

PYTHON PROGRAMMING LECTURE NOTES

**DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING**

SYLLABUS

OBJECTIVES:

- To read and write simple Python programs.
- To develop Python programs with conditionals and loops.
- To define Python functions and call them.
- To use Python data structures — lists, tuples, dictionaries.
- To do input/output with files in Python.

UNIT I**INTRODUCTION DATA, EXPRESSIONS, STATEMENTS**

Introduction to Python and installation, data types: Int, float, Boolean, string, and list; variables, expressions, statements, precedence of operators, comments; modules, functions--- function and its use, flow of execution, parameters and arguments.

UNIT II**CONTROL FLOW, LOOPS**

Conditionals: Boolean values and operators, conditional (if), alternative (if-else), chained conditional (if-elif-else); Iteration: while, for, break, continue.

UNIT III**FUNCTIONS, ARRAYS**

Fruitful functions: return values, parameters, local and global scope, function composition, recursion; Strings: string slices, immutability, string functions and methods, string module; Python arrays, Access the Elements of an Array, array methods.

UNIT IV**LISTS, TUPLES, DICTIONARIES**

Lists: list operations, list slices, list methods, list loop, mutability, aliasing, cloning lists, list parameters, list comprehension; Tuples: tuple assignment, tuple as return value, tuple comprehension; Dictionaries: operations and methods, comprehension;

UNIT V**FILES, EXCEPTIONS, MODULES, PACKAGES**

Files and exception: text files, reading and writing files, command line arguments, errors and exceptions, handling exceptions, modules (datetime, time, OS , calendar, math module), Explore packages.

OUTCOMES: Upon completion of the course, students will be able to

- Read, write, execute by hand simple Python programs.
- Structure simple Python programs for solving problems.
- Decompose a Python program into functions.
- Represent compound data using Python lists, tuples, dictionaries.
- Read and write data from/to files in Python Programs

TEXT BOOKS

1. Allen B. Downey, ``Think Python: How to Think Like a Computer Scientist``, 2nd edition, Updated for Python 3, Shroff/O'Reilly Publishers, 2016.
2. R. Nageswara Rao, "Core Python Programming", dreamtech
3. Python Programming: A Modern Approach, Vamsi Kurama, Pearson

REFERENCE BOOKS:

1. Core Python Programming, W.Chun, Pearson.
2. Introduction to Python, Kenneth A. Lambert, Cengage
3. Learning Python, Mark Lutz, Orielly

INDEX

UNIT	TOPIC	PAGE NO
I	INTRODUCTION DATA, EXPRESSIONS, STATEMENTS	1
	Introduction to Python and installation	1
	data types: Int	6
	float	7
	Boolean	8
	string	8
	List	10
	variables	11
	expressions	13
	statements	16
	precedence of operators	17
	comments	18
	modules	19
	functions----- function and its use	20
	flow of execution	21
	parameters and arguments	26
II	CONTROL FLOW, LOOPS	35
	Conditionals: Boolean values and operators,	35
	conditional (if)	36
	alternative (if-else)	37
	chained conditional (if-elif-else)	39
	Iteration: while, for, break, continue.	41
III	FUNCTIONS, ARRAYS	55
	Fruitful functions: return values	55
	parameters	57
	local and global scope	59
	function composition	62
	recursion	63
	Strings: string slices	64
	immutability	66
	string functions and methods	67
	string module	72
	Python arrays	73
	Access the Elements of an Array	75
	Array methods	76

IV	LISTS, TUPLES, DICTIONARIES	78
	Lists	78
	list operations	79
	list slices	80
	list methods	81
	list loop	83
	mutability	85
	aliasing	87
	cloning lists	88
	list parameters	89
	list comprehension	90
	Tuples	91
	tuple assignment	94
	tuple as return value	95
	tuple comprehension	96
	Dictionaries	97
	operations and methods	97
	comprehension	102
V	FILES, EXCEPTIONS, MODULES, PACKAGES	103
	Files and exception: text files	103
	reading and writing files	104
	command line arguments	109
	errors and exceptions	112
	handling exceptions	114
	modules (datetime, time, OS , calendar, math module)	121
	Explore packages	134

INTRODUCTION DATA, EXPRESSIONS, STATEMENTS

Introduction to Python and installation, data types: Int, float, Boolean, string, and list; variables, expressions, statements, precedence of operators, comments; modules, functions--
- function and its use, flow of execution, parameters and arguments.

Introduction to Python and installation:

Python is a widely used general-purpose, high level programming language. It was initially designed by **Guido van Rossum in 1991** and developed by Python Software Foundation. It was mainly developed for emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code.

