

Programming Foundations in Python

Prepared by Dr. Hamada I. AbdulWakel

Revision Lecture



Introduction to Computer Programming and Python

Lecture 1

Prepared by
Dr. Hamada I. AbdulWakel

Lecture

01

COMPUTER SCIENCE

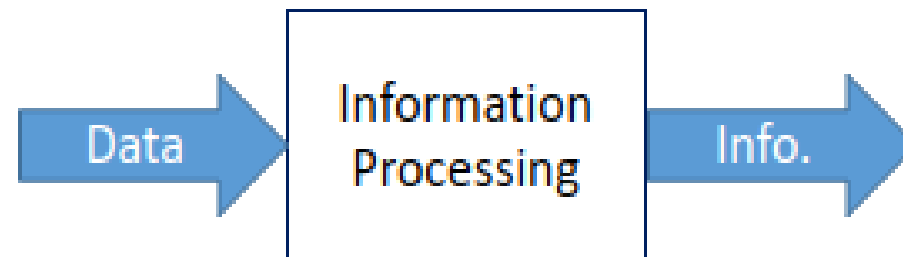
- **Computer Science (CS):** the study of computation and information.
 - Solving problems is the core of CS.
- **Problem-solving:** the process of transforming the description of a **problem** into a **solution** using the problem-solving tools, strategies, and techniques.

Lecture

01

INFORMATION PROCESSING

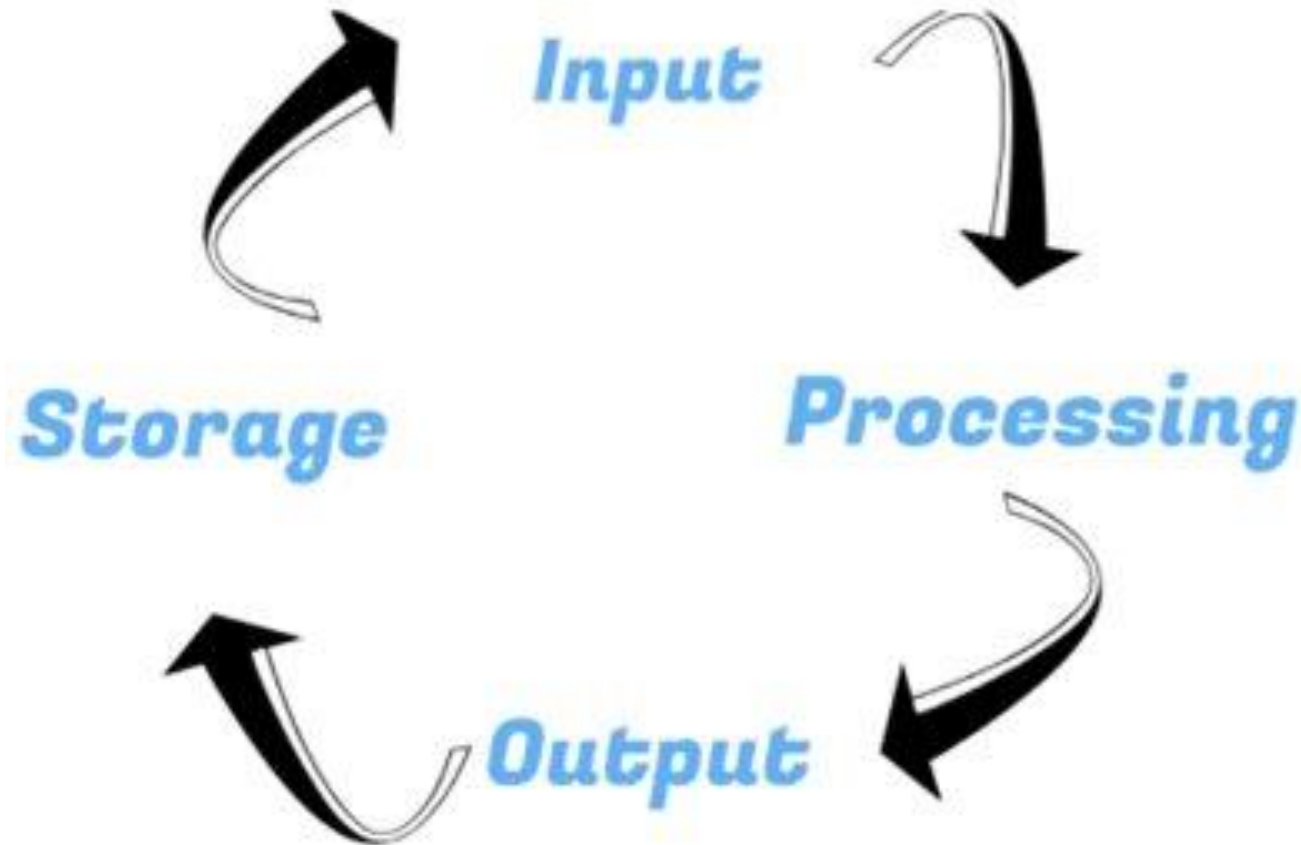
- **Data:** the raw, unorganized facts that can be in the form of text, graphics, audio, or video.
- **Information:** the processed data with a meaningful form.
- **Information processing:** the process of converting raw data into information.



Lecture

01

INFORMATION PROCESSING CYCLE



Lecture

01

INFORMATION PROCESSING CYCLE

Cont'd

- **Input:** the stage where the computer is fed with the data from its user by the use of the input devices.
- **Processing:** after the unorganized facts (data) have been sent (input) into the computer's system, the computer processes and organize them to make them meaningful.

Lecture

01

INFORMATION PROCESSING CYCLE

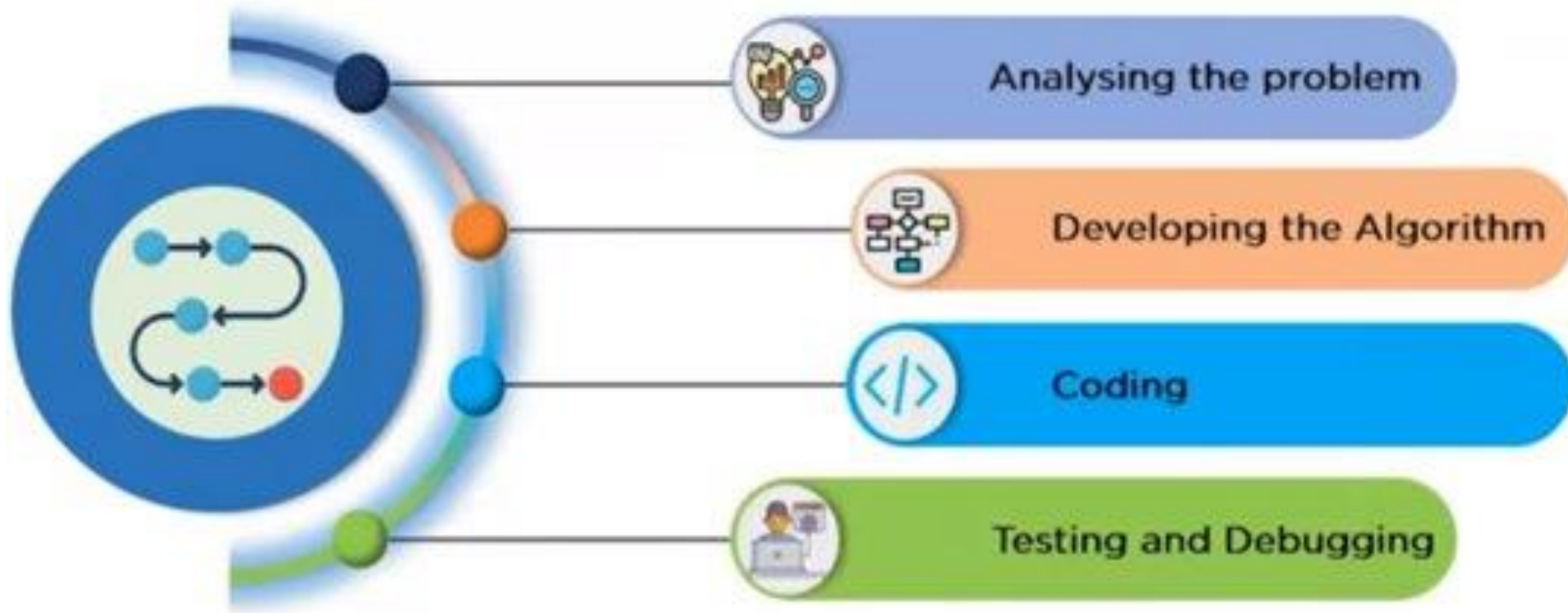
Cont'd

- **Output:** the step where the computer brings out information in the form understood to its users and displays it using the output devices.
- **Storage:** the organized data (information) is kept for future use. It can be stored on the internal hard disk or external on storage media.

Lecture

01

PROBLEM-SOLVING STEPS



Lecture

01

PROBLEM-SOLVING STEPS

Cont'd

- **Analyzing the problem (Step 1):**
 - We can figure out the inputs and outputs to be carried out.
- **Developing the algorithm (Step 2):**
 - It is required to decide a solution before writing a program.

