**Unit 1:- Introduction to python**

**Chapter 1:-**

**Syllabus: -** **Why should you learn to write programs?**

* Creativity and motivation
* Computer hardware architecture
* Understanding programming
* Words and sentences
* Conversing with Python
* Terminology: Interpreter and compiler
* Writing a program
* What is a program?
* The building blocks of programs
* What could possibly go wrong?
* Debugging
* The learning journey.

**Why should you learn to write programs …?**

* Powerful and general.
* Can hide a poem in a picture.
* Allows people to communicate securely.
* Can find optimal paths in huge maps.

**What is Programing..?**

* A programming language is a formal language comprising a set of instructions that produce various kinds of output.
* Programming languages are used in computer programming to implement algorithms.
* It is vocabulary and a collection of rules that command a computer, devices, and applications to work according to the written codes.
* The programing language enables us to write efficient programs and develop online solutions such as- mobile applications**,** web applications, and games, etc.
* Programming is used to automate, maintain, assemble, measure and interpret the processing of the data and information. It helps in accelerating the input and output of the devices or applications.

**Why programing languages are needed …?**

* To advance your ability to develop real algorithms- Most of the languages come with a lot of features for the Programmers. They can be used in a proper way to get the best results.
* To Improve Customization of Your Current Coding- By using basic features of the existing programming language you can simplify things to program a better option to write resourceful codes. There is no compulsion of writing code in a specific way. The thing which matters is the usage of features used and clarity of the concept.
* To Increase Your Vocabulary Of beneficial Programming Constructs- Programmers use high-level languages to express thoughts. And, by using the best features they can easily explain the working of a specific application, device, etc.

**History of Python**

In the late 1980s, history was about to be written. It was that time when working on Python started. Soon after that, Guido Van Rossum began doing its application based work in December of 1989 by at Centrum Wiskunde & Informatica (CWI) which is situated in Netherland. It was started firstly as a hobby project because he was looking for an interesting project to keep him occupied during Christmas.

The programming language which Python is said to have succeeded is ABC Programming Language, which had the interfacing with the Amoeba Operating System and had the feature of exception handling. He had already helped to create ABC earlier in his career and he had seen some issues with ABC but liked most of the features. After that what he did as really very clever. He had taken the syntax of ABC, and some of its good features. It came with a lot of complaints too, so he fixed those issues completely and had created a good scripting language which had removed all the flaws.

The inspiration for the name came from BBC’s TV Show – ‘Monty Python’s Flying Circus’, as he was a big fan of the TV show and also he wanted a short, unique and slightly mysterious name for his invention and hence he named it Python! He was the “Benevolent dictator for life” (BDFL) until he stepped down from the position as the leader on 12th July 2018. For quite some time he used to work for Google, but currently, he is working at Dropbox.  
The language was finally released in 1991.

When it was released, it used a lot fewer codes to express the concepts, when we compare it with Java, C++ & C. Its design philosophy was quite good too. Its main objective is to provide code readability and advanced developer productivity. When it was released it had more than enough capability to provide classes with inheritance, several core data types exception handling and functions.

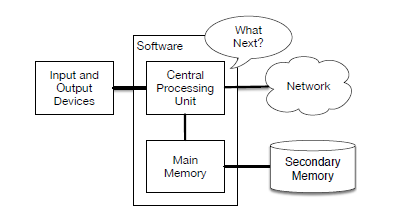
**What is python programing..?**

**Python** is an [interpreted](https://en.wikipedia.org/wiki/Interpreted_language), [high-level](https://en.wikipedia.org/wiki/High-level_programming_language), [general-purpose](https://en.wikipedia.org/wiki/General-purpose_programming_language) [programming language](https://en.wikipedia.org/wiki/Programming_language). Created by [Guido van Rossum](https://en.wikipedia.org/wiki/Guido_van_Rossum) and first released in 1991, Python's design philosophy emphasizes [code readability](https://en.wikipedia.org/wiki/Code_readability) with its notable use of [significant whitespace](https://en.wikipedia.org/wiki/Off-side_rule). Its [language constructs](https://en.wikipedia.org/wiki/Language_construct) and [object-oriented](https://en.wikipedia.org/wiki/Object-oriented_programming) approach aim to help programmers write clear, logical code for small and large-scale projects.

