Introduction to Python Progg.

SYLLABUS

Module – 1 : Why should you learn to write programs, Variables, expressions and statements, Conditional execution, Functions **–8 Hours**

Module – 2 : Iteration, Strings, Files **–8 Hours**

Module – 3 : Lists, Dictionaries, Tuples, Regular Expressions**–8 Hours**

Module – 4 : Classes and objects, Classes and functions, Classes and methods**–8 Hours**

Module – 5 : Networked programs, Using Web Services, Using databases and SQL**–8 Hours**

Python Introduction

- Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language.
- It was created by *Guido van Rossum* during 1985- 1990. Python is named after a TV Show called '*Monty Python's Flying Circus*' and not after Python-the snake.

Features of Python

- Simple
- Easy to Learn
- Free and Open Source
- High-level Language
- Python is a Beginner's Language
- Portable
- Interactive
- Interpreted
- Object Oriented
- Extensible
- Embeddable
- Extensive Libraries
- Databases
- GUI Programming
- Scalable

Programming Editors

- Python 3.6 Command Prompt
- Python 3.6 IDLE
- Anaconda Prompt
- Anaconda Jupyter Notebook
- Anaconda Spider

Some Python Packages/Libraries

- Numpy
- Scipy
- Pandas
- Matplotlib
- Seaborn
- Bokeh
- Scikit Learn
- Pygames
- dJango
- Flask
- Bottle
- Pyramid
- PyBrain
- PyMongo
- Tornado
- Web2py
- Json
- tKinter
- OpenCV
- PyGObject
- PyQt
- wxPython
- Kivy
- Buildozer
- Buildbot
- Trac
- Roundup
- Ansible
- Salt
- OpenStack
- Keras
- Tensor Flow
- Theano
- nltk
- Spacy
- TextBlog
- ScikitLearn
- Pattern
- SQLalchemy
- pyMySql
-

Different Modes

1. Interactive / Immediate mode (Command Prompt)
2. Script mode (Idle)
3. IDE mode (Jupyter , Spyder and Pycharm)

Module 1

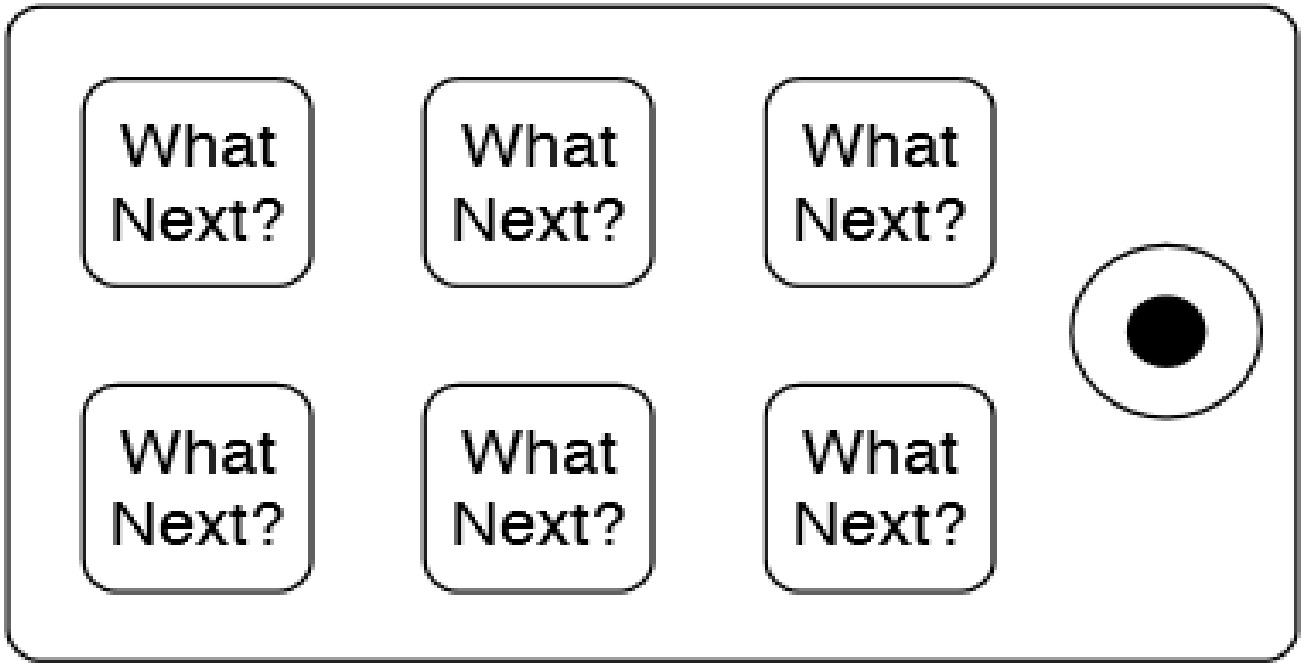
M1.1 Why should you learn to write programs,