Lecture

01

PROBLEM-SOLVING STEPS

Cont'd

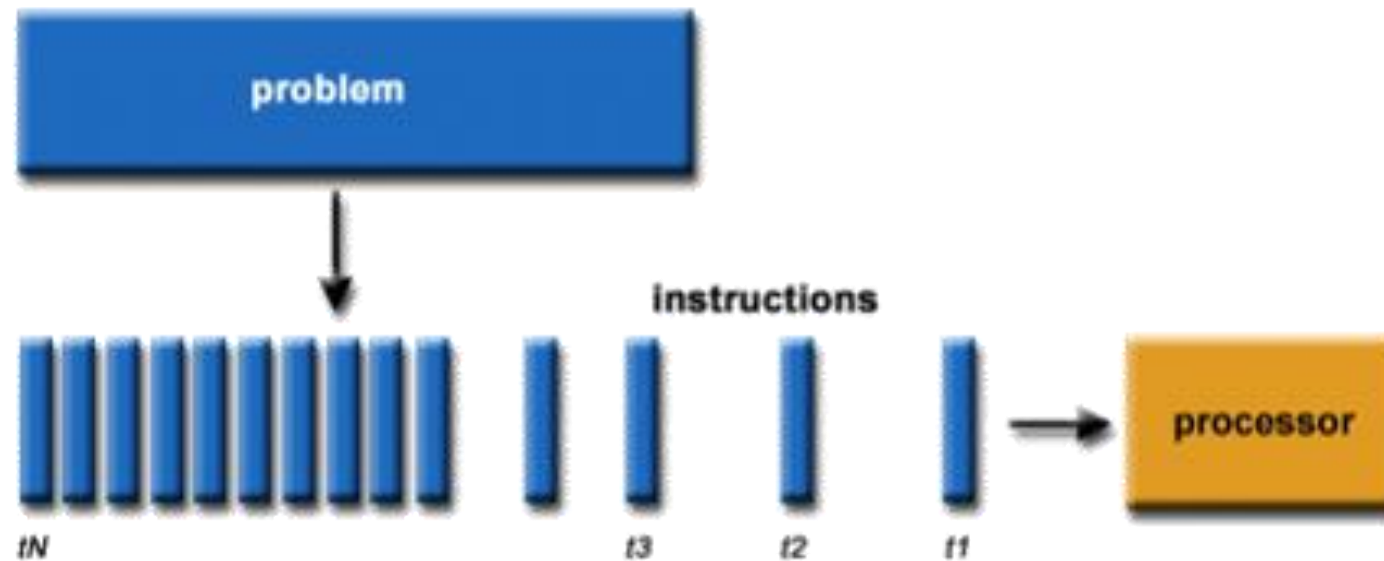
- **Coding (Step 3):**
 - We must convert the decided algorithm into a code.
- **Testing and debugging (Step 4):**
 - The developed code undergoes several rigorous tests.

Lecture

01

ALGORITHM

- The sequence of steps that describes each of the computational processes.



Lecture

01

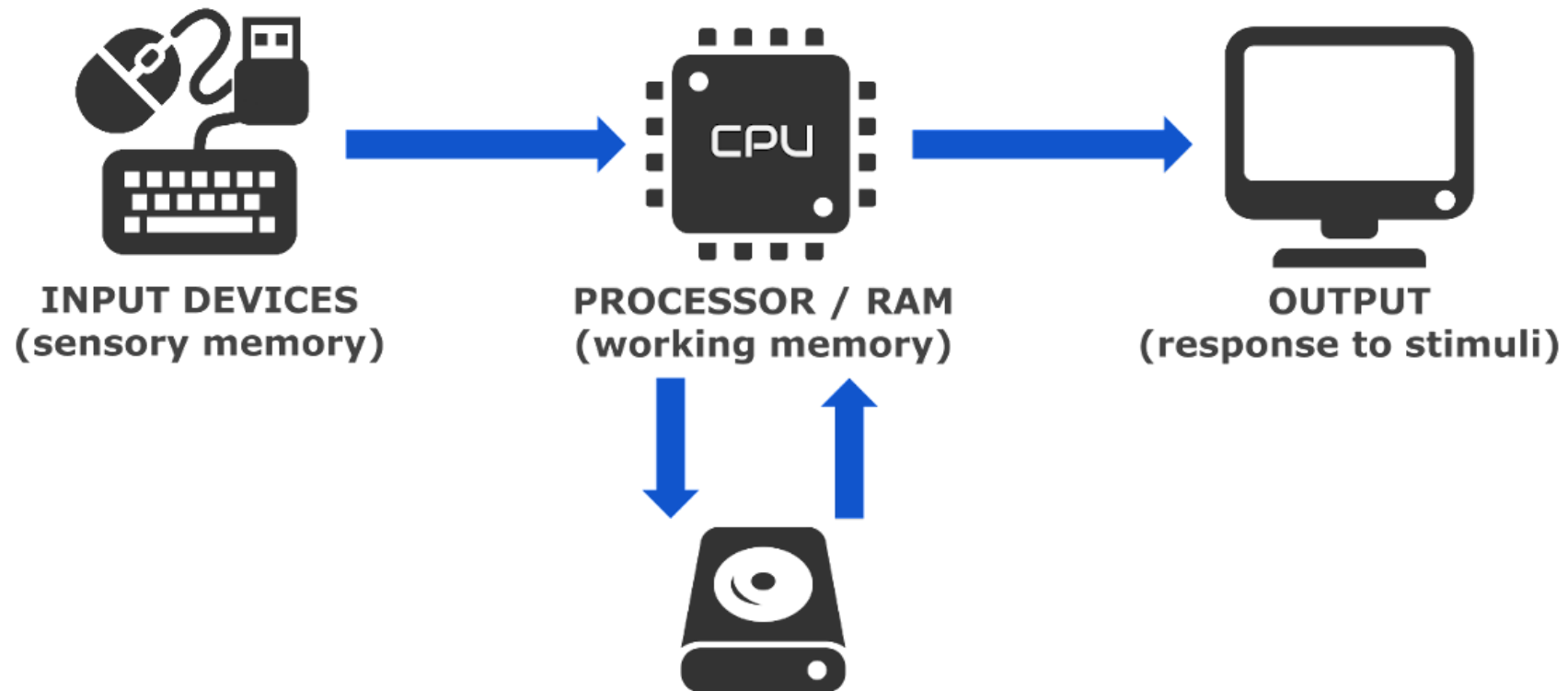
ALGORITHM FEATURES

- It must consist of a **finite number** of instructions.
- Each algorithm instruction is well defined and **not ambiguous**.
- It should take **zero or more input**.
- It should **terminate** after a finite time and produce at least **one output**.
- It should be **deterministic** means giving the same output for the same input case.

Lecture

01

HARDWARE & SOFTWARE COLLABORATION



Lecture

01

HARDWARE & SOFTWARE COLLABORATION

Cont'd

- A program (i.e., an algorithm) stored in computer memory must be represented in binary digits, or **machine code**.
- Each **loader** takes a set of machine language instructions as input and loads them into the appropriate memory locations.
- In CPU units performs the required computational, logical, and controlling the loaded instructions.

Lecture

01

PROGRAMMING & PROGRAMMING LANGUAGES

- **Programming:** the design of software using a set of basic instructions.
- **Programming language:** abstract hardware instructions into basic statements, which are much easier to understand and handle.
- Using a programming language, one can write complex software, achieving tailored functionalities by combination of basic statements.

Lecture

01

FIRST PROGRAM IN PYTHON

- In order to write Python scripts (codes) in your machine (**offline**), we **must** install Python language.
- Writing Python scripts using any **online** editors does **not require** to install Python language on your machine.

Lecture

01

FIRST PROGRAM IN PYTHON

Cont'd

- Follow these steps in writing codes:
 - Create a code folder in any location on your machine.
 - Create a Python script file.
 - Python has **.py** extension for its script files.
 - Open the script file in your IDE.
 - Write your code.

```
print("Hello World from Python")
```

Lecture

01

PYTHON FORMATING STYLE

- **Indentation:** the leading whitespace before any statement in Python.
- We can use even if **one space** for code indentation, the default is **four spaces**.
- **A code block:** a group of statements that have the same indentation level (have the same number of leading whitespaces).
- It used to make the code more **readability**.

Data Types & Input/Output Operations

Lecture 2

Prepared by
Dr. Hamada I. AbdulWakel

CASE STUDY: DESIGN INCOME TAX CALCULATOR

Cont'd

- An English-style algorithm is called **Pseudocode**.
 1. Input the gross income and number of dependents.
 2. Compute the taxable income using the formula
 3. $\text{Taxable income} = \text{gross income} - 10000 - (2000 * \text{number of dependent})$
 4. Compute the income tax using the formula
 5. $\text{Tax} = \text{taxable income} * 0.20$
 6. Print the tax

Lecture

02

CASE STUDY: TESTING INCOME TAX CALCULATOR

Cont'd

- Error types:
 - **Syntax error:** occurred due to the mistakes while typing or do not follow the syntax of the specified programming language.
 - **Run-time error:** When the program is running, and it is not able to perform the operation.
 - **Logical error:** an error that leads to an undesired output.