Python is [dynamically typed](https://en.wikipedia.org/wiki/Dynamic_programming_language) and [garbage-collected](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)). It supports multiple [programming paradigms](https://en.wikipedia.org/wiki/Programming_paradigms), including [structured](https://en.wikipedia.org/wiki/Structured_programming) (particularly, [procedural](https://en.wikipedia.org/wiki/Procedural_programming)), object-oriented, and [functional programming](https://en.wikipedia.org/wiki/Functional_programming). Python is often described as a "batteries included" language due to its comprehensive [standard library](https://en.wikipedia.org/wiki/Standard_library).

**Computer Hardware architecture**

Before we start learning the language we speak to give instructions to computer to develop software, we need to learn a small amount about how computers are built. If you were to take apart your computer or cell phone and look deep inside, you would find the following parts:-



**The high-level definitions of these parts are as follows:**

• **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).

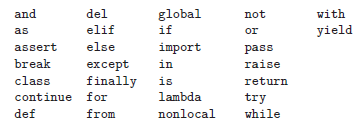
• **The Input and Output Devices**are simply our screen, keyboard, mouse, microphone, Speaker, touchpad, etc. They are all of the ways we interact with the computer.

• 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*. While most of the detail of how these components work is best left to computer builders, it helps to have some terminology so we can talk about these different parts as we write our programs.

**WORDS AND SENTENCES**

Unlike human languages, the Python vocabulary is actually pretty small. We call this “vocabulary” the “reserved words”. These are words that have very special meaning to Python. When Python sees these words in a Python program, they have one and only one meaning to Python. Later as you write programs you will make up your own words that have meaning to you called *variables*. You will have great latitude in choosing your names for your variables, but you cannot use any of Python’s reserved words as a name for a variable. When we train a dog, we use special words like “sit”, “stay”, and “fetch”. When you talk to a dog and don’t use any of the reserved words, they just look at you with a quizzical look on their face until you say a reserved word. For example, if you say, “I wish more people would walk to improve their overall health”, what most dogs likely hear is, “blah blah blah *walk* blah blah blah blah.” That is because “walk” is a reserved word in dog language. Many might suggest that the language between humans and cats has no reserved words.

The reserved words in the language where humans talk to Python include the following



That is it, and unlike a dog, Python is already completely trained. When you say “try”, Python will try every time you say it without fail. We will learn these reserved words and how they are used in good time, but for now we will focus on the Python equivalent of “speak” (in human-to-dog language).

The nice thing about telling Python to speak is that we can even tell it what to say by giving it a message in quotes: print ('Hello world!') And we have even written our first syntactically correct Python sentence. Our sentence starts with the function *print* followed by a string of text of our choosing Enclosed in single quotes. The strings in the print statements are enclosed in quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

**Conversing with Python**

Now that we have a word and a simple sentence that we know in Python, we need to know how to start a conversation with Python to test our new language skills. Before you can converse with Python, you must first install the Python software on your computer and learn how to start Python on your computer. That is too much detail for this chapter so I suggest that you consult www.py4e.com where I have detailed instructions and screencasts of setting up and starting Python on Macintosh and Windows systems. At some point, you will be in a terminal or command window and you will type *python* and the Python interpreter will start

Executing in interactive mode and appear somewhat as follows:-

Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:54:25)

[MSC v.1900 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" **for** more

information.

>>>

The >>> prompt is the Python interpreter’s way of asking you, “What do you want me to do next?” Python is ready to have a conversation with you. All you have to know is how to speak the Python language. Let’s say for example that you did not know even the simplest Python language words or sentences. You might want to use the standard line that astronauts use when they land on a faraway planet and try to speak with the inhabitants of the planet:

>>> I come in peace, please take me to your leader

File "<stdin>", line 1

I come in peace, please take me to your leader

^

SyntaxError: invalid syntax

>>>

>>> print('Hello world!')

Hello world!