M1.2 Variables, expressions and statements,

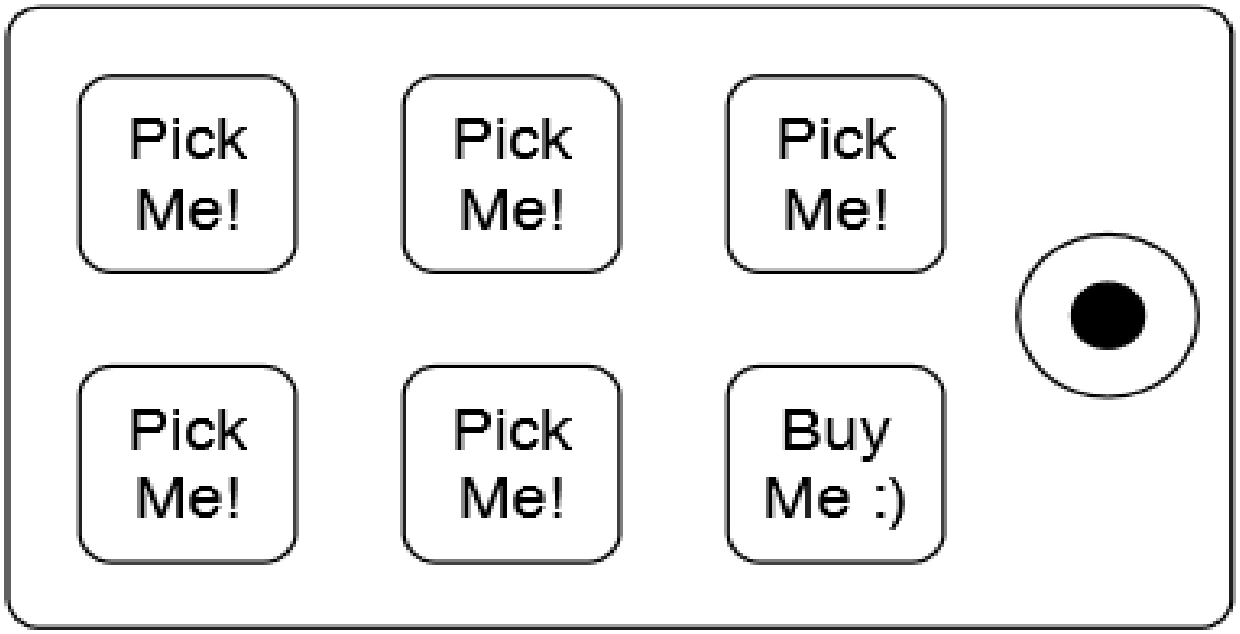
M1.3 Conditional execution,

M1.4 Functions

Why should you learn to Write programs



Creativity and

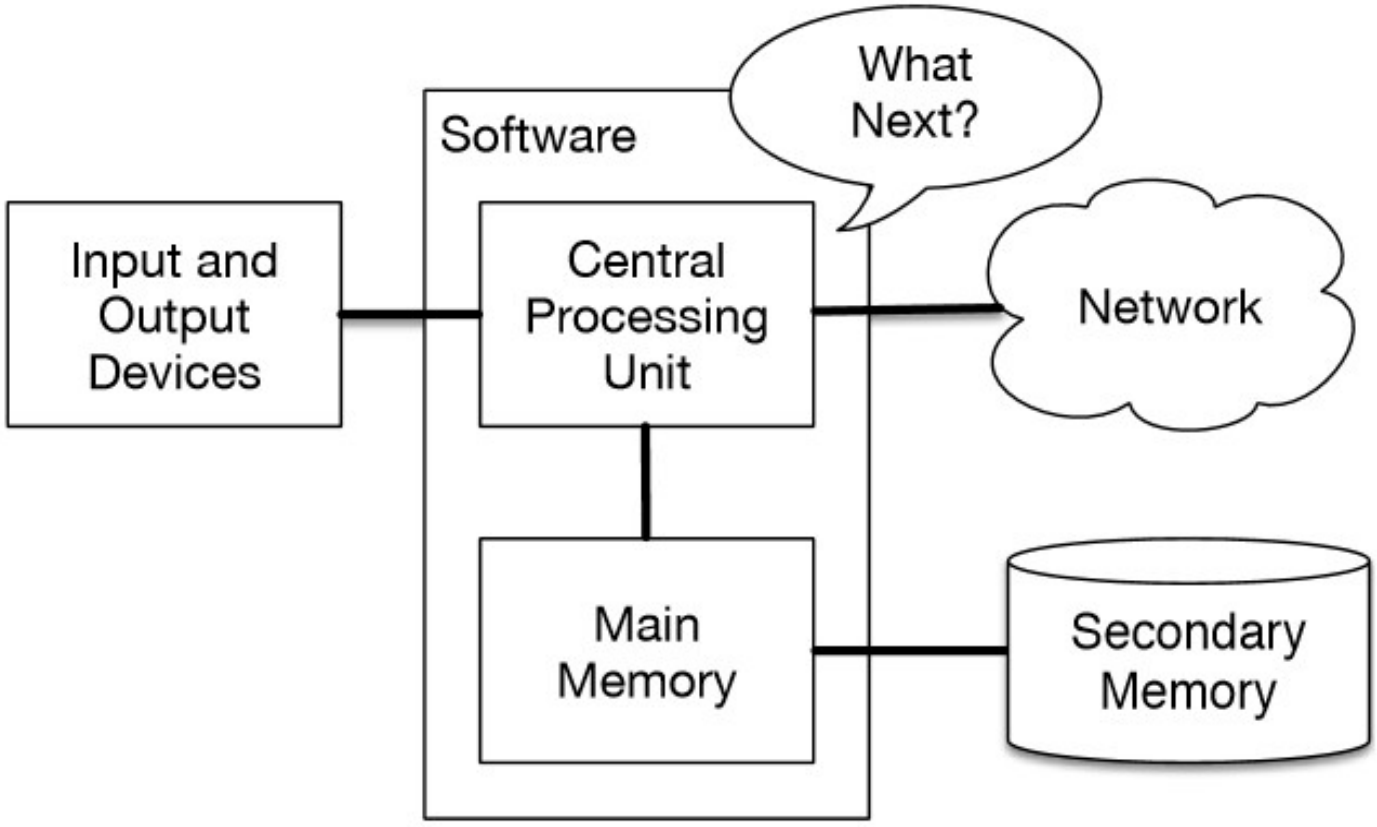


Programmers Talking to You

Question

–With a neat diagram explain the High level definition of the parts of Hardware Architecture. Explain the role of Programmer.