Lecture

02

DISPLAYING OUTPUTS IN PYTHON

- Almost all programs have solutions that needed to be displayed to the user through outputting data from the program.
 - Use the print() function.
`help(print)`
`print("Hello", "Ahmed")`
 - Change the default values of print() function.
`print("Hello", "Ahmed", sep="000")`
`print("Hello", "Ahmed", end="1111")`

Lecture

02

DATA TYPES IN PYTHON

- Almost variable is a reserved memory location to store values.
- Python is a type-less language.

```
X = 10
```

```
X = "Hello world"
```

- The variable name refers to the memory location where its value store.
 - The id() function is used to get the memory location.

```
I = 10
```

```
print( id( I ) )
```

DATA TYPES IN PYTHON

Cont'd

- A variable name must begin with either a **letter** or an **underscore** (_).
- It contains any number of letters, digits, or other underscores.
- Variable names are **case sensitive**.

`X = 10``x = 10`

- Variable name could be unicode.

`س = 5`

DATA TYPES IN PYTHON

Cont'd

- The `type()` function is used to know variable type:

```
X = 10
```

```
print( type( X ) )
```

- Escape sequences are characters used to express special characters as literals.

ESCAPE SEQUENCE	MEANING
<code>\b</code>	Backspace
<code>\n</code>	Newline
<code>\t</code>	Horizontal tab
<code>\\</code>	The <code>\</code> character
<code>\'</code>	Single quotation mark
<code>\"</code>	Double quotation mark

Lecture

02

ARITHMETIC EXPRESSIONS
IN PYTHON

Cont'd

Name	Meaning	Example	Result
+	Addition	15+20	35
-	Subtraction	24.5-3.5	21.0
*	Multiplication	15*4	60
/	Float Division	4/5	0.8
//	Integer Division	4//5	0
**	Exponentiation	4**2	16
%	Remainder	27%4	3

ARITHMETIC EXPRESSIONS IN PYTHON

Cont'd

Operator	Name	Example	Equivalent
<code>+=</code>	Addition assignment	<code>x += 4</code>	<code>x = x + 4</code>
<code>-=</code>	Subtraction assignment	<code>x -= 2</code>	<code>x = x - 2</code>
<code>*=</code>	Multiplication assignment	<code>x *= 5</code>	<code>x = x * 5</code>
<code>/*=</code>	Division assignment	<code>x /= 5</code>	<code>x = x / 5</code>
<code>//*=</code>	Integer division assignment	<code>x //= 5</code>	<code>x = x // 5</code>
<code>%*=</code>	Remainder assignment	<code>x %= 5</code>	<code>x = x % 5</code>
<code>**=</code>	Exponent assignment	<code>x **= 5</code>	<code>x = x ** 5</code>

Lecture

02

ARITHMETIC EXPRESSIONS IN PYTHON

Cont'd

- Rule of **priority**, that is some operator has highest priority than others.
 1. `()`
 2. `**`
 3. Unary operator
 4. `*`, `/`, `//`, `%`
 5. `+`, `-`
- Rule of **associativity**, evaluate expression of similar priority from left to right.

Lecture

02

FUNCTIONS WITH NUMERIC DATA TYPES

- Numeric functions:

```
print( bin(50) )
```

```
print( oct(50) )
```

```
print( hex(50) )
```

```
print( int(0x32) )
```

```
print( abs(-5) )
```

```
print( pow(2, 3) )
```

```
print( max(4, 5) )
```

```
print( min(4, 5) )
```

```
print( round(3.7) )
```

Lecture

02

GETTING INPUTS IN PYTHON

- Almost all programs **interact with the user** through giving data to the program to process it and display other data.

- Use the `input()` function.

```
input( )
```

```
input( "Enter your name: " )
```

```
name = input( "Enter your name: " )
```

- Change the type of the value given from the user.

```
stdCount = int( input( "Enter the number of students: " ) )
```

```
stdAge = float( input( "Enter student age: " ) )
```

Control and Iteration Statements

Lecture 3

Prepared by
Dr. Hamada I. AbdulWakel

Lecture

03

BOOLEAN DATA TYPE AND COMPARISON OPERATORS

- Boolean data type consists of two values **TRUE** and **FALSE**.
- The **"is"** operator is used to test if two variables refer to the same value.
- The **"is not"** operator is used to compare if the objects are pointing to the same memory location or not.

COMPARISON OPERATOR	MEANING
==	Equals
!=	Not equals
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

Lecture

03

COMPOUND BOOLEAN EXPRESSIONS

- The logical operators:
 - The **not** operator.
 - The **and** operator.
 - The **or** operator.

A	not A
True	False
False	True

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

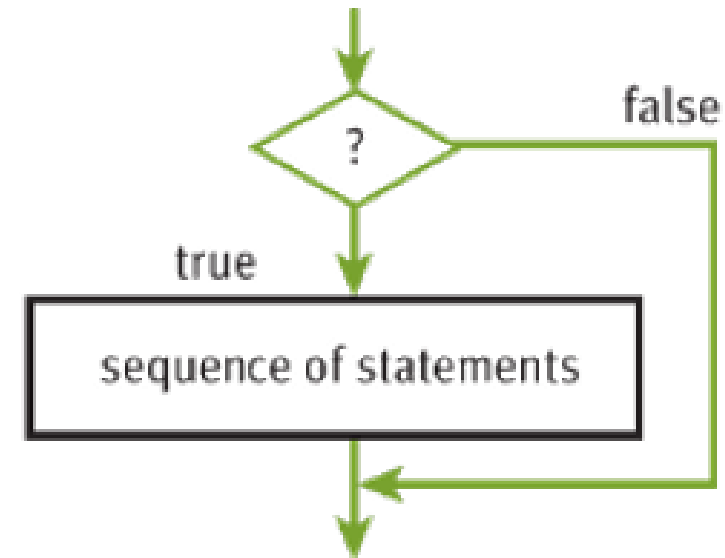
- A program's flow is the order in which the program's code executes.
- **The sequential flow:** a set of statements whose execution process happens in a sequence.
- **The conditional flow:** allows a program to test several conditions and execute instructions based on which condition is true.
- **The looping flow:** used to repeat a group (block) of programming instructions.

Lecture

03

THE CONDITIONAL FLOW
STATEMENTS

- The basic-selection statement (If statement).
 - Syntax:
if <condition> :
 <statement(s)>
 - Example:
 x = 5
 if x == 6 :
 print("Yes")



Lecture

03

THE CONDITIONAL FLOW
STATEMENTS

Cont'd

- The two-way selection statement (If...else statement).

- Syntax:

if <condition> :

 <true statement(s)>

else:

 <false statement(s)>

- Example:

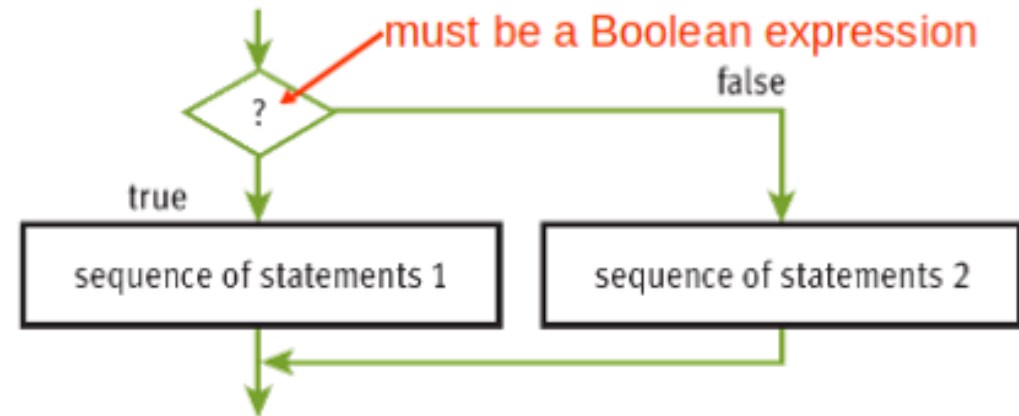
```
x = 5
```

```
if x == 6 :
```

```
    print( "Yes")
```

```
else:
```

```
    print("No")
```



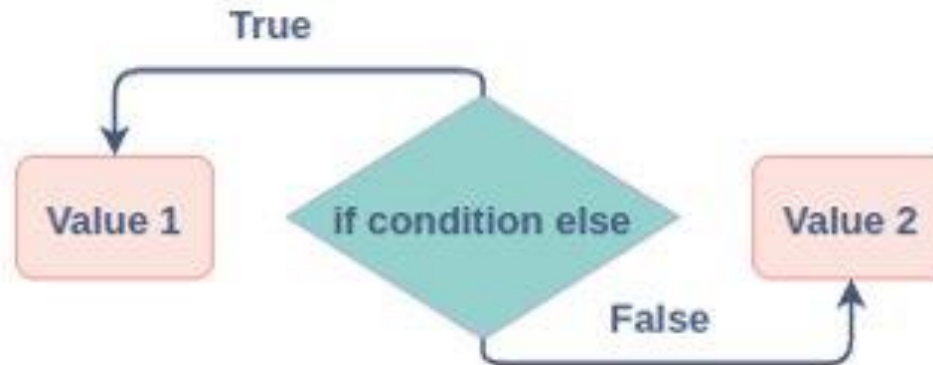
Lecture

03

THE CONDITIONAL FLOW STATEMENTS

Cont'd

- The ternary operator statement (If...else statement).
 - Syntax:
`<variable> = <true statement> if <condition> else <false statement>`