This is looking much better, so you try to communicate some more:

>>> print('You must be the legendary god that comes from the sky')

You must be the legendary god that comes from the sky

>>> print('We have been waiting for you for a long time')

We have been waiting **for** you **for** a long time

>>> print('Our legend says you will be very tasty with mustard')

Our legend says you will be very tasty **with** mustard

>>> print 'We will have a feast tonight unless you say

File "<stdin>", line 1

print 'We will have a feast tonight unless you say

^

SyntaxError: Missing parentheses in call to 'print'

>>>

The conversation was going so well for a while and then you made the tiniest mistake using the Python language and Python brought the spears back out. At this point, you should also realize that while Python is amazingly complex and powerful and very picky about the syntax you use to communicate with it, Python is *not* intelligent. You are really just having a conversation with yourself, but using proper syntax.

**Interpreter and compiler Terminology:-**

Python is a *high-level* language intended to be relatively straightforward for humans to read and write and for computers to read and process. Other high-level languages include 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. The CPU understands a language we call *machine language*. Machine language is very simple and frankly very tiresome to write because it is represented all in zeroes and ones:

001010001110100100101010000001111

11100110000011101010010101101101

...

Machine language seems quite simple on the surface, given that there are only zeroes and ones, but its syntax is even more complex and far more intricate than Python. So very few programmers ever write machine language. Instead we build various translators to allow programmers to write in high-level languages like Python or JavaScript and these translators convert the programs to machine language for actual execution by the CPU.

These programming language translators fall into two general categories: (1) interpreters and (2) compilers.

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. Some of the lines of Python tell Python that you want it to remember some value for later. We need to pick a name for that value to be remembered and we can use that symbolic name to retrieve the value later. We use the term *variable* to refer to the labels we use to refer to this stored data.

>>> x = 6

>>> print(x)

6

>>> y = x \* 7

>>> print(y)

42

>>>

In this example, we ask Python to remember the value six and use the label *x* so we can retrieve the value later. We verify that Python has actually remembered the value using *print*. Then we ask Python to retrieve *x* and multiply it by seven and put the newly computed value in *y*. Then we ask Python to print out the value currently in *e*ven though we are typing these commands into Python one line at a time, Python is treating them as an ordered sequence of statements with later statements able to retrieve data created in earlier statements. We are writing our first simple paragraph with four sentences in a logical and meaningful order. It is the nature of an *interpreter* to be able to have an interactive conversation as shown above.

A compilerneeds 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. If you have a Windows system, often these executable machine language programs have a suffix of “.exe” or “.dll” which stand for “executable” and “dynamic link library” respectively. In Linux and Macintosh, there is no suffix that uniquely marks a file as executable.

**Writing a program**

Typing commands into the Python interpreter is a great way to experiment with Python’s features, but it is not recommended for solving more complex problems. When we want to write a program, we use a text editor to write the Python instructions into a file, which is called a *script*. By convention, Python scripts have names that end with .py. To execute the script, you have to tell the Python interpreter the name of the file.

In a command window, you would type python hello.py as follows:

$ **cat** hello.py

**print**('Hello world!')

$ **python** hello.py

**Hello** world!

The “$” is the operating system prompt, and the “cat hello.py” is showing us that

the file “hello.py” has a one-line Python program to print a string. We call the Python interpreter and tell it to read its source code from the file “hello.py” instead of prompting us for lines of Python code interactively.You will notice that there was no need to have *quit()* at the end of the Python

Program in the file. When Python is reading your source code from a file, it knows to stop when it reaches the end of the file.

**What is a program?**

The definition of a *program* at its most basic is a sequence of Python statements that have been crafted to do something. Even our simple *hello.py* script is a program. It is a one-line program and is not particularly useful, but in the strictest definition, it is a Python program. It might be easiest to understand what a program is by thinking about a problem.

It might be easiest to understand what a program is by thinking about a problem that a program might be built to solve, and then looking at a program that would solve that problem.