Python is a programming language that lets you work quickly and integrate systems more efficiently.

There are two major Python versions- **Python 2 and Python 3**.

- On 16 October 2000, Python 2.0 was released with many new features.
- On 3rd December 2008, Python 3.0 was released with more testing and includes new features.

Beginning with Python programming:

1) Finding an Interpreter:

Before we start Python programming, we need to have an interpreter to interpret and run our programs. There are certain online interpreters like <https://ide.geeksforgeeks.org/>, <http://ideone.com/> or <http://codepad.org/> that can be used to start Python without installing an interpreter.

Windows: There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from <http://python.org/downloads/>

2) Writing first program:

```
# Script Begins
```

```
Statement1
```

Statement2

Statement3

Script Ends

Differences between scripting language and programming language:

SCRIPTING LANGUAGE	PROGRAMMING LANGUAGE
A programming language that supports scripts: programs written for a special run-time environment that automate the execution of tasks	A formal language, which comprises a set of instructions used to produce various kinds of output
Execution speed is slow	Compiler-based languages are executed much faster while interpreter-based languages are executed slower
Can be divided into client-side scripting languages and server-side scripting languages	Can be divided into high-level, low-level languages or compiler-based or interpreter-based languages
Easier to learn	Not as easy to learn
Ex: JavaScript, Perl, PHP, Python and Ruby	Ex: C, C++, and Assembly
Mostly used for web development	Used to develop various applications such as desktop, web, mobile, etc.

Why to use Python:

The following are the primary factors to use python in day-to-day life:

1. Python is object-oriented

Structure supports such concepts as polymorphism, operation overloading and multiple inheritance.

2. Indentation

Indentation is one of the greatest feature in python

3. It's free (open source)

Downloading python and installing python is free and easy

4. It's Powerful

- Dynamic typing
- Built-in types and tools
- Library utilities
- Third party utilities (e.g. Numeric, NumPy, sciPy)
- Automatic memory management

5. It's Portable

- Python runs virtually every major platform used today
- As long as you have a compatible python interpreter installed, python programs will run in exactly the same manner, irrespective of platform.

6. It's easy to use and learn

- No intermediate compile
- Python Programs are compiled automatically to an intermediate form called byte code, which the interpreter then reads.
- This gives python the development speed of an interpreter without the performance loss inherent in purely interpreted languages.
- Structure and syntax are pretty intuitive and easy to grasp.

7. Interpreted Language

Python is processed at runtime by python Interpreter

8. Interactive Programming Language

Users can interact with the python interpreter directly for writing the programs

9. Straight forward syntax

The formation of python syntax is simple and straight forward which also makes it popular.

Installation:

There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from <http://python.org/downloads/>

Steps to be followed and remembered:

Step 1: Select Version of Python to Install.

Step 2: Download Python Executable Installer.

Step 3: Run Executable Installer.

Step 4: Verify Python Was Installed On Windows.

Step 5: Verify Pip Was Installed.

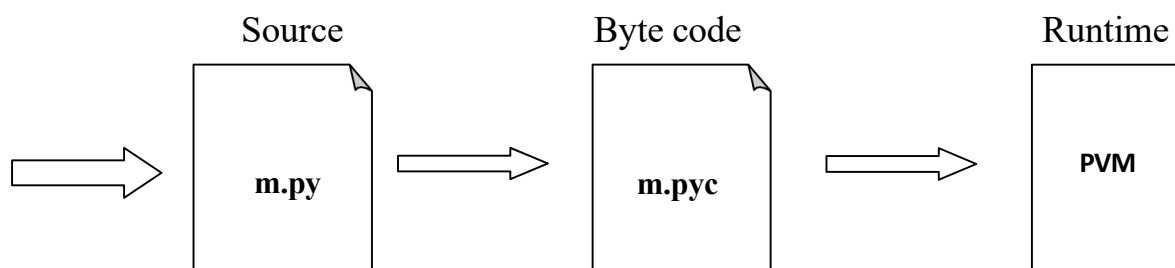
Step 6: Add Python Path to Environment Variables (Optional)



Working with Python

Python Code Execution:

Python's traditional runtime execution model: Source code you type is translated to byte code, which is then run by the Python Virtual Machine (PVM). Your code is automatically compiled, but then it is interpreted.



Source code extension is .py
Byte code extension is .pyc (Compiled python code)

There are two modes for using the Python interpreter:

- Interactive Mode
- Script Mode

Running Python in interactive mode:

Without passing python script file to the interpreter, directly execute code to Python prompt. Once you're inside the python interpreter, then you can start.

```
>>> print("hello world")
```

```
hello world
```

Relevant output is displayed on subsequent lines without the >>> symbol

```
>>> x=[0,1,2]
```

Quantities stored in memory are not displayed by default.