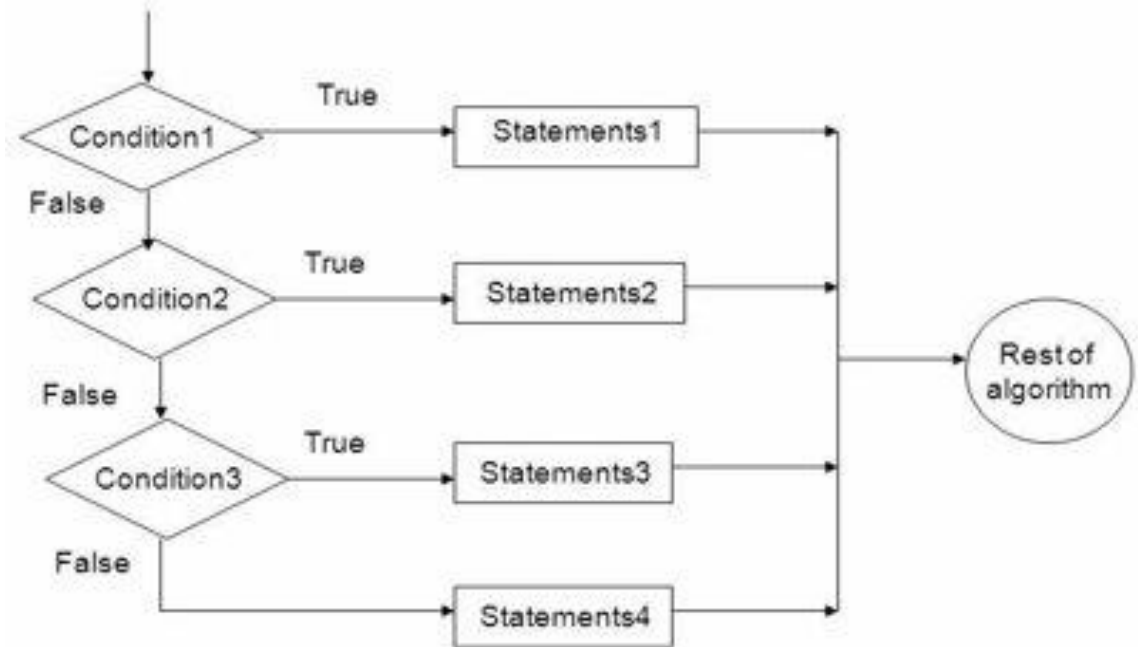
THE CONDITIONAL FLOW STATEMENTS

Cont'd

- The multi-way selection statement (If...elif statement).

- Syntax:

```
if <condition_1> :  
    <true cond_1 statement(s)>  
elif <condition_2> :  
    <true cond_2 statement(s)>  
else:  
    <otherwise statement(s)>
```



Lecture

03

THE CONDITIONAL FLOW
STATEMENTS

Cont'd

- The nested selection statement (If...if statement).

- Syntax:

if <condition1>:

if <condition2>:

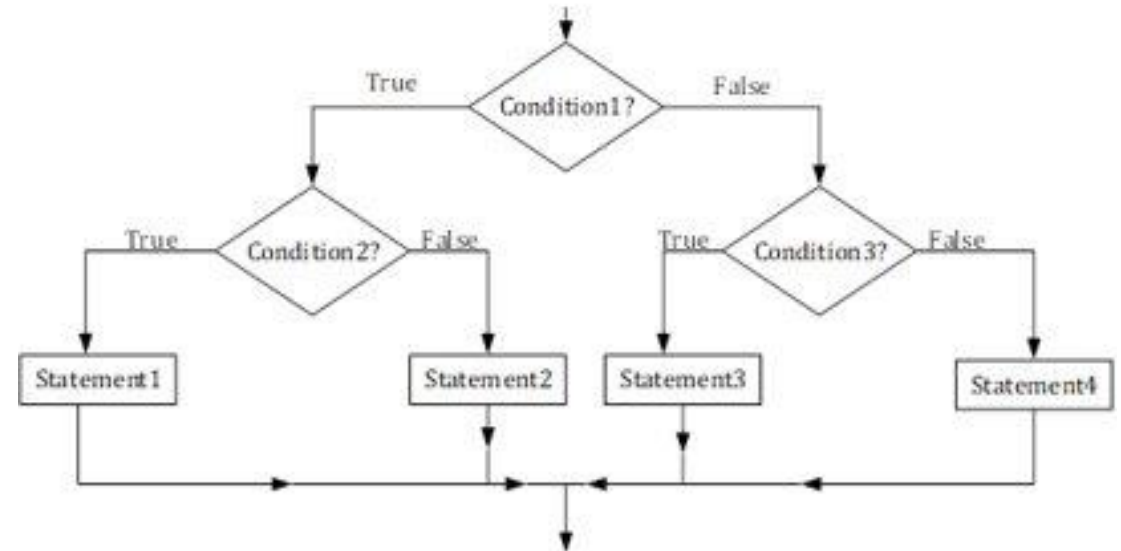
 <true cond2_statement(s)>

else:

 <otherwise statement(s)>

else:

 <otherwise statement(s)>



Lecture

03

ITERATION/LOOPING STATEMENTS

- An **iteration statement**, or loop, repeatedly executes a statement, known as the loop body, until the controlling expression is False.
- The **while statement** evaluates the **control expression** before executing the loop body.
- The **for statement** is used to repeat the execution of a set of statements for every element of a **sequence**.

Lecture

03

ITERATION/LOOPING STATEMENTS

Cont'd

- The `range` function returns a sequence of numbers between the given range.
 - Examples:
`range(10)`
`range(-10)` #empty sequence
`range(2, 10)`
`range(-2, 10)`
`range(10, 2)` #empty sequence
`range(1, 10, 2)`
`range(10, 1, -1)`

Lecture

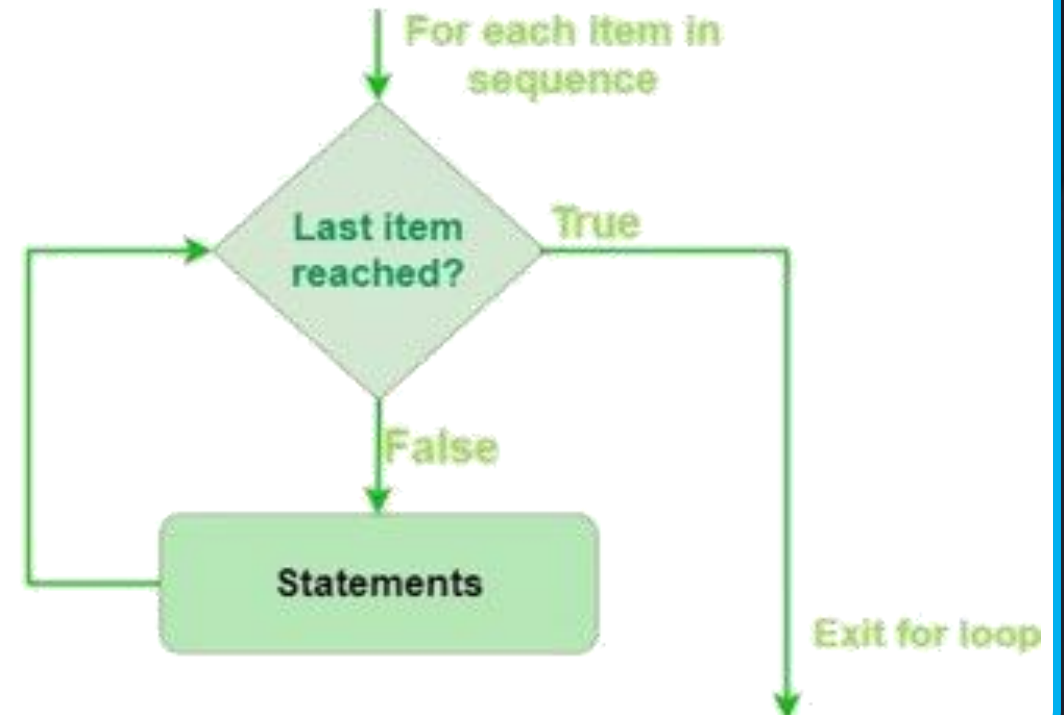
03

ITERATION/LOOPING STATEMENTS

Cont'd

- The **for** statement:
 - Syntax:
for <variable> **in** <sequence> :
 <statement 1>