**The building blocks of programs**

* Objects - everything in Python is an object - that includes integers, floats, strings etc.
* All objects have a reference
* Python variables are Names associated with references
* Python has some fundamental types - boolean, int, float, string, list, dict(ionary), set, tuple
* Python is dynamically and strongly typed; Names don’t have a type, but objects have a type.
* Names have scope, objects don’t. Objects only ‘disappear’ if nothing references them. Objects are kept if references still exist to them, regardless of which scope the reference exists in.
* list, dict, set & tuple (containers) do not need to be homogeneous; i.e. they don’t have to record the same type all the way through
* Containers hold references to objects - not the objects themselves
* Containers can hold other containers.
* Blocks within Python code is identified by indentation
* Python has a few control structures : for, if, while
* Python has functions : **def** <name>(<arguments>)
* Python has classes with inheritance and multiple inheritance: **class** <name>( <subclasses>)
* Python has modules (source files which can be imported which then extends the functionality available to a given program), and packages (groups of modules). Modules can define variables, functions and classes in any combination.

**WHAT COULD POSSIBLY GO WRONG?**

>>> I hate you Python!

File "<stdin>", line 1

I hate you Python!

^

SyntaxError: invalid syntax

>>> **if** you come out of there, I would teach you a lesson

File "<stdin>", line 1

**if** you come out of there, I would teach you a lesson

^

SyntaxError: invalid syntax

>>>

There is little to be gained by arguing with Python. It is just a tool. It has no emotions and it is happy and ready to serve you whenever you need it. Its error messages sound harsh, but they are just Python’s call for help. It has looked at what you typed, and it simply cannot understand what you have entered. Python is much more like a dog, loving you unconditionally, having a few key words that it understands, looking you with a sweet look on its face (>>>), and waiting for you to say something it understands.

When Python says “SyntaxError: invalid syntax”, it is simply wagging its tail and saying, “You seemed to say something but I just don’t understand what you meant, but please keep talking to me (>>>).”

As your programs become increasingly sophisticated, you will encounter three general type of errors.

**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. Python does its best to point right at the line and character where it noticed it was confused.

**Logic errors: -** 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. A good example of a logic error might be, “take a drink from your water bottle, put it in your backpack, walk to the library, and then put the top back on the bottle.”

**Semantic errors: -** 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.

**Assignment Questions:-**

1) What is python language?

2) What are the key features of python?

3) How is python an interpreted language?

4) How is memory managed in python?

5) What is python path?

6) What are the generators in python?

7) Does python have oops concepts?

8) Is python case sensitive?

9) How long can an identifier be in Python?

10) With Python, how do you find out which directory you are currently in?

**Unit 1**

**Chapter 2**

**Variables, expressions, and statements**

**Syllabus:**

**Variables, expressions, and statements:** Values and types ,Variables, Variable names and keywords , Statements , Operators and operands , Expressions ,Order of operations , Modulus operator , String operations , Asking the user for input , Comments , Choosing mnemonic variable names

**Values and types**

A *value* is one of the basic things a program works with, like a letter or a number. The values we have seen so far are 1, 2, and “Hello, World!” These values belong to different *types*: 2 is an integer, and “Hello, World!” is a *string*, so called because it contains a “string” of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks. The print statement also works for integers. We use the python command to start the interpreter.python

>>> print(4)

4

If you are not sure what type a value has, the interpreter can tell you.

>>> type('Hello, World!')

<**class** 'str'>

>>> type(17)

<**class** 'int'>

Not surprisingly, strings belong to the type str and integers belong to the type int. Less obviously, numbers with a decimal point belong to a type called float, because these numbers are represented in a format called *floating point*.

>>> type(3.2)

<**class** 'float'>

What about values like “17” and “3.2”? They look like numbers, but they are in

quotation marks like strings.

>>> type('17')

<**class** 'str'>

>>> type('3.2')

<**class** 'str'>

They’re strings.When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000. This is not a legal integer in Python, but it is legal:

>>> print(1,000,000)

1 0 0

Well, that’s not what we expected at all! Python interprets 1,000,000 as a commaseparated sequence of integers, which it prints with spaces between. This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn’t do the “right” thing.

**2.2 Variables**

One of the most powerful features of a programming language is the ability to manipulate *variables*. A variable is a name that refers to a value. An *assignment statement* creates new variables and gives them values:

>>> message = 'And now for something completely different'

>>> n = 17

>>> pi = 3.1415926535897931

This example makes three assignments. The first assigns a string to a new variable named message; the second assigns the integer 17 to n; the third assigns the (approximate) value of *\_* to pi.To display the value of a variable, you can use a print statement:

>>> print(n)

17

>>> print(pi)

3.141592653589793

The type of a variable is the type of the value it refers to.