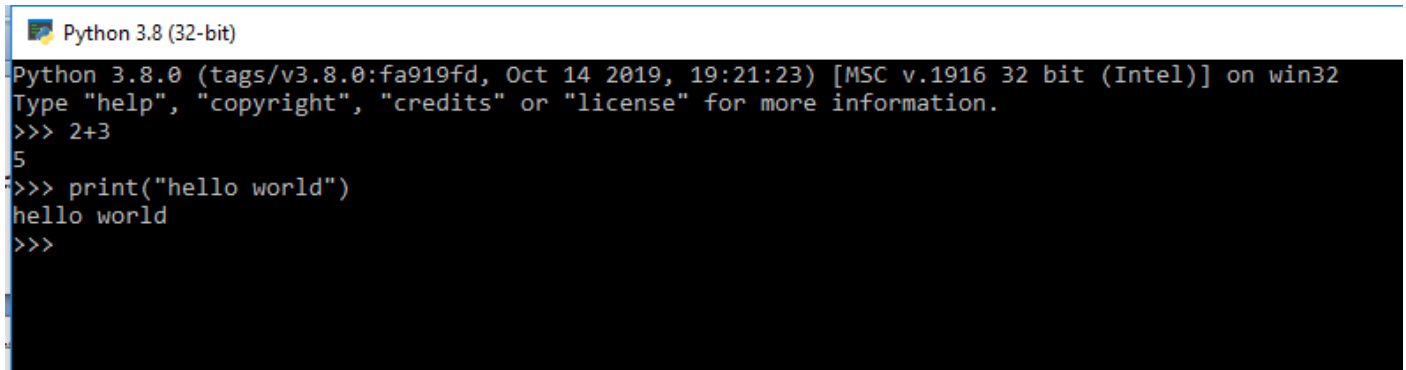
```
>>> x
```

#If a quantity is stored in memory, typing its name will display it.

```
[0, 1, 2]
```

```
>>> 2+3
```

```
5
```



```
Python 3.8 (32-bit)
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3
5
>>> print("hello world")
hello world
>>>
```

The chevron at the beginning of the 1st line, i.e., the symbol >>> is a prompt the python interpreter uses to indicate that it is ready. If the programmer types 2+6, the interpreter replies 8.

Running Python in script mode:

Alternatively, programmers can store Python script source code in a file with the .py extension, and use the interpreter to execute the contents of the file. To execute the script by the interpreter, you have to tell the interpreter the name of the file. For example, if you have a script name MyFile.py and you're working on Unix, to run the script you have to type:

python MyFile.py

Working with the interactive mode is better when Python programmers deal with small pieces of code as you can type and execute them immediately, but when the code is more than 2-4 lines, using the script for coding can help to modify and use the code in future.

Example:

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\python>python e1.py
resource open
the no cant be divisibile zero division by zero
resource close
finished
```

Data types:

The data stored in memory can be of many types. For example, a student roll number is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Int:

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

```
>>> print(24656354687654+2)
```

```
24656354687656
```

```
>>> print(20)
```

```
20
```

```
>>> print(0b10)
```

```
2
```

```
>>> print(0B10)
```

```
2
```

```
>>> print(0X20)
```

```
32
```

```
>>> 20
```

```
20
```

```
>>> 0b10
```

```
2
```

```
>>> a=10
```

```
>>> print(a)
```

```
10
```

To verify the type of any object in Python, use the type() function:

```
>>> type(10)
```

```
<class 'int'>
```

```
>>> a=11
```

```
>>> print(type(a))
```

```
<class 'int'>
```

Float:

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

Float can also be scientific numbers with an "e" to indicate the power of 10.

```
>>> y=2.8
```

```
>>> y
```

```
2.8
```

```
>>> y=2.8
```

```
>>> print(type(y))
```

```
<class 'float'>
```

```
>>> type(.4)
```

```
<class 'float'>
```

```
>>> 2.
```

2.0

Example:`x = 35e3``y = 12E4``z = -87.7e100``print(type(x))``print(type(y))``print(type(z))`**Output:**`<class 'float'>``<class 'float'>``<class 'float'>`**Boolean:**

Objects of Boolean type may have one of two values, True or False:

`>>> type(True)``<class 'bool'>``>>> type(False)``<class 'bool'>`**String:**

1. Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes.

- 'hello' is the same as "hello".
- Strings can be output to screen using the print function. **For example: print("hello").**

`>>> print("mrcet college")``mrcet college``>>> type("mrcet college")``<class 'str'>`