 <statement n>



Lecture

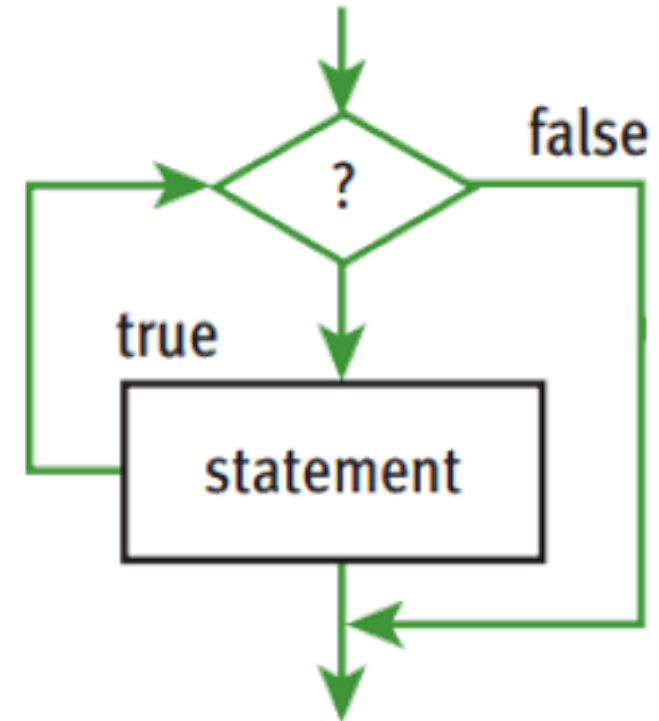
03

ITERATION/LOOPING STATEMENTS

Cont'd

- The **while** statement:
 - Syntax:
while <condition> :
 <statement 1>

 <statement n>



String Data Type

Lecture 4

Prepared by
Dr. Hamada I. AbdulWakel

Lecture

04

STRING DATA TYPE

- String declaration.
 - Using double quotes marks " "
 - `S1 = " this is a string "`
 - Using single quotes marks ' '
 - `S2 = ' this is a string '`
 - Using `str( )` function
 - `S3 = str('this is a string')`

STRING DATA TYPE

Cont'd

- When the string itself contains single or double quotes.

```
S1 = "He said that "I am from Egypt"" #syntax error
```

- Solutions:

- Using the scape character

```
S1 = "He said that \"I am from Egypt\""
```

- Using three-double/single quotes. Also, it used to write a string in multiple lines.

```
S1 = """ He said that "I am from Egypt """
```

```
S2 = ''' He said that  
I am from Egypt '''
```

Lecture

04

STRING DATA TYPE FEATURES

- Immutable: we can not change its content in the same location.
- Subscript operator `[ ]` is useful when you want to use the position/index to get the element in this position, as we done in strings to get the character in a specific position.

Lecture

04

STRING DATA TYPE FEATURES

Cont'd

- Immutable description.

```
s1 = "Hello"
```

```
s1[2] = 'x' #get run-time error
```

```
print( id( s1 ) )
```

```
s1 += " world"
```

```
print( id( s1 ) ) #get a new memory location
```

Lecture

04

STRING OPERATORS

- The + operator means **concatenation**.

```
s1 = "Hello"
```

```
s2 = 'World'
```

```
print( s1 + s2 )
```

- The * operator means **repeatation**.

```
s1 = "Hello World"
```

```
print( s1 * 3 )
```

Lecture

04

STRING FUNCTIONS

- **ord('character'):** to get the ASCII code for the given character.
`print( ord( 'A' ) )`
- **chr(number):** to get the character corresponds to the given number.
`print( chr( 95 ) )`
- **len(string):** to get the string length.
`s1 = "Hello"`
`print( len( s1 ) )`

STRING FUNCTIONS

Cont'd

- **isalpha()**: to check if the string contains **only** alphabetic.
`s1 = "Hello123"`
`print(s1.isalpha())`
- **isdigit()**: to check if the string contains **only** numbers.
`s1 = "123"`
`print(s1.isdigit())`
- **isalnum()**: to check if the string contains alphabetic and number.
`s1 = "Hello123"`
`print(s1.isalnum())`

STRING FUNCTIONS

Cont'd

- **islower():** to check if the string is written in lower case.
`s1 = "ESU"`
`print(s1.islower())`
- **isupper():** to check if the string is written in upper case.
`s1 = "ESU"`
`print(s1.isupper())`
- **isspace():** to check if the string contains whitespaces.
`s1 = " "`
`print(s1.isspace())`

STRING FUNCTIONS

Cont'd

- **startswith(substring):** to check if the string begins with the given substring.
`s1 = "ESU"`
`print(s1.startswith("H"))`
- **endswith(substring):** to check if the string ends with the given substring.
`s1 = "ESU"`
`print(s1.endswith("U"))`

List, Tuple, and Dictionary Data Types

Lecture 5

Prepared by
Dr. Hamada I. AbdulWakel

Lecture

05

NEW PYTHON DATA TYPES

- A **list** allows the programmer to manipulate a **dynamic** sequence of data values of any types.
- A **tuple** is a special case of lists which allows the programmer to manipulate a **fixed** sequence of data values of any types.
- A **dictionary** organizes **dynamic** data values by association with other data values rather than by sequential position.

Lecture

05

LIST FEATURES

- A list literal consists of a comma-separated list of items, enclosed in a pair of square brackets [].
- Each item in a list has a **unique index** that specifies its position.
 - Zero-based indexing: from 0 to length – 1
- Lists are mutable, means it can be changed after they are created.
- List item values can be duplicated and any data types.

Lecture

05

CREATING LISTS

- Empty list

```
L1 = [ ]
```

```
L2 = list()
```

- Same data type.

```
L3 = [1, 2, 3]
```

- Different data types.

```
L4 = [1, True, "ESU"]
```

```
L5 = list( [1, True, "ESU"] )
```

```
L6 = list( "ESU" ) #What about this?
```

ACCESSING LIST ITEMS

Cont'd

- Access items example:

```
Numbers = [1, 2, 3, 4, 5]  
print( Numbers[5] )  
print( Numbers[-1] )  
print( Numbers[10] )  
Numbers[2] = 15  
print( Numbers[2] )  
Numbers[ 0 : 2 ] = [ 6, 7 ]  
print( Numbers )
```

Lecture

05

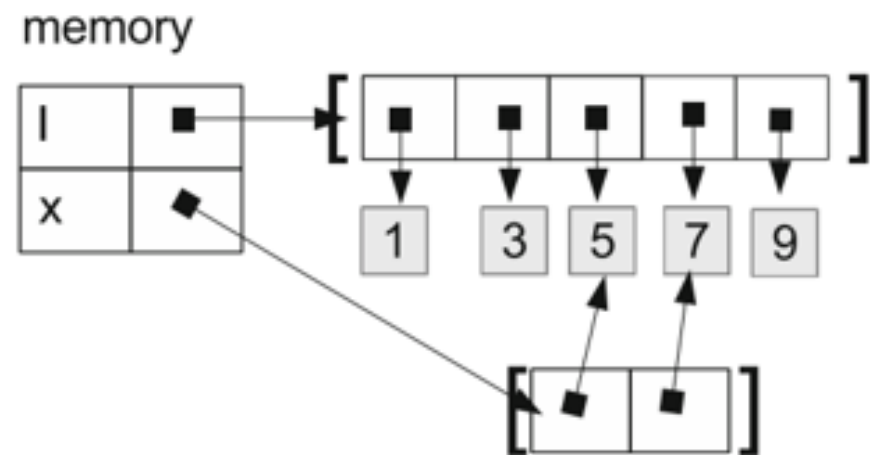
LIST SLICING

Cont'd

- Using this form [start : stop].
- The slicing generate a new list.

```
L1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]  
print( L1[ 3 :    ] )  
print( L1[    : 5 ] )  
print( L1[ 2 : 5 ] )
```

```
l = [1, 3, 5, 7, 9]  
x = l[2:4]
```



Lecture

05

LIST OPERATORS

- The list type supports operations including `+`, `*`, `in`, and `not in`.

```
L1 = [1, 2, 3, 4]
```

```
L2 = L1 + [5, 6]
```

```
print( L2 )
```

```
L2 = L1 * 2
```

```
print( L2 )
```

```
print( 1 in L1 )
```

```
print( 1 not in L1 )
```

Lecture

05

LIST FUNCTIONS

- **len(list):** to get the list length.
`L1 = [1, 2, 3, 4, 5]`
`print( len( L1 ) )`
- **min(list):** to get the minimum value in a list.
`print( min( L1 ) )`
- **max(list):** to get the maximum value in a list.
`print( max( L1 ) )`

LIST FUNCTIONS

Cont'd

- **append(element):** to add the element to the end of a list.
`L1.append(6)`
- **insert(index, element):** to insert element at specified index.
`L1 = [1, 2, 3, 4, 5]`
`L1.insert( 6, 2 )`
- **index(element):** to get the index of an element.
`L1 = [1, 2, 3, 4, 5]`
`L1.index( 2 )`
- **remove(element):** to remove the first occurrence of the element.
`L1.remove(3)`

Lecture

05

DICTIONARY DATA TYPE/STRUCTURE

- A **dictionary** organizes elements between curly braces { }.
- Data structures organized by association are also called **tables** or **association lists**.
- In Python, a dictionary associates a set of **keys** with **data values**.
 - Keys must be immutable type (i.e., number, string, tuple)
 - Keys are unique.
 - Values can be of any type.