>>> type(message)

<**class** 'str'>

>>> type(n)

<**class** 'int'>

>>> type(pi)

<**class** 'float'>

**2.3 Variable names and keywords**

Programmers generally choose names for their variables that are meaningful and document what the variable is used for. Variable names can be arbitrarily long. They can contain both letters and numbers, but they cannot start with a number. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter (you’ll see why later). The underscore character ( \_ ) can appear in a name. It is often used in names with multiple words, such as my\_name or airspeed\_of\_unladen\_swallow. Variable names can start with an underscore character, but we generally avoid doing this unless we are writing library code for others to use. If you give a variable an illegal name, you get a syntax error:

>>> 76trombones = 'big parade'

SyntaxError: invalid syntax

>>> more@ = 1000000

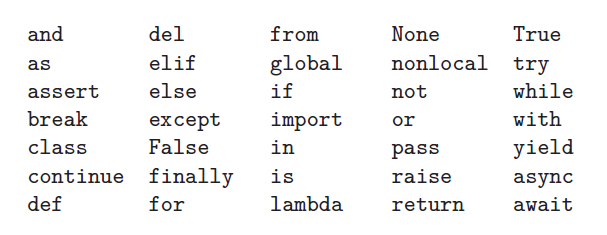
SyntaxError: invalid syntax

>>> **class** = 'Advanced Theoretical Zymurgy'

SyntaxError: invalid syntax

76trombones is illegal because it begins with a number. more@ is illegal because it contains an illegal character, @. But what’s wrong with class? It turns out that class is one of Python’s *keywords*. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Python reserves 35 keywords:



You might want to keep this list handy. If the interpreter complains about one of your variable names and you don’t know why, see if it is on this list.

**2.4 Statements**

A *statement* is a unit of code that the Python interpreter can execute. We have seen two kinds of statements: print being an expression statement and assignment. When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one. A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute. For example, the script

print(1)

x = 2

print(x)

produces the output

1

2

The assignment statement produces no output.

**2.5 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)

There has been a change in the division operator between Python 2.x and Python 3.x. In Python 3.x, the result of this division is a floating point result:

>>> minute = 59

>>> minute/60

0.9833333333333333

The division operator in Python 2.0 would divide two integers and truncate the result to an integer:

>>> minute = 59

>>> minute/60

0

To obtain the same answer in Python 3.0 use floored ( // integer) division.

>>> minute = 59

>>> minute//60

0

In Python 3.0 integer division functions much more as you would expect if you entered the expression on a calculator.

**2.6 Expressions**

An *expression* is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable x has been assigned a value):

17

x

x + 17

If you type an expression in interactive mode, the interpreter *evaluates* it and displays the result:

>>> 1 + 1

2

But in a script, an expression all by itself doesn’t do anything! This is a common source of confusion for beginners.

**Exercise 1: Type the following statements in the Python interpreter to**

**see what they do:**

5

x = 5

x + 1

**2.7 Order of operations**

When more than one operator appears in an expression, the order of evaluation depends on the *rules of precedence*. For mathematical operators, Python follows mathematical convention. The acronym *PEMDAS* is a useful way to remember the rules:

• *P*arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 \* (3-1) is 4, and (1+1)\*\*(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute \* 100) / 60, even if it doesn’t change the result.

• *E*xponentiation has the next highest precedence, so 2\*\*1+1 is 3, not 4, and 3\*1\*\*3 is 3, not 27.

• *M*ultiplication and *D*ivision have the same precedence, which is higher than *A*ddition and *S*ubtraction, which also have the same precedence. So 2\*3-1 is 5, not 4, and 6+4/2 is 8, not 5.

• Operators with the same precedence are evaluated from left to right. So the expression 5-3-1 is 1, not 3, because the 5-3 happens first and then 1 is subtracted from 2.

When in doubt, always put parentheses in your expressions to make sure the computations are performed in the order you intend.

**2.8 Modulus operator**

The *modulus operator* works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators:

>>> quotient = 7 // 3

>>> print(quotient)

2

>>> remainder = 7 % 3

>>> print(remainder)

1

So 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if x % y is zero, then x is divisible by y.