Computer hardware

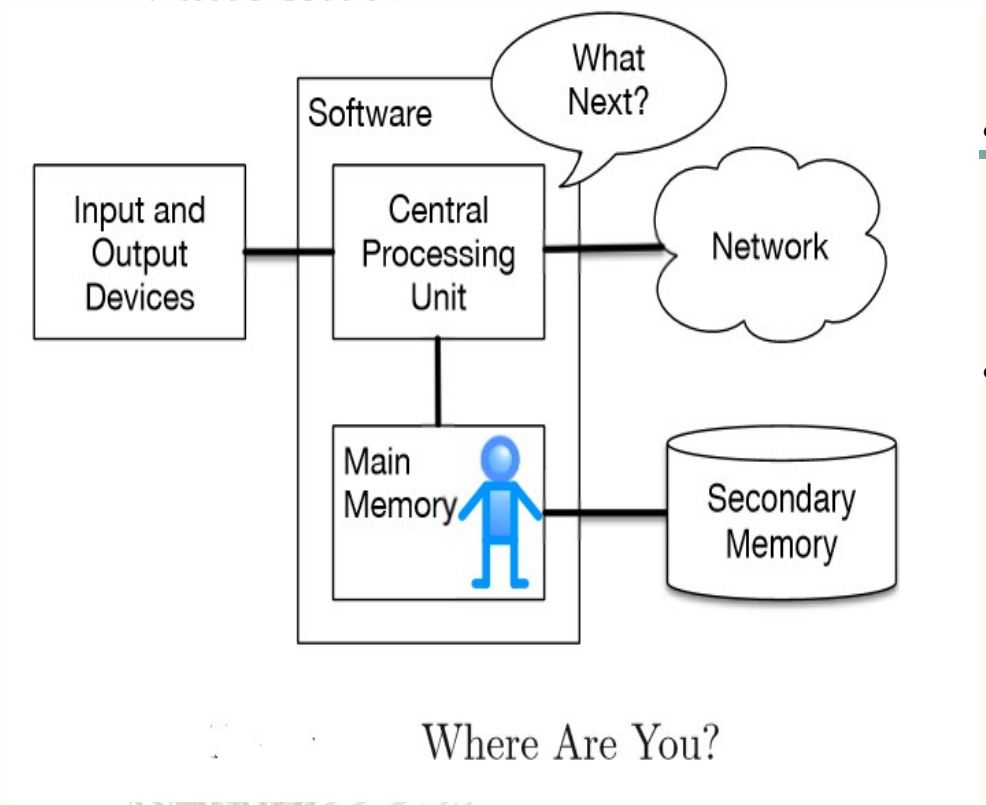


Hardware Archicture

Different Parts of Hardware architecture and their High Level Definitions:

- **The Central Processing Unit (or CPU)** is the part of the computer that is built to be obsessed with "what is next?" If your computer is rated at **3.0 Gigahertz**, it means that the CPU will ask "What next?" three billion times per second. You are going to have to learn how to talk fast to keep up with the CPU.
- **The Main Memory** is used to store information that the CPU needs in a hurry. The main memory is nearly as fast as the CPU. But the information stored in the main memory vanishes when the computer is turned off.
- **The Secondary Memory** is also used to store information, but it is much slower than the main memory. The advantage of the secondary memory is that it can store information even when there is no power to the computer. Examples of secondary memory are disk drives or flash memory (typically found in USB sticks and portable music players).
- **The Input and Output Devices** are simply our screen, keyboard, mouse, microphone, speaker, touchpad, etc. They are all of the ways we interact with the computer.
- These days, most computers also have a Network Connection to retrieve information over a network. We can think of the network as a very slow place to store and retrieve data that might not always be "up". So in a sense, the network is a slower and at times unreliable form of Secondary Memory.

Job of Programmer



- To use and orchestrate each of these resources to solve the problem that you need to solve and analyze the data you get from the solution.
- "Talking" to the CPU and telling it what to do next. Sometimes programmer will tell the CPU to use the main memory, secondary memory, network, or the input/output devices.

What is Program ?

- A set of stored instructions that specifies a computation.

What is Programming ?

- The act of writing these instructions down and getting the instructions to be correct programming

Who is Programmer?

- A person who is skilled in the art of programming

Two skills to be a programmer

- **First, you need to know the programming language (Python)** - you need to know the vocabulary and the grammar. You need to be able to spell the words in this new language properly and know how to construct well-formed "sentences" in this new language.
- **Second, you need to "tell a story".** In writing a story, you *combine words and sentences to convey an idea to the reader.* There is a skill and art in constructing the story, and skill in story writing is improved by doing some writing and getting some feedback. In programming, **our program is the "story"** and the problem you are trying to solve is the **"idea".**

Words

•Word is a ***fundamental unit of python programming language*** . Word may be reserved or unreserved words. The Python vocabulary is called the "**reserved words**". These are words that have very special meaning to Python.

•**Variables** are the words made by the programmer as per need based.

Keywords				
False	**class**	**finally**	**is**	**return**
None	**continue**	**for**	**lambda**	**try**
True	**def**	**from**	**nonlocal**	**while**
and	**del**	**global**	**not**	**with**
as	**elif**	**if**	**or**	**yield**
assert	**else**	**import**	**pass**	
break	**except**	**in**	**raise**	

Sentence

- A sentence is a set of words , expressions and symbols which is syntactically correct .
- Example : ***print('Hello world!')***

Conversing with Python

```
>>> print(‘ My Name is Mr.XYZ')
>>> print(‘Student at SDMIT')
>>> print ‘UJIRE
>>> good-bye
```

One can converse with python only through the syntactically correct statements .

What is High Level Language ?

- A programming language like Python that is designed to be easy for humans to read and write.
- Examples : **Java, C++, PHP, Ruby, Basic, Perl, JavaScript**, and many more
- *The actual hardware inside the Central Processing Unit (CPU) does not understand any of these high-level languages.*

What is Machine Language ?

- The lowest-level language for software, which is the language that is directly executed by the central processing unit (CPU).
- Machine language is very simple and frankly very tiresome to write because it is represented all in zeros and ones:
- 00101000111010010010101000001111 11100110000011101010010101101101 ...

Portability

- A property of a program that can run on more than one kind of computer.
- Since machine language is tied to the computer hardware, machine language is not portable across different types of hardware. Programs written in high-level languages can be moved between different computers by using a different interpreter on the new machine or recompiling the code to create a machine language version of the program for the new machine.

Programming language translators

These programming language translators fall into two general categories:

(1) Interpreters and

(2) Compilers.

1. Interpreter

- An ***interpreter*** reads the source code of the program as written by the programmer, parses the source code, and interprets the instructions on the fly.
- **Python is an interpreter** and when we are running Python interactively, we can type a line of Python (a sentence) and Python processes it immediately and is ready for us to type another line of Python.

Example :

```
>>> x = 6
>>> print(x)
 6
>>> y = x * 7
>>> print(y)
42
>>>
```

2.Compiler

- It translates a program written in a high-level language into a ***low-level language all at once***, in preparation for later execution.
- A compiler needs to be handed the entire program in a file, and then it runs a process to translate the high-level source code into machine language and then the compiler puts the resulting machine language into a file for later execution.
- In Windows, the executable machine code for Python itself is likely in a file with a name like:
- **C:\Python35\python.exe**

Exercise

- Create a file called samp.txt in *jupyter* and enter only plain text without containing any special character. Save the file.