```
>>> print('mrcet college')
```

```
mrcet college
```

```
>>> " "
```

```
''
```

If you want to include either type of quote character within the string, the simplest way is to delimit the string with the other type. If a string is to contain a single quote, delimit it with double quotes and vice versa:

```
>>> print("mrcet is an autonomous (') college")
```

```
mrcet is an autonomous (') college
```

```
>>> print('mrcet is an autonomous (") college')
```

```
mrcet is an autonomous (") college
```

Suppressing Special Character:

Specifying a backslash (\) in front of the quote character in a string “escapes” it and causes Python to suppress its usual special meaning. It is then interpreted simply as a literal single quote character:

```
>>> print("mrcet is an autonomous (\') college")
```

```
mrcet is an autonomous (') college
```

```
>>> print('mrcet is an autonomous (\") college')
```

```
mrcet is an autonomous (") college
```

The following is a table of escape sequences which cause Python to suppress the usual special interpretation of a character in a string:

```
>>> print('a\
```

```
....b')
```

```
a....b
```

```
>>> print('a\
```

```
b\
```

```
c')
```

```
abc
>>> print('a \n b')
a
b
>>> print("mrcet \n college")
mrcet
college
```

Escape Sequence	Usual Interpretation of Character(s) After Backslash	“Escaped” Interpretation
\'	Terminates string with single quote opening delimiter	Literal single quote (') character
\"	Terminates string with double quote opening delimiter	Literal double quote (") character
\newline	Terminates input line	Newline is ignored
\\	Introduces escape sequence	Literal backslash (\) character

In Python (and almost all other common computer languages), a tab character can be specified by the escape sequence `\t`:

```
>>> print("a\tb")
a      b
```

List:

- It is a general purpose most widely used in data structures
- List is a collection which is ordered and changeable and allows duplicate members. (Grow and shrink as needed, sequence type, sortable).
- To use a list, you must declare it first. Do this using square brackets and separate values with commas.
- We can construct / create list in many ways.

Ex:

```
>>> list1=[1,2,3,'A','B',7,8,[10,11]]
>>> print(list1)
[1, 2, 3, 'A', 'B', 7, 8, [10, 11]]
```

```
-----  
>>> x=list()
```

```
>>> x
```

```
[]  
-----
```

```
>>> tuple1=(1,2,3,4)
```

```
>>> x=list(tuple1)
```

```
>>> x
```

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

Variables:

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Assigning Values to Variables:

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

For example –

```
a= 100      # An integer assignment
```

```
b = 1000.0   # A floating point
```

```
c = "John"   # A string
```

```
print (a)
```

```
print (b)
```

```
print (c)
```

This produces the following result –

```
100
```

```
1000.0
```

```
John
```

Multiple Assignment:

Python allows you to assign a single value to several variables simultaneously.

For example :

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables.

For example –

```
a,b,c = 1,2,"mrcet"
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Output Variables:

The Python print statement is often used to output variables.

Variables do not need to be declared with any particular type and can even change type after they have been set.

```
x = 5          # x is of type int
x = "mrcet "    # x is now of type str
print(x)
```

Output: mrcet

To combine both text and a variable, Python uses the “+” character:

Example

```
x = "awesome"
print("Python is " + x)
```

Output

Python is awesome

You can also use the + character to add a variable to another variable:

Example

```
x = "Python is "
y = "awesome"
z = x + y
print(z)
```

Output:

Python is awesome

Expressions:

An expression is a combination of values, variables, and operators. An expression is evaluated using assignment operator.

Examples: $Y = x + 17$

```
>>> x=10
```

```
>>> z=x+20
```

```
>>> z
```

30

```
>>> x=10
>>> y=20
>>> c=x+y
>>> c
30
```

A value all by itself is a simple expression, and so is a variable.

```
>>> y=20
>>> y
20
```

Python also defines expressions only contain identifiers, literals, and operators. So,

Identifiers: Any name that is used to define a class, function, variable module, or object is an identifier.

Literals: These are language-independent terms in Python and should exist independently in any programming language. In Python, there are the string literals, byte literals, integer literals, floating point literals, and imaginary literals.

Operators: In Python you can implement the following operations using the corresponding tokens.

Operator	Token
add	+
subtract	-
multiply	*
Integer Division	/
remainder	%
Binary left shift	<<
Binary right shift	>>
and	&
or	\
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=
Check equality	==
Check not equal	!=

Some of the python expressions are:**Generator expression:**

Syntax: (compute(var) for var in iterable)

```
>>> x = (i for i in 'abc') #tuple comprehension
>>> x
<generator object <genexpr> at 0x033EEC30>
```

```
>>> print(x)
<generator object <genexpr> at 0x033EEC30>
```

You might expect this to print as ('a', 'b', 'c') but it prints as <generator object <genexpr> at 0x02AAD710> The result of a tuple comprehension is not a tuple: it is actually a generator. The only thing that you need to know now about a generator now is that you can iterate over it, but ONLY ONCE.