Lecture

05

DICTIONARY DATA TYPE/STRUCTURE

Cont'd

- A dictionary creation/declaration.

```
dict0 = {}
```

```
dict1 = dict() #empty dictionaries
```

```
dict2 = { 1 : 123 , "a" : 456 , 2 : True }
```

```
dict3 = dict( [ (1 , 123), ("a" , 456), (2 , True) ] )
```

Lecture

05

ACCESSING DICTIONARY ELEMENTS

- Using the index operator/subscript operator `[]` with element associated key.

```
friends = { 'Mohamed': 30, 'Ahmed': 25, 'Ali': 40 }  
print(friends)  
print(friends[1]) #error: elements are not accessed by numerical index.  
print(friends['Mohamed'])  
print(friends['Abdulrahman']) #error: keyerror 'Abdulrahman'
```

Functions in Python

Lecture 6

Prepared by
Dr. Hamada I. AbdulWakel

Lecture

06

FUNCTIONS AS AN ABSTRACTION MECHANISM

- Functions serve as abstraction mechanisms.
- Functions are the re-usable pieces/block of code which help us to organize structure of the code and do a task.
- Functions allow us to run a specific statements multiple times without repeating these statements.
 - This achieve the concept of "DRY" **don't repeat yourself**.

Lecture

06

BUILT-IN VS. USER-DEFINED FUNCTIONS

- The built-in functions: the functions provided by python language.

`print()`
`input()`
`popitem()`

- The user-defined functions: the functions that the user creates it to do a specific task.

`getUserAccount()`
`updateStudentAddress()`

FUNCTION CREATION

Cont'd

- Code:

```
def sayHello( ):
    print("Hello to Python from ESU")
```

```
def Mysum( ):
    X = 4
    Y = 5
    print("Sum = ", X + Y)
```

Lecture

06

FUNCTION CALLING

- The `def` statement only creates/declares a function but does not call it.
- After creating the function, you can call (run) the function by adding parentheses after the function's name.

```
sayHello( )
```

```
Mysum( )
```

```
mysum( )    #NameError
```

Lecture

06

FUNCTION PARAMETERS AND ARGUMENTS

- A parameter is a **variable** in a function definition. It is a placeholder and hence does not have a concrete value.

```
def Mysum(x, y):  
    print("Sum = ", x + y)
```

- An argument is a **value** passed during function calling.

```
Mysum(2, 3)
```

Lecture

06

RETURNING VALUES FROM FUNCTIONS

- Any function may return a computed value or the default value **None**.
- A function can return a value using **the return statement**.
- Unlike C, C++ or Java, multiple values can be returned by a python function.

Lecture

06

IDENTIFIER SCOPES

- Identifier scope refers to the **accessibility** of identifiers in different parts of code; it is an important concept related to functions.
- The **local scope** refers to the variable defined and used inside a block of code, (i.e., function).

```
def testLocal():  
    a = "Hello im a local variable :)"  
    print(a)
```

```
print(a)
```

- The **global scope** refers to the variable not belongs to a specific block and used inside the whole program.

```
a = "Hello im a global variable :)"  
def testGlobal():  
    print(a)  
  
testGlobal()
```

Lecture

06

IDENTIFIER SCOPES

Cont'd

- If you modify a global scope variable inside a function, this will throw an error called **UnboundLocalError**.

```
a = "Hello im a global variable :)"
```

```
def testGlobal():
```

```
    print(a)    # will throw an error
```

```
    a = "Hello"    #this means a new variable
```

```
    print(a)
```

```
testGlobal()
```

Lecture

06

IDENTIFIER SCOPES

Cont'd

- If you need to modify a global scope variable, you need to use the **global** reserved word.

```
a = "Hello im a global variable :)"
```

```
def testGlobal():
```

```
    global a
```

```
    print(a)
```

```
    a = "Hello"
```

```
    print(a)
```

```
testGlobal()
```

Functions and Modules

Lecture 7

Prepared by
Dr. Hamada I. AbdulWakel

FUNCTION NAMES

Cont'd

- Function names are references to functions and that we can assign multiple names to the same function.
- Functions are stored in variables like everything else.

```
def increment(x):  
    return x + 1  
Myincrease = increment  
print(Myincrease)  
print(increase(5))
```

FUNCTION WITH DEFAULT ARGUMENTS

Cont'd

- Following the required arguments are one or more default arguments.

```
def function(y = 5, x): #SyntaxError
    pass
```

- When function is called with these arguments, default values are **overridden** by caller's values.

```
function(2)
function(2, 3)
```

FUNCTION WITH DEFAULT ARGUMENTS

Cont'd

- Example2:

```
def greet(name, desc="Mr."):
    print("Hello,", desc, name)
```

```
greet("Ahmed")
greet("Ahmed", "Dr.")
```

FUNCTION KEYWORD ARGUMENTS

Cont'd

- Example3:

```
def accessArguments(x, y = 1, z = 2):  
    print("X= ", x, "Y= ", y, "Z= ", z)
```

```
accessArguments(3)
```

```
accessArguments(3, 10)
```

```
accessArguments(3, z = 5)
```

```
accessArguments(3, z = 4, y = 6)
```

Lecture

07

FUNCTION WITH ARBITRARY ARGUMENTS

Cont'd

- Sometimes, we **do not know** in advance the number of arguments that will be passed into a function.
- In the function definition, we use an asterisk * before the parameter name to denote this kind of argument.
- The *args syntax allows a function to receive any number of positional arguments as a **tuple**.

FUNCTION WITH ARBITRARY ARGUMENTS

Cont'd

- Example4:

```
def greet(*names):  
    for name in names:  
        print("Hello," , name)
```

```
greet("Ahmed", "Mohamed")  
greet("Ahmed", "Mohamed", "Aly")  
greet("Ahmed", "Mohamed", "Aly", "Mahmoud")
```

ANONYMOUS FUNCTIONS

Cont'd

- An anonymous function is a function that is defined **without a name**.
- Anonymous functions are defined using the **lambda** keyword.
- Lambda functions can have any number of arguments but **only one expression**.

lambda arguments: expression

ANONYMOUS FUNCTIONS

Cont'd

- Example 11:

```
double = lambda x: x * 2  
print(double(5))
```

- This function has no name. It returns a **function object** which is assigned to the identifier **double**. We can now call it as a normal function.

Lecture

07

MODULES IN PYTHON

Cont'd

- Modules are assemblies of code that provide specific functionalities and constants for **reuse**.
- Adding a module to your program helping you doing some tasks to save time and effort, where you can add/call multiple modules.
- Constants and functions defined in a module can be accessed by using the module name and **the dot (.) operator**.
`<module>.<identifier>`

- Calling modules in a program:

```
import module_name1, module_name2, .....
```

- Example 16.

```
import random  
print(random)  
print(random.random())  
print(dir(random))
```

The background features a series of vertical panels with rounded ends. The leftmost panel shows a person's hands typing on a laptop. The subsequent panels are semi-transparent and contain various white line-art icons: a graduation cap, a play button, a target with an arrow, an open book, a magnifying glass, and a group of people. The overall color palette is warm, with shades of orange and brown.

Object-Oriented Programming

Lecture 8

Prepared by
Dr. Hamada I. AbdulWakel

Lecture

08

POP: FEATURES

- The main concern in POP is **doing things**.
- Depend on the built-in and user-defined **functions**.
- Based on local and global variables in parameter passing.
- **Modularity**: means splitting the program into parts where the top-down design can be applied.

Lecture

08

PRIMITIVE DATA TYPES

- **Primitive data types**, a.k.a. basic or fundamental data types, are the simplest and most basic building blocks in a programming language. It store single simple value.
- The primitive data types vary between programming languages, but some common ones include:
 - Integer, Short, and Long.
 - Floating-point, Double.
 - Character.
 - Boolean.
 - String.

Lecture

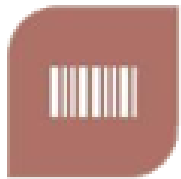
08

COMPLEX/CUSTOMER DATA TYPES

Cont'd



Customer



Product



Post



Name, address, payment method



Name, **description**, is it in stock?



User, **post text**, date

Lecture

08

OBJECT-ORIENTED PROGRAMMING

- Object-Oriented Programming (OOP) is a programming paradigm that organizes code into objects, which can encapsulate data and behavior.
 - Program is divided into classes and objects.
 - The main concern on **data**.
 - Strong modeling to real world problems.
 - It is easy to maintain program even if it is too complex.
 - Provide strong data security.
 - It is highly extensible programming paradigm.
 - Unit of programming is **class**.
 - Ex: Python, Java, C#.



Lecture

08

OOP: CLASS

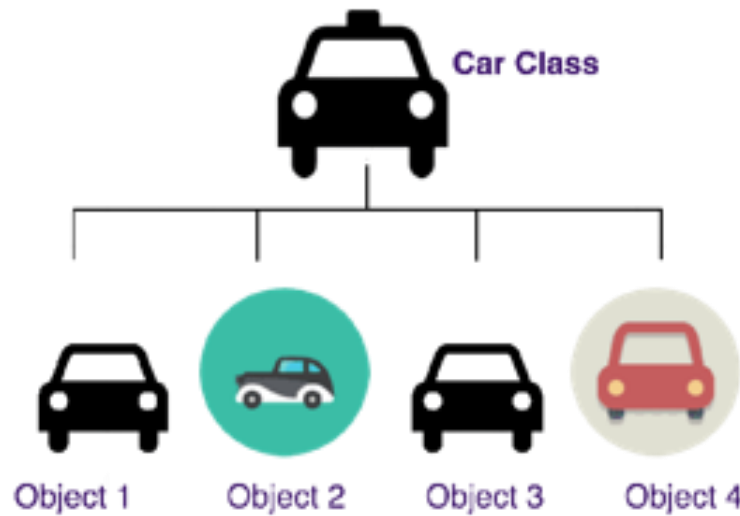
- **Classes:** the user-defined data types that act as the blueprint for individual objects, attributes and methods.
 - **Attributes:** are defined in the class template and represent the state of an object. Class attributes belong to the class itself.
 - **Methods:** are functions that are defined inside a class that describe the behaviors of an object.