Execute the following program

```
name = input('Enter file:')
handle = open(name,'r')
counts = dict()
for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1
bigcount = None
bigword = None
for word, count in list(counts.items()):
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count
print(bigword, bigcount)
```

What are the building blocks of programs?

1. **INPUT :** Get data from the "outside world". This might be reading data from a file, or even some kind of sensor like a microphone or GPS. In our initial programs, our input will come from the user typing data on the keyboard.
2. **OUTPUT :** Display the results of the program on a screen or store them in a file or perhaps write them to a device like a speaker to play music or speak text.
3. **SEQUENTIAL EXECUTION :** Perform statements one after another in the order they are encountered in the script.
4. **CONDITIONAL EXECUTION :** Check for certain conditions and then execute or skip a sequence of statements.
5. **REPEATED EXECUTION :** Perform some set of statements repeatedly, usually with some variation.
6. **REUSE :** Write a set of instructions once and give them a name and then reuse those instructions as needed throughout your program.

Three general types of errors:

1. **Syntax errors :** These are the first errors you will make and the easiest to fix. A syntax error means that you have violated the "grammar" rules of Python.
2. **Logic errors:** A logic error is when your program has good syntax but there is a mistake in the order of the statements or perhaps a mistake in how the statements relate to one another.
3. **Semantic errors:** A semantic error is when your description of the steps to take is syntactically perfect and in the right order, but there is simply a mistake in the program. The program is perfectly correct but it does not do what you intended for it to do.

M1.2 Variables, Expressions and Statements

Ananth G S

What are Values and types?

1. **Value :** A value is a letter or a number.

   Example : 1, 2, and “Hello, World!”

2. **Type** : Types are the data types to which the Values belong :

   Example : 2 is an integer, “Hello, World!” is a string and 3.3 is a float

   type(arg) function returns the data type of the argument

Examples

```
>>> type('Hello, World!')
<class 'str'>
>>> type(17)
<class 'int'>
>>> type(3.2)
<class 'float'>
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
>>> print(1,000,000)
1 0 0
>>> print(4)
4
```

What are the different Data Types?

1. **Numbers:** Number data types store numeric values. Number objects are created when you assign a value to them.
2. **Strings:** Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows either ***pair of single or double quotes***.
3. **Lists:** Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed ***within square brackets ([ ]).***
4. **Tuples:** A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, ***tuples are enclosed within parenthesis.***
5. **Dictionary:** Python's dictionaries are kind of hash-table type. They work like associative arrays or hashes found in Perl and consist of **key-value** pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object. ***Dictionaries are enclosed within curly braces***.

Numbers

- Python recognizes several different types of data numbers.
  - 23 and −75 are *integers*,
  - 5.0 and −23.09 are floats *or floating point numbers.*
  - 2 + 3j is a *complex number*

To check the number type

- type(-75)
- type(5.0)
- type(12345678901)
- type(-1+2j)
- complex(2,3)
- type("This is a string")
- type([1,3,4,1,6])
- type(True)
- type(False)

What are Variables ?

Variables

- **A variable is a name that refers to a value.**

  Example : x, si , area_of _Circle ,etc

- **An assignment statement creates new variables and gives them values**

  Example :

  Message = '*Python Programming* '
  p =1000
  t= 2
  r=3.142
  Si = p*t*r/100
  pi = 3.1415926535897931
  area_of _circle = *pi*r*r*

- **To know the type of the variable one can use type() function** . Ex: type(p)

- **To display the value of a variable , you can use a print statement** :

Ex:

  - **print (Si)**
  - **print(pi)**

Rules for writing Variable names

1. **Variable names** can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_).
2. **Variable names** cannot start with a digit.
3. **Keywords** cannot be used as Variable names .
4. We cannot use special symbols like **!, @, #, $, %** etc. in Variable names .
5. Variable names can be of any length.

Illegal Variable names

- >>> 76trombones = 'big parade'
  - Syntax Error: invalid syntax
- >>> more@ = 1000000
  - Syntax Error: invalid syntax
- >>> class = 'Advanced Theoretical Zymurgy'
  - Syntax Error: invalid syntax

Statements

- A statement is a unit of code that the Python interpreter can execute. Each statement is a set of words (reserved/unreserved and special symbols).
- Types of statements :
  - Expression statement **Ex:** a + 2
  - Assignment statement **Ex:** a = 1
- **Print statement , if** statement, **for** statement, **while** statement etc

Multi-Line Statement

- In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\). For example:

```
a = 1 + 2 + 3 + \
    4 + 5 + 6 + \
    7 + 8 + 9
```

This is explicit line continuation. In Python, line continuation is implied inside parentheses ( ), brackets [ ] and braces { }. For instance, we can implement the above multi-line statement as

```
a = (1 + 2 + 3 +
     4 + 5 + 6 +
     7 + 8 + 9)
```

Here, the surrounding parentheses ( ) do the line continuation implicitly. Same is the case with [ ] and { }. For example:

```
colors = ['red',
          'blue',
          'green']
```

We could also put multiple statements in a single line using semicolons, as follows

```
a = 1; b = 2; c = 3
```

Operators and operands

- Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called operands.

- The operators **+, -, *, /, and **** perform *addition, subtraction, multiplication, division, and exponentiation*, as in the following examples:
  - **20 + 32**
  - **hour - 1**
  - **hour*60 + minute**
  - **minute/60**
  - **5**2**
  - **(5+9)*(15-7)**

Expressions

- An expression is a combination of ***values, variables, and operators***.

- Example :

  17 , x , x + 17 , 1 + 1 , x**2, x**2 + y **2

Order of operations

- When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence.

PEMDAS order of operation is followed in Python :

- **Parentheses** have the highest precedence and can be used to force an expression to evaluate in the order you want.
- **Exponentiation** has the next highest precedence,
- **Multiplication and Division** have the same precedence, which is higher than
- **Addition and Subtraction**, which also have the same precedence.
- **Operators with the same precedence** are evaluated from left to right.

Quotient and Modulus operator

- **// is Quotient operator .**

  Example : 7 // 3 yields 2

- **% is a modulus operator** and works on integers which yields the remainder when the first operand is divided by the second.

  Example : r = 7 % 3 yields 1

String operations with +

- The + operator **perform concatenation** with strings,
- For example:

```
first = '100'
second = '150'
print(first + second)
```

The output of this program is 100150.