Conditional expression:

Syntax: true_value if Condition else false_value

```
>>> x = "1" if True else "2"
```

```
>>> x
```

```
'1'
```

Statements:

A statement is an instruction that the Python interpreter can execute. We have normally two basic statements, the assignment statement and the print statement. Some other kinds of statements that are if statements, while statements, and for statements generally called as control flows.

Examples:

An assignment statement creates new variables and gives them values:

```
>>> x=10
```

```
>>> college="mrcet"
```

An print statement is something which is an input from the user, to be printed / displayed on to the screen (or) monitor.

```
>>> print("mrcet colege")
```

```
mrcet college
```

Precedence of Operators:

Operator precedence affects how an expression is evaluated.

For example, $x = 7 + 3 * 2$; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first multiplies $3*2$ and then adds into 7.

Example 1:

```
>>> 3+4*2
```

```
11
```

Multiplication gets evaluated before the addition operation

```
>>> (10+10)*2
```

```
40
```

Parentheses () overriding the precedence of the arithmetic operators

Example 2:

```
a = 20
```

```
b = 10
```

```
c = 15
```

```
d = 5
```

```
e = 0
```

```
e = (a + b) * c / d    #( 30 * 15 ) / 5
```

```
print("Value of (a + b) * c / d is ", e)
```

```
e = ((a + b) * c) / d  # (30 * 15 ) / 5
```

```
print("Value of ((a + b) * c) / d is ", e)
```

```
e = (a + b) * (c / d);  # (30) * (15/5)
```

```
print("Value of (a + b) * (c / d) is ", e)
```

```
e = a + (b * c) / d;    # 20 + (150/5)
print("Value of a + (b * c) / d is ", e)
```

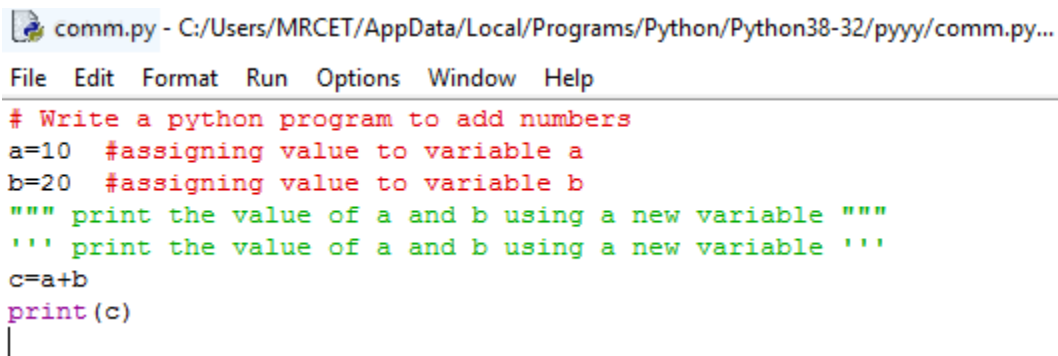
Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/opprec.py
Value of (a + b) * c / d is 90.0
Value of ((a + b) * c) / d is 90.0
Value of (a + b) * (c / d) is 90.0
Value of a + (b * c) / d is 50.0
```

Comments:

Single-line comments begins with a hash(#) symbol and is useful in mentioning that the whole line should be considered as a comment until the end of line.

A Multi line comment is useful when we need to comment on many lines. In python, triple double quote(“ “ “) and single quote(‘ ‘ ‘)are used for multi-line commenting.

Example:

```
comm.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/comm.py...
File Edit Format Run Options Window Help
# Write a python program to add numbers
a=10 #assigning value to variable a
b=20 #assigning value to variable b
""" print the value of a and b using a new variable """
''' print the value of a and b using a new variable '''
c=a+b
print(c)
|
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/comm.py
```

30

Modules:

Modules: Python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module. A module in Python provides us the flexibility to organize the code in a logical way. To use the functionality of one module into another, we must have to **import** the specific module.

Syntax:

```
import <module-name>
```

Every module has its own functions, those can be accessed with . (dot)

Note: In python we have help ()

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

Some of the modules like os, date, and calendar so on.....