Lecture

08

OOP: OBJECT

- **Objects:** are **instances** of classes and represent real-world entities. They encapsulate data (attributes) and behavior (methods) related to those entities.



Lecture

08

CLASS DEFINITION

- All class definitions start with the `class` keyword, which is followed by the name of the class and a **colon**.
- Any code that is **indented** below the class definition is considered part of the class's body.

```
class className :  
    #class body
```

Lecture

08

INSTANCE ATTRIBUTES

- Instance attributes are the **properties** that all objects must have and defined inside the method called `__init__()`.
- When a new object is created, `__init__()` sets **the initial state** of the object by assigning the values of the object's properties.
- If you change the attributes of an instance, it won't affect other instances.
- The `__init__()` can take any number of parameters, but the first parameter will always be a variable called **self**.

INSTANCE ATTRIBUTES

Cont'd

- Example:

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

- Both **name** and **age** are called instance attributes.

Lecture

08

CLASS ATTRIBUTES

- The class attributes are properties that have **the same value** for all class instances.
- Class attributes are defined directly **under the line of the class name**, outside the `__init__()` function.
- You can access the class attribute via **instances** of the class or via the **class name**.

Lecture

08

DEALING WITH INSTANCE AND CLASS ATTRIBUTES

- When you access an attribute via an **instance of the class**, Python searches for the attribute in **the instance attribute list**.
- If the instance attribute list doesn't have that attribute, Python continues looking up the attribute in **the class attribute list**.
- However, if you access an attribute via **the class name**, Python directly searches for the attribute in **the class attribute list**.

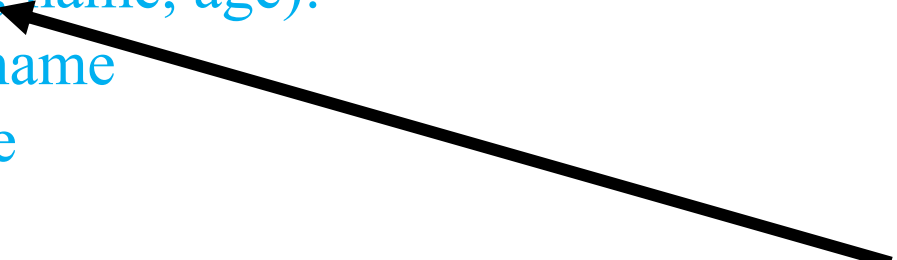
INSTANTIATING OBJECTS

Cont'd

- Example:

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

`std1 = Student("Ahmed", 20)` → The object **std1** is automatically passed where `self` is referring to the object **std1**





Object-Oriented Programming II

Lecture 9

Prepared by
Dr. Hamada I. AbdulWakel

Lecture

09

THE GETTER AND SETTER METHODS

- **getattr(obj, 'paramName')**: returns the current value of this parameter.
- **setattr(obj, 'paramName', value)**: allows you to set/change the value of an attribute.
- **hasattr(obj, 'Name')**: checks whether the object has this attribute or not.
- **isinstance(obj, className)**: returns True if the specified object is of the specified type, otherwise False .

Lecture

09

THE OPERATOR OVERLOADING

- Example:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
p1 = Point(1, 2)
p2 = Point(2, 3)
print(p1+p2)
```

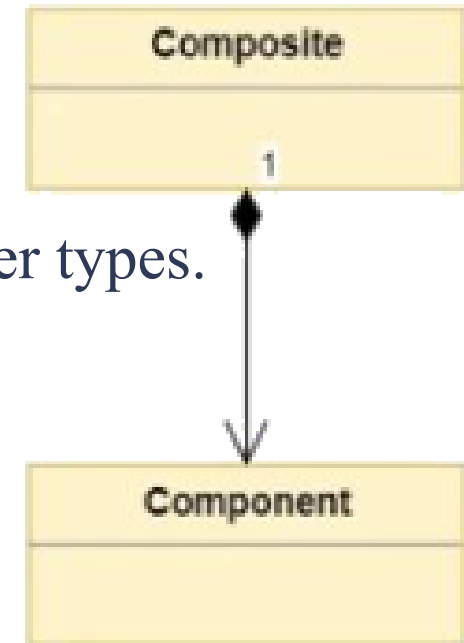
THE OPERATOR OVERLOADING

Cont'd

- Among the special functions for operators are:
 - + by overloading `__add__(self, otherObj)` for **obj1 + obj2**
 - - by overloading `__sub__(self, otherObj)` for **obj1 - obj2**
 - * by overloading `__mul__(self, otherObj)` for **obj1 * obj2**
 - / by overloading `__div__(self, otherObj)` for **obj1 / obj2**
 - % by overloading `__mod__(self, otherObj)` for **obj1 % obj2**
 - ** by overloading `__pow__(self, otherObj)` for **obj1 ** obj2**

CLASS COMPOSITION

- **Composition:** is a concept that models a **has a** relationship.
- It enables creating complex types by combining objects of other types.
- This means that a class Composite can contain an object of another class Component.
- A composition relationship occurs if one class has an **instance attribute** which is an object from the other class (car has wheels).



CLASS COMPOSITION

Cont'd

- Example:

```
class Course:
    def __init__(self, name):
        self.cname = name;    self.hours = 3

class Student:
    def __init__(self, name):
        self.course = Course("OOP")
        self.name = name

s = Student("ahmed")
print( s.course.cname )
```

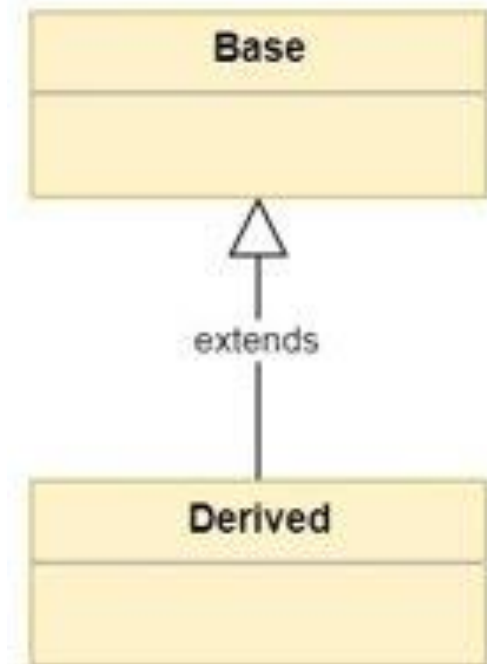
Lecture

09

INHERITANCE

- **Inheritance:** models what is called an **is a** relationship.
- A class can inherit **all** the methods and attributes of another class. These methods and attributes become available to the new class without the need to redefine them from scratch.
- A **derived class** that inherits from a **Base class**.

```
class DrivedClassName(BaseClassName):  
    # body
```



Lecture

09

POLYMORPHISM

- Polymorphism is taken from the Greek words **Poly** (many) and **morphism** (forms).
- it means that the methods with **the same name** can be executed on **many** objects or classes.
- There are two ways to perform polymorphism:
 - Using functions.
 - Using class methods.

Lecture

09

CLASS POLYMORPHISM

- Polymorphism defines methods in **the child class** that have the same name as the methods in the parent class.
- The inherited child class method will modify the parent's implementation with an implementation that suitable for the child class.
- This process of re-implementing a method in the child class is known as **Method Overriding**.



File Handling and Graphical User Interface

Lecture 10

Prepared by
Dr. Hamada I. AbdulWakel

Lecture

10

FILES

- Since RAM is volatile (which loses its data when the computer is turned off).
- **Files** are named locations on disk to store related information.
- They are used to permanently store data in a non-volatile memory (e.g. hard disk).



Lecture

10

FILE OPENING

- The syntax of opening a file is:
`File_object_name = open(filename, mode)`
- The "filename" is a **string** argument which specify filename along with its path.
- The mode is a **string** argument which used to specify how file will be used.
 - The reading mode represented by "r".
 - The writing mode represented by "w".
 - The appending mode represented by "a".

Lecture

10

FILE CLOSING

- When we are done with performing operations on the file, we need to properly close the file.
- Closing a file will free up the resources that were tied with the file.
- The syntax for closing a file is:

```
File_object_name.close()
```

Lecture

10

FILE READING DATA

- We must open the file in reading **r** mode.
- To read data back from the file you need one of these methods:
 - **read([number])**: return specified number of characters from the file.
 - If the number is omitted, it will read the entire content.
 - **readline()**: returns the next line of the file.
 - **readlines()**: read all the lines as a list of strings.
- **FileNotFoundError** is the error generated if the file you try to read is not exists.

Lecture

10

FILE WRITING DATA

Cont'd

- We need to open it in write w, append a or exclusive creation x mode.
- We need to be careful with the w mode, as it will overwrite into the file if it already exists.
- Code:

```
import os
os.chdir("/path/to/the/folder/")
with open("myfile.txt", "w") as f:
    f.write("My Name: Hamada I.")
    f.write("I'm a teacher at \n" )
```

Lecture

10

FILE RENAMING AND DELETING

- The os module provides methods that help to perform file-processing operations, such as renaming and deleting.
- To rename an existing file, rename() method is used.
- It takes two arguments, current file name and new file name.
- Syntax:

```
os.rename(current_file_name, new_file_name)
```

```
os.remove(filename)
```

Lecture

10

COMMAND LINE INTERFACE

- **The command-line interface (CLI):** takes input from the user via text commands entered in a terminal or console window.
- **CLI features:**
 - Based on commands a user inputs in the terminal.
 - More efficient and faster since it doesn't require much memory or processing power as the entire interaction is performed in a terminal window.
 - Requires memorizing commands, which sometimes have complex syntax and many arguments.
 - Takes input from the keyboard.
 - Users cannot change CLI's appearance.

COMMAND-LINE
INTERFACE
(CLI)

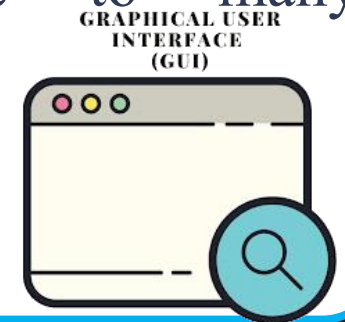
> _

Lecture

10

GRAPHICAL USER INTERFACE

- **The graphical-user interface (GUI) version:** based on graphics and visual elements, allowing users to interact with the system using a keyboard and a mouse.
- **GUI features:**
 - Based on visual elements such as icons, buttons, windows, dialog boxes, etc.
 - Requires more memory and processing power due to many graphical components that need to be loaded.
 - Icons method reduces the required knowledge level.
 - Takes input from the mouse and keyboard.
 - Appearance can be modified to suit the users' needs.

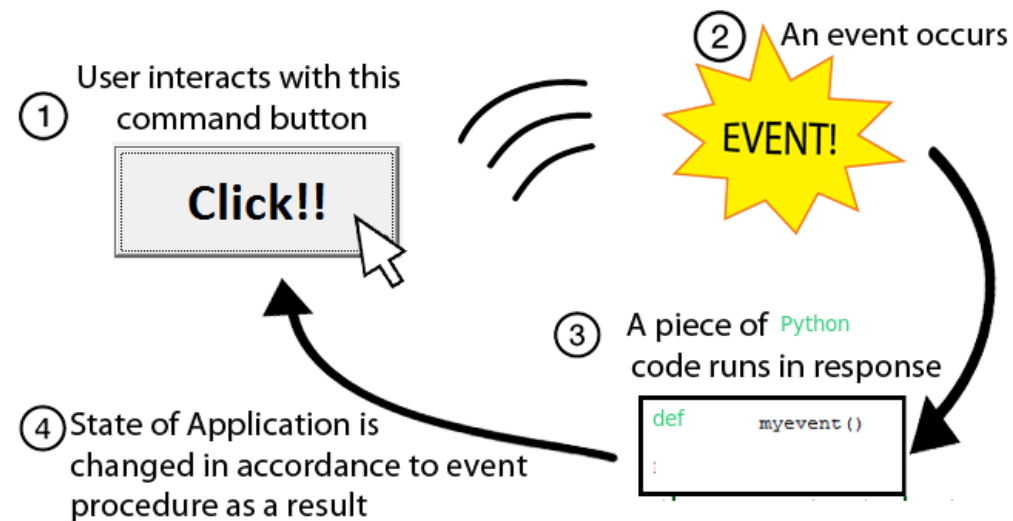


Lecture

10

EVENT-DRIVEN PROGRAMMING

- In GUI, the user-generated events (i.e., mouse clicks, button clicks) trigger operations in the program to respond by pulling in inputs, processing them, and displaying results.



Lecture

10

TKINTER MESSAGE BOXES

Cont'd

- Code:

```
from tkinter import messagebox
```

```
messagebox.showinfo("title","Information")
```

```
messagebox.showwarning("title","Warning")
```

```
messagebox.showerror("title","Error")
```

```
messagebox.askquestion("title", "Do you love GUI?")
```

```
response = messagebox.askyesnocancel("title", "Do you love GUI?")
```

```
print(response)
```

- Code:

```
from tkinter import *
```

```
mainFrame = Tk()
```

```
mainFrame.title("The main frame title")
```

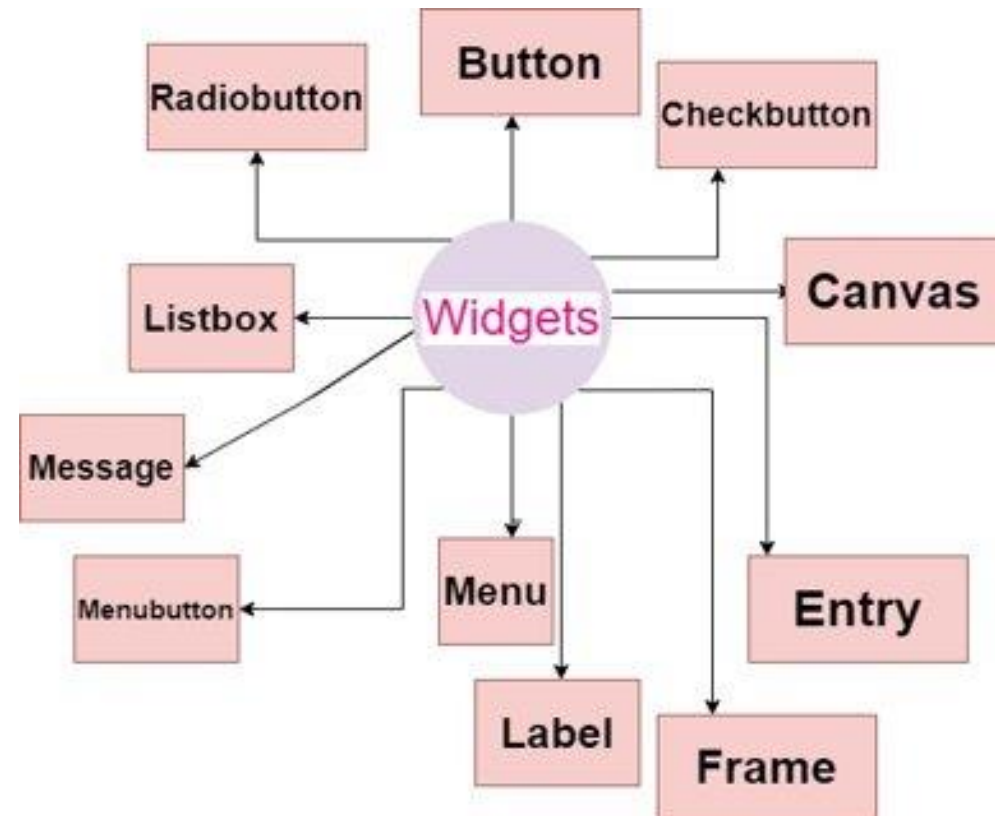
```
mainFrame.geometry("300x300")
```

```
mainFrame.mainloop() #listens for events
```

Lecture

10

TKINTER WIDGETS

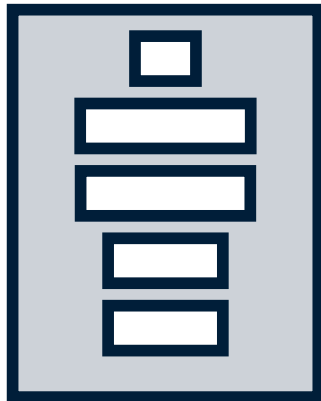


Lecture

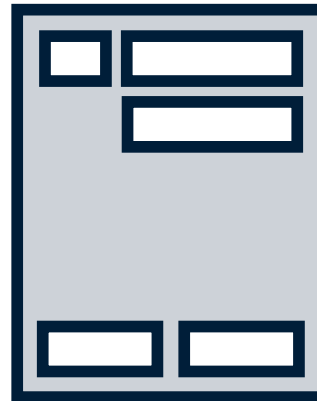
10

TKINTER GEOMETRY MANAGER

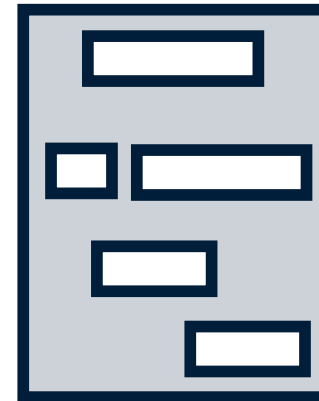
- The geometry measurements allow you to organize the widgets and throughout the parent frames or parent widget area.

**Pack**

One after another,
vertically or horizontally

**Grid**

Assigned to grid cells,
using rows and columns

**Place**

Put stuff anywhere,
using coordinates



Thank You

