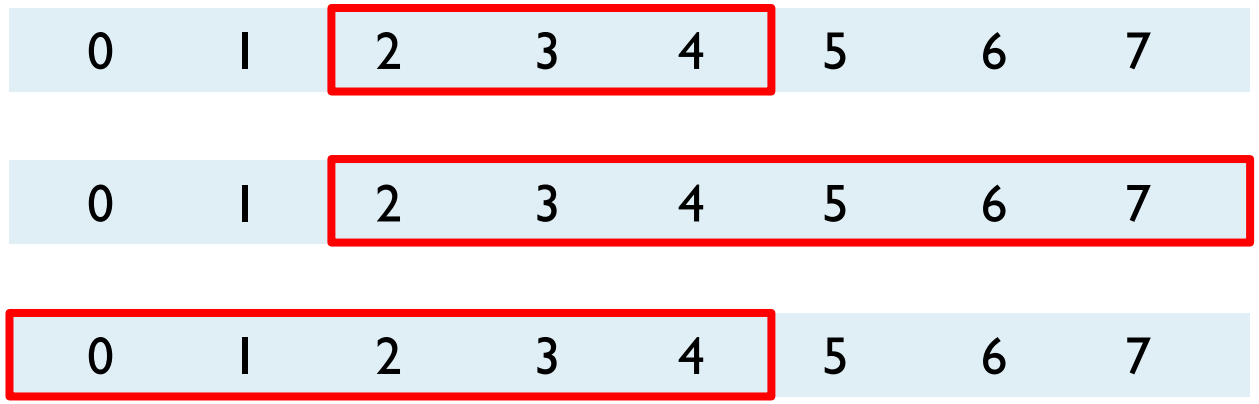
Accessing Range of Items

- The access a range of items, the colon (:) is used (slice operation)
- Note the post-colon number is NOT included in the index

0	1	2	3	4	5	6	7

```
my_list = [14, 20, 93, 41, 56, 77, 38, 62]

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



Output:

```
[93, 41, 56]
[93, 41, 56, 77, 38, 62]
[14, 20, 93, 41, 56]
```

Changing Range of Items

- The range of items, accessed with the colon (:), can be changed


```
my_list = ["apple", "banana", "cherry", "orange", "mango"]
print( my_list)

my_list[2:4] = ["grapes", "melon"]
print( my_list)
```

Output:

```
['apple', 'banana', 'cherry', 'orange', 'mango']
['apple', 'banana', 'grapes', 'melon', 'mango']
```

Changing Range of Items

- If you insert more items than you replace, the list will increase in length
- In the example below, banana and cherry are replaced by 3 items


```
my_list = ["apple", "banana", "cherry", "orange“]
print( my_list)

my_list[1:3] = ["carrot", "potato", "turnip"]
print( my_list)
```

Output:

```
['apple', 'banana', 'cherry', 'orange']
['apple', 'carrot', 'potato', 'turnip', 'orange']
```

Changing Range of Items

- If you insert less items than you replace, the list will decrease in length
- In the example below, banana and cherry are replaced by 1 item


```
my_list = ["apple", "banana", "cherry", "orange"]
print( my_list)

my_list[1:3] = ["strawberry"]
print( my_list)
```

Output:

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

Checking Items in List

- The **in** and **not in** keywords can used to check for items in a list:

```
my_list = ["apple", "banana", "cherry", "orange"]

print( "orange" in my_list )
print( "mango" in my_list )

print( "orange" not in my_list )
print( "mango" not in my_list )
```

Output:

```
True
False
False
True
```

Looping a List

- The **for** loop can used to iterate through the items of a list
- The number of times the loop executes is equal to the number of items
- The iterator (fruit) takes the value of each item of every iteration

```
my_list = ["apple", "banana", "cherry", "orange"]

for fruit in my_list:
    print(fruit)
```

Output:

```
apple
banana
cherry
orange
```

Looping a List

- If the indices are needed in the loop, then the **range** and **length** functions can be used

```
my_list = [1,2,3,4,5]

for i in range(len(my_list)):
    my_list[i] = 2 * my_list[i]

print(my_list)
```

Output:

```
[2, 4, 6, 8, 10]
```

List Functions

- List functions (also called methods) are useful functions that can be used with a list
- Python provides the following list functions:
    - Append()
    - Insert()
    - Extend()
    - Remove()
    - Pop()
    - Count()
    - Index()
    - Reverse()

Functions to Add Items

Functions to Remove Items

Functions to Add Items

- To add items, Python provides 3 list functions:
  - Append
  - Insert
  - Extend

- The **append** function adds an item to the end of a list:

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

listA.append("mango")
print(listA)
```

Output:

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

Functions to Add Items

- The **insert** function adds an item to a certain index
- The items are readjusted and the list size increases

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

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

Output:

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

Functions to Add Items

- The **extend** function adds a list to another list

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

listB = ["carrot", "onion"]
listA.extend(listB)
print(listA)
```

- Another way to do this is to use the addition operation to concatenate the lists (`listA = listA + listB`)

Output:

```
['apple', 'banana', 'cherry']
['apple', 'banana', 'cherry', 'carrot', 'onion']
```

Functions to Remove Items

- To remove items, Python provides 2 list functions:
  - Pop
  - Remove

- The **remove** function can remove a specified item:

```
listA = ["apple","banana","cherry", "mango"]
print(listA)

listA.remove("cherry")
print(listA)
```

Output:

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

Functions to Remove Items

- The **pop** function can remove an item from a specific index:

```
listB = ["apple","banana","cherry", "mango"]
print(listB)

listB.pop(1)
print(listB)
```

Output:

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

Other List Functions

- The **count** function returns the total number of a specified item

```
listC = [1,1,3,4,3,5,7,4,3,7,8,3,2]
number = listC.count(3)
print(number)
```

Output:

```
4
```

- The **index** function returns the index of the first occurrence of a specified item

```
index_val = listC.index(3)
print(index_val)
```

Output:

```
2
```

Other List Functions

- The **reverse** function reverses the order of the items in the list

```
listA = ["apple","banana","cherry", "mango"]
listA.reverse()
print(listA)
```

Output:

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

2-D Lists

- The following is a 2-D list (a list of lists)
- The element of such lists can be accessed by multiple indexes
- The first index selects among the "items" of the outer list
- The second index selects among the items of the inner sub-list
- Remember that indexing in python starts from zero

```
my_2d_list = [[11, 22, 33, 44, 55],
              [71, 27, 42, 99, 58],
              [61, 62, 63, 64, 65],
              [83, 85, 19, 24, 21]]

print( my_2d_list[2][3] )
```

Output: **64**

Strings - Review

- Python supports the string data type which is an array of characters

```
h = "Manipulator"
print(h)
```

Output:

```
Manipulator
```

- You can get individual characters with square brackets

```
print(h[0])
print(h[1])
print(h[8])
```

Output:

```
M
a
p
```

- You can get the number of characters with the len() function

```
g = len(h)
print(g)
```

Output:

```
11
```

Strings - Review

- You can concatenate strings easily in python

```
c = "Computer"
d = "Vision"
e = c + d
print(e)

f = c + " " + d
print(f)
```

Output:

```
ComputerVision
```

Output:

```
Computer Vision
```

- You can check if a character is present in the string with the "in" keyword

```
print("t" in c)
print("s" in c)
```

Output:

```
True
False
```

Strings - Review

- The **for** loop can go through a sequence of characters
- The iterable object will be a string variable in this case

```
for i in "PYTHON":
    print(i)
```

Output:

```
P
Y
T
H
O
N
```

Dictionaries

- A dictionary is another data structure
- A dictionary is like a list but somewhat more general
- In a list, index positions have to be integers; in a dictionary, index positions can be (almost) any type

- A Dictionary is a collection of data that is
  - Ordered (as of Python 3.7)
  - Changeable (Mutable)
  - Does NOT Allow Duplicate Members

Dictionaries - Creation

- A dictionary is a mapping of a set of indices (called keys) to a set of values
- Each key maps to a value
- Each key-value pair is an item of a dictionary
- To create a dictionary, key-value pairs are enclosed in braces

```
eng2span = {'one': 'uno',
            'two': 'dos',
          'three': 'tres'}

print(eng2span)
```

Output:

```
{'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Dictionaries - Accessing

- The following dictionary maps English (keys) to Spanish (values)

```
eng2span = {'one': 'uno',
            'two': 'dos',
          'three': 'tres'}
```

- The keys are used as indexes to look up the corresponding values

```
print(eng2span['one'])
print(eng2span['two'])
print(eng2span['three'])
```

Output:

```
uno
dos
tres
```

Dictionaries - Accessing

- The following dictionary maps English (keys) to Numbers (values)

```
eng2num = {'one': 1,
           'two': 2,
         'three': 3}
```

- The keys are used as indexes to look up the corresponding values

```
print(eng2num['one'])
print(eng2num['two'])
print(eng2num['three'])
```

Output:

```
1
2
3
```

Dictionaries – Changing Value

- Consider the following dictionary which maps prices

```
prices = {'burger': 250,
          'sandwich': 150,
            'pizza': 400}
```

- The value can be changed by using its key:

```
prices['sandwich'] = 180
print(prices)
```

Output:

```
{'burger': 250, 'sandwich': 180, 'pizza': 400}
```

Dictionaries

- The following example illustrates a use of dictionaries:

```
commands = { "moveForward" : [1, 0],
             "moveBack"    : [-1, 0],
             "turnRight"   : [0, 1],
             "turnLeft"    : [0, -1],
             "stop"        : [0, 0]}
```

- Each key is a command for a robot's movement
- Note that the value in this case is a list of two elements
- The first element of the list is the linear motion
- The second element of the list is the angular motion