```
>>> import sys
```

```
>>> print(sys.version)
```

```
3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)]
```

```
>>> print(sys.version_info)
```

```
sys.version_info(major=3, minor=8, micro=0, releaselevel='final', serial=0)
```

```
>>> print(calendar.month(2021,5))
```

```
    May 2021
Mo Tu We Th Fr Sa Su
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

```
>>> print(calendar.isleap(2020))
```

```
True
```

```
>>> print(calendar.isleap(2017))
```

```
False
```


Functions:

Functions and its use: Function is a group of related statements that perform a specific task. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable. It avoids repetition and makes code reusable.

Basically, we can divide functions into the following two types:

1. **Built-in functions** - Functions that are built into Python.

Ex: abs(),all().ascii(),bool().....so on....

```
integer = -20
```

```
print('Absolute value of -20 is:', abs(integer))
```

Output:

Absolute value of -20 is: 20

2. **User-defined functions** - Functions defined by the users themselves.

```
def add_numbers(x,y):  
    sum = x + y  
    return sum
```

```
print("The sum is", add_numbers(5, 20))
```

Output:

The sum is 25

Flow of Execution:

1. The order in which statements are executed is called the flow of execution
2. Execution always begins at the first statement of the program.
3. Statements are executed one at a time, in order, from top to bottom.
4. Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
5. Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.

Note: When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the def statements as you are scanning from top to bottom, but you should skip the statements of the function definition until you reach a point where that function is called.

Example:

#example for flow of execution

```
print("welcome")
for x in range(3):
    print(x)
print("Good morning college")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py

welcome

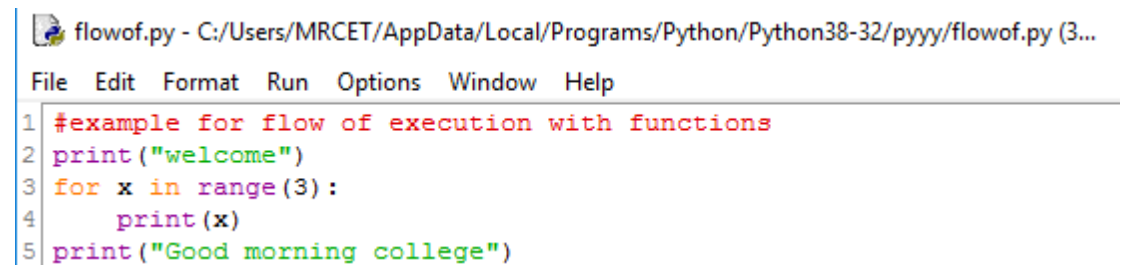
0

1

2

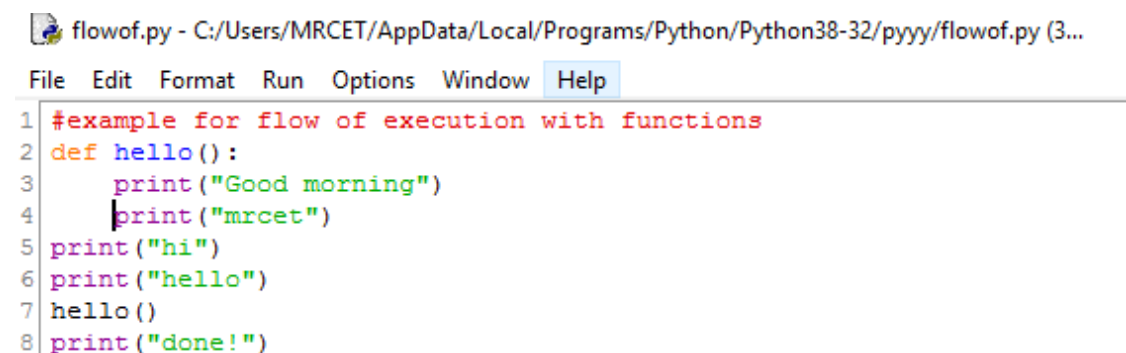
Good morning college

The flow/order of execution is: 2,3,4,3,4,3,4,5



The screenshot shows a Python IDE window titled 'flowof.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py (3...'. The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code editor contains the following code:

```
1 #example for flow of execution with functions
2 print("welcome")
3 for x in range(3):
4     print(x)
5 print("Good morning college")
```



The screenshot shows a Python IDE window titled 'flowof.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py (3...'. The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code editor contains the following code:

```
1 #example for flow of execution with functions
2 def hello():
3     print("Good morning")
4     print("mrcet")
5 print("hi")
6 print("hello")
7 hello()
8 print("done!")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py

hi

hello

Good morning

mrcet

done!

The flow/order of execution is: 2,5,6,7,2,3,4,7,8

Parameters and arguments:

Parameters are passed during the definition of function while Arguments are passed during the function call.