Comments

- Adding notes to the programs to explain in natural language about what the program is doing are called comments, and in Python they start with the **#** symbol:
- Example 1:

```
# compute the percentage of the hour
        percentage = (minute * 100) / 60
```

- Example 2:

```
percentage = (minute * 100) / 60      # percentage of an hour
```

- Example 3: (This comment is redundant with the code and useless:)

```
v = 5     # assign 5 to v
```

- Example 4: (This comment contains useful information)
- v = 5 **# velocity in meters/second.**

Asking the user for input

In order to get the input from the user through the keyboard Python provides a built-in function called **input**. When this function is called, the program stops and waits for the user to type something. When the user presses **Return or Enter,** the program resumes and input returns what the user typed as a string.

Example

```
# Program to find the simple interest

p = int(input("Enter the principal amount "))

t = int(input ("Enter the time period"))

r = float(input("Enter the rate of interest"))

si = p*t*r/100

print("The simple interest = ",si)
```

```
Enter the principal amount 1000
Enter the time period 2
Enter the rate of interest 2.56
The simple interest =  51.2
```

Python Program to Add Two Numbers

- **# This program adds two numbers**
- num1 = 1.5
- num2 = 6.3
- **# Add two numbers**
- sum = float(num1) + float(num2)
- **# Display the sum**
- print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))

```python
# This program adds two numbers
num1 = 1.5
num2 = 6.3
# Add two numbers
sum = float(num1) + float(num2)
# Display the sum
print('\nThe sum of {0} and {1} is {2}'.format(num1, num2, sum))
```

```
The sum of {0} and {1} is {2} 1.5 6.3 7.8

The sum of 1.5 and 6.3 is 7.8
```

Add Two Numbers Provided by The User

- # Store input numbers
- num1 = input('Enter first number: ')
- num2 = input('Enter second number: ')
- # Add two numbers
- sum = float(num1) + float(num2)
- # Display the sum
- print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))

```python
# Store input numbers
num1 = input('Enter first number: ')
num2 = input('Enter second number: ')
# Add two numbers
sum = float(num1) + float(num2)
# Display the sum
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```

```
Enter first number: 2
Enter second number: 3
The sum of 2 and 3 is 5.0
```

Exercise

- Assume that we execute the following assignment statements:

  width = 17 height = 12.0

For each of the following expressions, write the value of the expression and the type (of the value of the expression).

1. width//2
2. width/2.0
3. height/3
4. 1 + 2 * 5

Exercise

1. Write a program that uses input to prompt a user for their name and then welcomes them.
2. Write a program to prompt the user for hours and rate per hour to compute gross pay.
3. Write a program to read the following input from user and display

   Name :

   USN :

   Roll NO:

   Mobile No:

   E-Mail id :

   Percentage of Marks :

Choosing mnemonic/ meaningful variable names

```
a = 35.0
b = 12.50
c = a * b
print(c)
```

```
hours = 35.0
rate = 12.50
pay = hours * rate
print(pay)
```

```
x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

Examples

```
for word in words:
    print(word)

for slice in pizza:
    print(slice)
```

M1.3 Conditional Execution

Explain the following with examples

- Boolean Expression
- Logical Operators
- Conditional Execution : **if**
- Alternative Execution: **if else**
- Chained Conditionals : if **elif** else
- Nested Conditionals : if else **if else**
- Catching exception using **try and except**
- **Short circuit** evaluation of logical circuits

Boolean Expression

- A boolean expression is an expression that is either **true or false**.
- Example :

```
>>> 5 == 5
True
>>> 5 == 6
False
```

The == operator is one of the comparison operators; the others are:

- x != y # x is not equal to y
- x > y # x is greater than y
- x < y # x is less than y
- x >= y # x is greater than or equal to y
- x <= y # x is less than or equal to y
- x **is** y # x is the same as y
- x **is** not y # x is not the same as y

Special operators

- Python language offers some special type of operators *like the identity operator* or the *membership operator*.
- They are described below with examples

Identity operators

- ***is*** and ***is not*** are the identity operators in Python.
- They are used to check ***if two values (or variables) are located on the same part of the memory***.
- Two variables that are equal does not imply that they are identical.

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

Example #4: Identity operators in Python

- x1 = 5
- y1 = 5
- x2 = 'Hello‘
- y2 = 'Hello‘
- x3 = [1,2,3]
- y3 = [1,2,3]
- print(x1 is not y1) **# Output: False**
- print(x2 is y2) **# Output: True**
- print(x3 is y3) **# Output: False**

Membership operators

- **in** and **not** in are the membership operators in Python.
- They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).
- In a dictionary we can only test for presence of key, not the value.

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Example #5: Membership operators in Python

- x = 'Hello world‘
- y = {1:'a',2:'b'}
- print('H' in x) **# Output: True**
- print('hello' not in x) **# Output: True**
- print(1 in y) **# Output: True**
- print('a' in y) **# Output: False**

Logical operators

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Example #3: Logical Operators in Python

- x = True
- y = False
- print('x and y is', **x and y**) # Output: x and y is False
- print('x or y is', ***x or y***) # Output: x or y is True
- print('not x is', ***not x***) # Output: not x is False

Summary 2

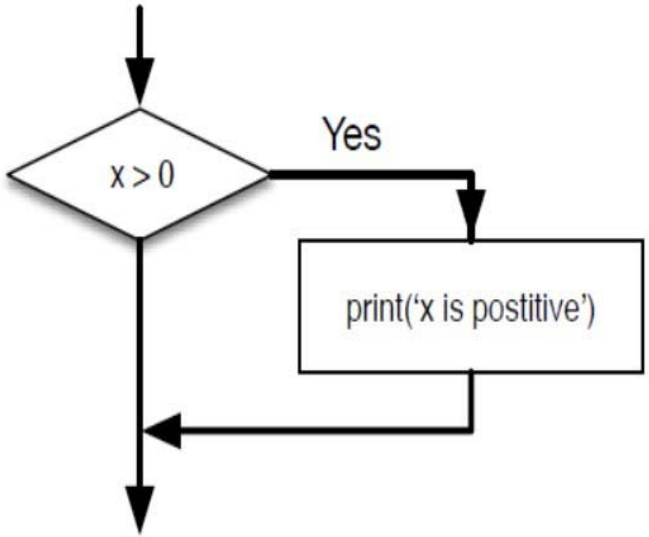
- Why one Should write Program?
- Program , Programming , Hardware Architecture of Computer, Interpreter and Compiler.
- Building Blocks of Program
- Types of Errors
- Variables , Expressions and Statements , order of operations
- Reading input from the user through keyboard
- Special Operators (identity (is , is not) and membership (in , not in) operators)
- Boolean Expressions , Logical Operators
- Conditional Executions (*Conditional Execution :* ***if ,*** *Alternative Execution:* ***if else*** *Chained Conditionals : if* ***elif*** *else , Nested Conditionals : if else* ***if else*** *Catching exception using* ***try and except*** ***and Short circuit*** *evaluation of logical circuits*)