Example:

#here a and b are parameters

```
def add(a,b):  
    return a+b
```

#12 and 13 are arguments

#function call

```
result=add(12,13)
```

```
print(result)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/paraarg.py

25

There are three types of Python function arguments using which we can call a function.

1. Default Arguments
2. Keyword Arguments
3. Variable-length Arguments

Syntax:

```
def functionname():
```

statements

·
·
·

functionname()

Function definition consists of following components:

1. Keyword **def** indicates the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A **colon (:)** to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional return statement to return a value from the function.

Example:

```
def hf():
```

```
    hello world
```

```
hf()
```

In the above example we are just trying to execute the program by calling the function. So it will not display any error and no output on to the screen but gets executed.

To get the statements of function need to be use print().

#calling function in python:

```
def hf():
```

```
    print("hello world")
```

```
hf()
```

Output:

```
hello world
```

```
-----
```

```
def hf():
```

```
    print("hw")
```

```
    print("gh kfjg 66666")
```

```
hf()
```

```
hf()
```

```
hf()
```

Output:

```
hw
```

```
gh kfjg 66666
```

```
hw
```

```
gh kfjg 66666
```

```
hw
```

```
gh kfjg 66666
```

```
def add(x,y):
```

```
    c=x+y
```

```
    print(c)
```

```
add(5,4)
```

Output:

```
9
```

```
def add(x,y):
```

```
    c=x+y
```

```
    return c
```

```
print(add(5,4))
```

Output:

```
9
```

```
def add_sub(x,y):  
    c=x+y  
    d=x-y  
    return c,d  
print(add_sub(10,5))
```

Output:

(15, 5)

The **return** statement is used to exit a function and go back to the place from where it was called. This statement can contain expression which gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the **None** object.

```
def hf():  
    return "hw"  
print(hf())
```

Output:

hw

```
def hf():  
    return "hw"  
hf()
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu.py

>>>

```
def hello_f():  
    return "hellocollege"  
print(hello_f().upper())
```

Output:

HELLOCOLLEGE

Passing Arguments

```
def hello(wish):  
    return '{}'.format(wish)  
print(hello("mrcet"))
```

Output:

mrcet

Here, the function wish() has two parameters. Since, we have called this function with two arguments, it runs smoothly and we do not get any error. If we call it with different number of arguments, the interpreter will give errors.

```
def wish(name,msg):  
    """This function greets to  
    the person with the provided message"""  
    print("Hello",name + ' ' + msg)  
wish("MRCET","Good morning!")
```

Output:

Hello MRCET Good morning!

Below is a call to this function with one and no arguments along with their respective error messages.

```
>>> wish("MRCET")  # only one argument
TypeError: wish() missing 1 required positional argument: 'msg'
>>> wish()  # no arguments
TypeError: wish() missing 2 required positional arguments: 'name' and 'msg'
```

```
-----

def hello(wish,hello):

    return "hi" '{} {}'.format(wish,hello)

print(hello("mrcet","college"))
```

Output:

```
himrcet,college
```

#Keyword Arguments

When we call a function with some values, these values get assigned to the arguments according to their position.

Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed.

(Or)

If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them - this is called **keyword arguments** - we use the name (keyword) instead of the position (which we have been using all along) to specify the arguments to the function.

There are two *advantages* - one, using the function is easier since we do not need to worry about the order of the arguments. Two, we can give values to only those parameters which we want, provided that the other parameters have default argument values.

```
def func(a, b=5, c=10):
    print 'a is', a, 'and b is', b, 'and c is', c
```



```
func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Output:

```
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

Note:

The function named func has one parameter without default argument values, followed by two parameters with default argument values.

In the first usage, func(3, 7), the parameter a gets the value 3, the parameter b gets the value 5 and c gets the default value of 10.

In the second usage func(25, c=24), the variable a gets the value of 25 due to the position of the argument. Then, the parameter c gets the value of 24 due to naming i.e. keyword arguments. The variable b gets the default value of 5.

In the third usage func(c=50, a=100), we use keyword arguments completely to specify the values. Notice, that we are specifying value for parameter c before that for a even though a is defined before c in the function definition.

For example: if you define the function like below

```
def func(b=5, c=10,a): # shows error : non-default argument follows default argument
```

```
def print_name(name1, name2):
```

```
    """ This function prints the name """
```

```
    print (name1 + " and " + name2 + " are friends")
```

```
#calling the function
```

```
print_name(name2 = 'A',name1 = 'B')
```

Output:

B and A are friends

#Default Arguments

Function arguments can have default values in Python.