Conditional execution

- The simplest form is the conditional if statement:

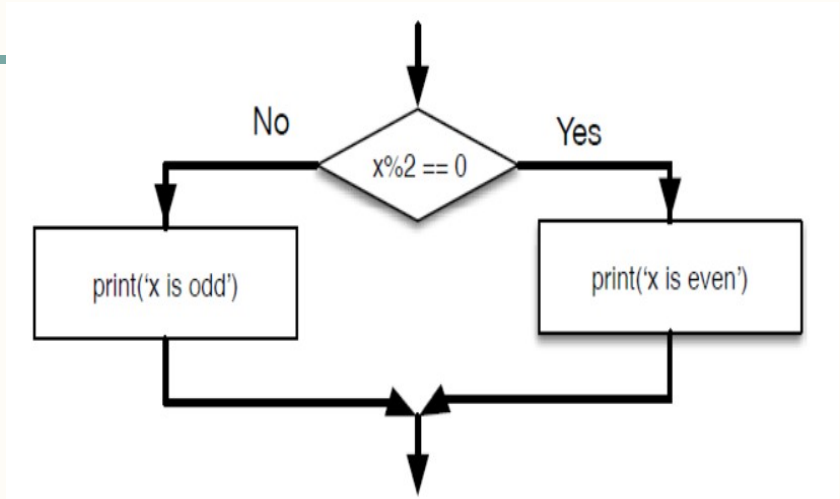
```
if x > 0 :
    print('x is positive')
```

- The boolean expression after the if statement is called the *condition. We end the*
- *if* statement with a colon character *(:)* and the *line(s)* after the *if* statement are indented.



Alternative execution

- A second form of the **if statement** is *alternative execution, in which there are two* possibilities and the condition determines which one gets executed. The syntax looks like this:




```
if x%2 == 0 :
    print('x is even')
else :
    print('x is odd')
```

Exercise

1. Write a program to prompt the user ***for hours and rate per hour*** to compute gross pay. Also to give the employee 1.5 times the hourly rate for hours worked above 40 hours.

Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a *chained conditional:*

```
if  x < y:
        print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

Note : elif is an abbreviation of “else if.” Again, exactly one branch will be executed.

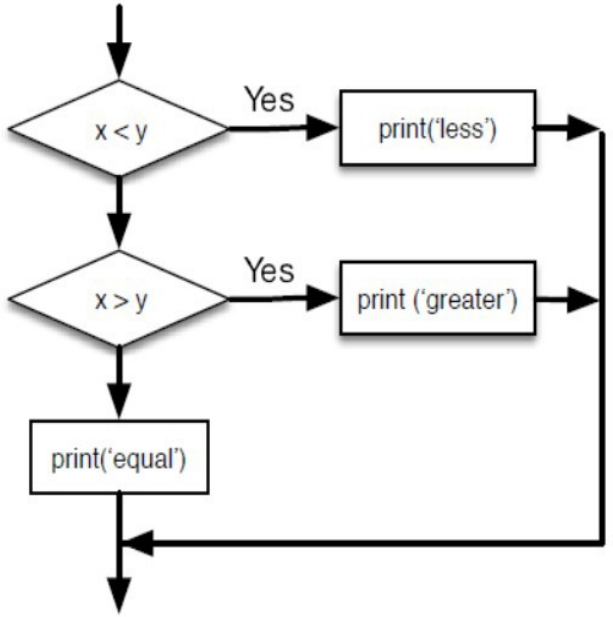


Figure 3.3: If-Then-ElseIf Logic

Note

- There is no limit on the number of ***elif*** statements.
- If there is an *else clause*, it has to be at the end, but there doesn't have to be one.

Example

```
if  choice == 'a':
        print('Bad guess')
elif  choice == 'b':
        print('Good guess')
elif  choice == 'c':
        print('Close, but not correct')
```

- Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends.
- Even if more than one condition is true, only the first true branch executes

Exercise

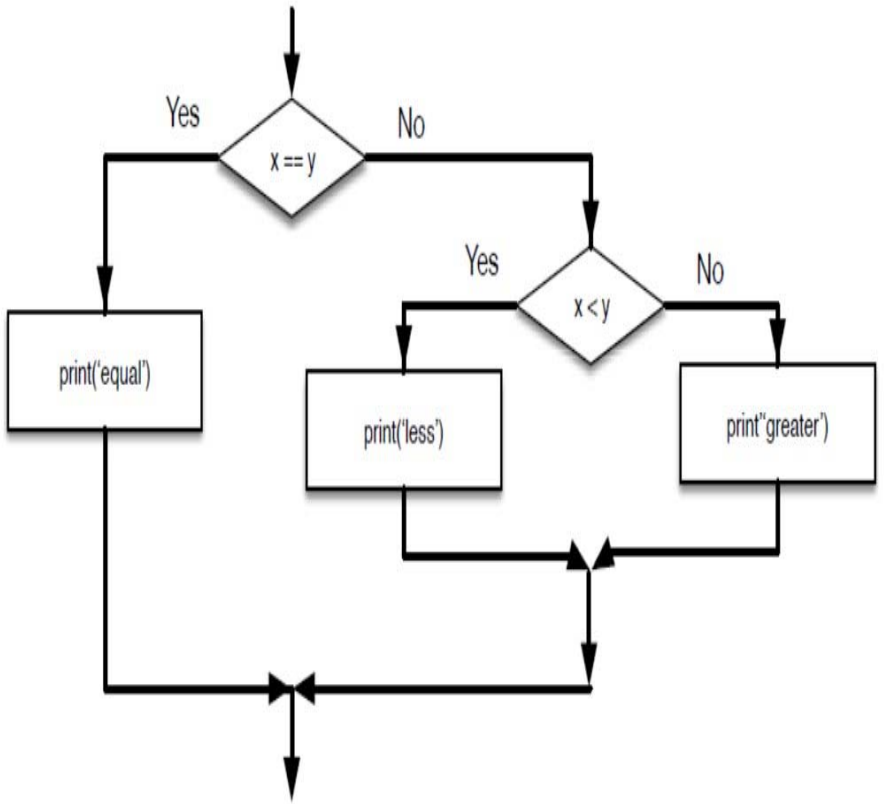
Write a program to prompt for a score between 0.0 and 1.0. If the score is out of range, print an error message.

[*Score Grade >= 0.9 A >= 0.8 B >= 0.7 C >= 0.6 D < 0.6 F*]

Nested conditionals

- One condition can also be nested within another. We could have written the three-branch example like this:

```
if x == y:
        print('x and y are equal')
else:
        if x < y:
        print('x is less than y')
       else:
       print('x is greater than y')
```



Note

- Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if  0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

- The print statement is executed only if we make it past both conditionals, so we can get the same effect with the and operator:

```
if 0 < x and x < 10:
print('x is a positive single-digit number.')
```

Catching exceptions using try and except

- There is a conditional execution structure built into Python to handle these types of expected and unexpected errors called "*try / except*".
- The idea of try and except is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs.
- These extra statements (the except block) are ignored if there is no error.

Example

```
inp = input('Enter Fahrenheit Temperature:')
 try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
   print('Please enter a number')
```

Short-circuit evaluation of logical expressions

- x = 6
- y = 0
- x >= 2 and y != 0 and (x/y) > 2

- **x = 6**
- **y = 0**
- **x >= 2 and (x/y) > 2 and y != 0**

Exercise

1. Write a program to prompt the user ***for hours and rate per hour*** to compute gross pay. Also to give the employee 1.5 times the hourly rate for hours worked above 40 hours.
2. Rewrite the above program using try and except so that your program handles non-numeric input gracefully by printing a message and exiting the program.

M1.4 Functions

Explain the following with example

1. **Functions**
2. **Function Call**
3. **Built in Function**
4. **Type Conversion Functions**
5. **Random Numbers**
6. **Math Functions**
7. **Adding new Functions**
8. **Defining and using the new functions**
9. **Flow of execution**
10. **Parameter and Arguments**
11. **Fruitful and void Functions**
12. **Advantages of Functions**

Functions

- In the context of programming, a *function is a named sequence of statements that* performs a computation to do particular task.
- **Syntax for Defining a function :**

```
def  fun_name(parameter list):
        stmt1
        stmt2
        stmt3
        -------
        stmtn
        return stmt
```

- **Syntax for calling a function** :

```
fun_name(parameter list)
```

Parameters and arguments

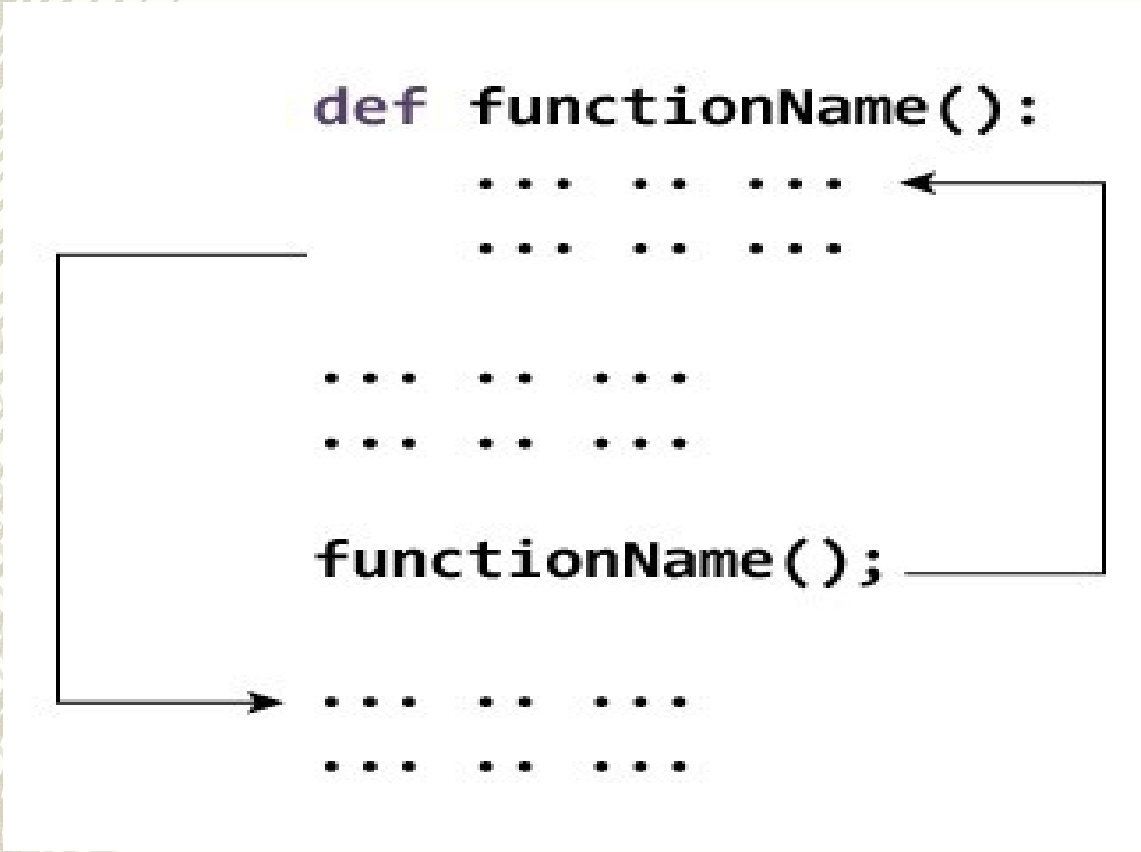
- **Parameters are temporary variable names within functions**. The argument can be thought of as the value that is assigned to that temporary variable.

- **Function Definition :**

```
def   cube(number):
      return   number**3
```

- '**number**' here is the parameter for the function 'cube'. This means that anywhere we see 'number' within the function will act as a placeholder until number is passed an argument.
- **Function Call** :
  - **cube(3)**
- Here **3** is the argument. The parameters are used as place holder or arguments.
- **Parameters are used in function definition and arguments are used in function call.**

Working of function



Example 1:

```
# Defining a Function
def  print_Address():
     print(“Rajashree")
     print(‘ D.NO 07 ')
     print (‘ Ajit Nagara’)
     print(‘Ujire -574240’)
# Function Call
print_Address()
```

Output

Rajashree

D.NO 07

Ajit Nagara

Ujire -574240

Example 2 :

```
def  print_lyrics():
    print(“First Line of   a Song")
    print(‘Second Line of a Song')

def  repeat_lyrics():
        print_lyrics()
        print_lyrics()

repeat_lyrics()
```

Output

```
First Line of a Song
Second Line of a Song
First Line of a Song
Second Line of a Song
```

Example 3

- def print_twice(bruce):
  - print(bruce)
  - print(bruce)

- print_twice('Spam')

print_twice(17)

import math

- Output

Spam

Spam 17

- 17
- 3.141592653589793
- 3.141592653589793
- Spam Spam Spam Spam

Spam Spam Spam Spam

- -1.0 -1.0

Example

- michael = 'Eric, the half a bee.'
- print_twice(michael)

Output :
Eric, the half a bee.
Eric, the half a bee.

Types of Functions based on return

- void Function
- fruitful Function

void function does not return a value

```
def  print_name():
     print(“RAHUL MODI”);
```

Fruitful functions *returns a value*

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)
```

Output

8

Example for Fruitful Functions

- radians =30
- x = math.cos(radians)
- math.sqrt(5)

  2.23606797749979
- golden = (math.sqrt(5) + 1) / 2

Why functions?

- Function gives you an opportunity to name a group of statements,
- Which makes your program easier to read, understand, and debug.
- Functions can make a program smaller by eliminating repetitive code.
- Dividing a long program into functions allows to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it

Categories of Functions

1. Built in Functions : Functions that are built into Python
2. User Defined Functions :Functions defined by the users themselves.

1.Built in Functions

- The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, print() function prints the given object to the standard output device (screen) or to the text stream file.
- In Python 3.6 (latest version), there are 68 built-in functions.
- Examples : print(),len(),max(),min(),input(),help() ,abs(),flo at(),int(),hex(),bin(),oct()

Built-in functions

- Python provides a number of important built-in functions that we can use without needing to provide the function definition. The creators of Python wrote a set of functions to solve common problems and included them in Python for us to use.

- **max**('Hello world')
  w
- **min(**'Hello world')
  ‘ ‘
- **len**('Hello world')
  11

2.User Defined Function

- Functions that we define ourselves to do certain specific task are referred as user-defined functions. Functions that readily come with Python are called built-in functions. If we use functions written by others in the form of library, it can be termed as library functions. All the other functions that we write on our own fall under user-defined functions. So, our user-defined function could be a library function to someone else.
- **Example of a user-defined function**

```
def add_numbers(x,y):
        sum = x + y
        return sum
num1 = 5
num2 = 6
print("The sum is", add_numbers(num1, num2))
```

Math Functions and constants

Function name	Description
`abs(`**value**`)`	absolute value
`ceil(`**value**`)`	rounds up
`cos(`**value**`)`	cosine, in radians
`floor(`**value**`)`	rounds down
`log10(`**value**`)`	logarithm, base 10
`max(`**value1**`, `**value2**`)`	larger of two values
`min(`**value1**`, `**value2**`)`	smaller of two values
`round(`**value**`)`	nearest whole number
`sin(`**value**`)`	sine, in radians
`sqrt(`**value**`)`	square root

Constant	Description
`e`	2.7182818...
`pi`	3.1415926...

- To use these commands, place this line atop your program:

```
from math import *
```

Math functions example

- import math
- print(math)
- ratio = signal_power / noise_power
- decibels = 10 * math.log10(ratio)
- radians = 0.7
- height = math.sin(radians)
- radians = degrees / 360.0 * 2 * math.pi
- math.sin(radians)
- math.sqrt(2) / 2.0

Type conversion functions: used to convert one form to other

- int('32') #string to int
- int('Hello') # string to int
- int(3.99999) # float to int
- int(-2.3) # float to int
- float(32) # int to float
- float('3.14159') # string to float

Random numbers

- The random module provides functions that generate pseudorandom numbers
- To see a sample, run this loop:
- **import random**

```
for i in range(10):
    x = random.random()
    print(x)
```

Output

0.14350103211784326

0.11775378050385343

0.593629658284786

0.8866024608748476

0.6316710639256297

0.871088071436763

0.05665609859315035

0.5586754374930559

0.6952228377929273

0.009905091780000652

Other randoms

- **random.randint(5, 10)**

Output : Any random number between 5 to 10

- **t = [1, 2, 3]**

  random.choice(t)

Output : Any random number between 1 to 3

Exercise

- Write the grade program using a function called compute grade that takes a score as its parameter and returns a grade as a string.

Glossary

- **algorithm** : A general process for solving a category of problems.
- **Argument:** A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.
- **body :**The sequence of statements inside a function definition.
- **composition :**Using an expression as part of a larger expression, or a statement as part of a larger statement.
- **Deterministic:** Pertaining to a program that does the same thing each time it runs, given the same inputs.
- **dot notation:** The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.
- **flow of execution :**The order in which statements are executed during a program run.
- **fruitful function:** A function that returns a value.
- **function :**A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

Glossary

- **function call** A statement that executes a function. It consists of the function name followed by an argument list.
- **function object** A value created by a function definition. The name of the function is a variable that refers to a function object.
- **header** The first line of a function definition.
- **import statement** A statement that reads a module file and creates a module object.
- **module object** A value created by an import statement that provides access to the data and code defined in a module.
- **parameter** A name used inside a function to refer to the value passed as an argument.
- **pseudorandom** Pertaining to a sequence of numbers that appear to be random, but are generated by a deterministic program.
- **return value** The result of a function. If a function call is used as an expression, the return value is the value of the expression.
- **void function** A function that does not return a value.

Exercise 4: What is the purpose of the "def" keyword in Python?

a) It is slang that means "the following code is really cool"

b) It indicates the start of a function

c) It indicates that the following indented section of code is to be stored for later

d) b and c are both true

e) None of the above

What will the following Python program print out?

```
def fred():
    print("Zap")
def jane():
    print("ABC")
jane()
fred()
jane()
```

End of Modul1