We can provide a default value to an argument by using the assignment operator (=)

```
def hello(wish,name='you'):
    return '{},{ {}'.format(wish,name)

print(hello("good morning"))
```

Output:

good morning,you

```
def hello(wish,name='you'):
    return '{},{ {}'.format(wish,name)    //print(wish + ' ' + name)

print(hello("good morning","nirosha")) // hello("good morning","nirosha")
```

Output:

good morning,nirosha // good morning nirosha

Note: Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

This means to say, non-default arguments cannot follow default arguments. For example, if we had defined the function header above as:

```
def hello(name='you', wish):
```

Syntax Error: non-default argument follows default argument

```
def sum(a=4, b=2): #2 is supplied as default argument
```

```
""" This function will print sum of two numbers

    if the arguments are not supplied

    it will add the default value """

print (a+b)

sum(1,2) #calling with arguments

sum( )  #calling without arguments
```

Output:

3

6

Variable-length arguments

Sometimes you may need more arguments to process function then you mentioned in the definition. If we don't know in advance about the arguments needed in function, we can use variable-length arguments also called arbitrary arguments.

For this an asterisk (*) is placed before a parameter in function definition which can hold non-keyworded variable-length arguments and a double asterisk (**) is placed before a parameter in function which can hold keyworded variable-length arguments.

If we use one asterisk (*) like *var, then all the positional arguments from that point till the end are collected as a tuple called 'var' and if we use two asterisks (**) before a variable like **var, then all the positional arguments from that point till the end are collected as a dictionary called 'var'.

```
def wish(*names):
    """This function greets all
    the person in the names tuple."""

    # names is a tuple with arguments
    for name in names:
        print("Hello",name)

wish("MRCET","CSE","SIR","MADAM")
```

Output:

Hello MRCET
Hello CSE
Hello SIR
Hello MADAM

#Program to find area of a circle using function use single return value function with argument.

```
pi=3.14
def areaOfCircle(r):

    return pi*r*r
r=int(input("Enter radius of circle"))

print(areaOfCircle(r))
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ful.py
Enter radius of circle 3
28.259999999999998

#Program to write sum different product and using arguments with return value function.

```
def calculate(a,b):

    total=a+b

    diff=a-b

    prod=a*b

    div=a/b

    mod=a%b
```

```
    return total,diff,prod,div,mod

a=int(input("Enter a value"))
b=int(input("Enter b value"))

#function call
s,d,p,q,m = calculate(a,b)

print("Sum= ",s,"diff= ",d,"mul= ",p,"div= ",q,"mod= ",m)

#print("diff= ",d)

#print("mul= ",p)

#print("div= ",q)

#print("mod= ",m)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
Enter a value 5
Enter b value 6
Sum= 11 diff= -1 mul= 30 div= 0.8333333333333334 mod= 5
```

#program to find biggest of two numbers using functions.

```
def biggest(a,b):
    if a>b :
        return a
    else :
        return b

a=int(input("Enter a value"))
b=int(input("Enter b value"))
#function call
big= biggest(a,b)
print("big number= ",big)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ful.py

Enter a value 5

Enter b value-2

big number= 5

#program to find biggest of two numbers using functions. (nested if)

```
def biggest(a,b,c):
```

```
    if a>b :
```

```
        if a>c :
```

```
            return a
```

```
        else :
```

```
            return c
```

```
    else :
```

```
        if b>c :
```

```
            return b
```

```
        else :
```

```
            return c
```

```
a=int(input("Enter a value"))
```

```
b=int(input("Enter b value"))
```

```
c=int(input("Enter c value"))
```

```
#function call
```

```
big=      biggest(a,b,c)
```

```
print("big number= ",big)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ful.py

Enter a value 5

Enter b value -6

Enter c value 7

big number= 7

#Writer a program to read one subject mark and print pass or fail use single return values function with argument.

```
def result(a):
```

```
    if a>40:
```

```
        return "pass"
```

```
else:  
    return "fail"  
a=int(input("Enter one subject marks"))  
  
print(result(a))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ful.py  
Enter one subject marks 35  
fail
```

#Write a program to display mrcet cse dept 10 times on the screen. (while loop)

```
def usingFunctions():  
    count =0  
    while count<10:  
        print("mrcet cse dept",count)  
        count=count+1  
  
usingFunctions()
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ful.py  
mrcet cse dept 0  
mrcet cse dept 1  
mrcet cse dept 2  
mrcet cse dept 3  
mrcet cse dept 4  
mrcet cse dept 5  
mrcet cse dept 6  
mrcet cse dept 7  
mrcet cse dept 8  
mrcet cse dept 9
```