You can also extract the right-most digit or digits from a number. For example, x % 10 yields the right-most digit of x (in base 10). Similarly, x % 100 yields the last two digits.

**2.9 String operations**

The + operator works with strings, but it is not addition in the mathematical sense. Instead it performs *concatenation*, which means joining the strings by linking them end to end. For example:

>>> first = 10

>>> second = 15

>>> print(first+second)

25

>>> first = '100'

>>> second = '150'

>>> print(first + second)

100150

The \* operator also works with strings by multiplying the content of a string by an integer. For example:

>>> first = 'Test '

>>> second = 3

>>> print(first \* second)

Test Test Test

**2.10 Asking the user for input**

Sometimes we would like to take the value for a variable from the user via their keyboard. Python provides a built-in function called input that gets input from the keyboard1. 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.

>>> inp = input()

Some silly stuff

>>> print(inp)

Some silly stuff

Before getting input from the user, it is a good idea to print a prompt telling the user what to input. You can pass a string to input to be displayed to the user before pausing for input:

>>> name = input('What is your name?\n')

What is your name?

Chuck

>>> print(name)

Chuck

The sequence \n at the end of the prompt represents a *newline*, which is a special character that causes a line break. That’s why the user’s input appears below the prompt.

If you expect the user to type an integer, you can try to convert the return value to int using the int() function:

>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'

>>> speed = input(prompt)

What...is the airspeed velocity of an unladen swallow?

17

>>> int(speed)

17

>>> int(speed) + 5

22

But if the user types something other than a string of digits, you get an error:

>>> speed = input(prompt)

What...is the airspeed velocity of an unladen swallow?

What do you mean, an African or a European swallow?

>>> int(speed)

ValueError: invalid literal **for** int() **with** base 10:

We will see how to handle this kind of error later.

**2.11 Comments**

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why. For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called *comments*, and in Python they start with the # symbol:

*# compute the percentage of the hour that has elapsed*

percentage = (minute \* 100) / 60

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

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

Everything from the # to the end of the line is ignored; it has no effect on the program. Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out *what* the code does; it is much more useful to explain *why*. This comment is redundant with the code and useless:

v = 5 *# assign 5 to v*

This comment contains useful information that is not in the code:

v = 5 *# velocity in meters/second.*

Good variable names can reduce the need for comments, but long names can make

complex expressions hard to read, so there is a trade-off.

**2.12 Choosing mnemonic variable names**

As long as you follow the simple rules of variable naming, and avoid reserved words, you have a lot of choice when you name your variables. In the beginning, this choice can be confusing both when you read a program and when you write your own programs. For example, the following three programs are identical in terms of what they accomplish, but very different when you read them and try to understand them.

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)

The Python interpreter sees all three of these programs as *exactly the same* but humans see and understand these programs quite differently. Humans will most quickly understand the *intent* of the second program because the programmer has chosen variable names that reflect their intent regarding what data will be stored in each variable. We call these wisely chosen variable names “mnemonic variable names”. The word *mnemonic* means “memory aid”. We choose mnemonic variable names to help us remember why we created the variable in the first place. While this all sounds great, and it is a very good idea to use mnemonic variable names, mnemonic variable names can get in the way of a beginning programmer’s ability to parse and understand code. This is because beginning programmers have not yet memorized the reserved words (there are only 33 of them) and sometimes variables with names that are too descriptive start to look like part of the language and not just well-chosen variable names. Take a quick look at the following Python sample code which loops through some data. We will cover loops soon, but for now try to just puzzle through what this means:

**for** word in words:

print(word)

What is happening here? Which of the tokens (for, word, in, etc.) are reserved words and which are just variable names? Does Python understand at a fundamental level the notion of words? Beginning programmers have trouble separating what parts of the code *must* be the same as this example and what parts of the code are simply choices made by the programmer.

The following code is equivalent to the above code:

**for** slice in pizza:

print(slice)

It is easier for the beginning programmer to look at this code and know which parts are reserved words defined by Python and which parts are simply variable names chosen by the programmer. It is pretty clear that Python has no fundamental understanding of pizza and slices and the fact that a pizza consists of a set of one or more slices. But if our program is truly about reading data and looking for words in the data, pizza and slice are very un-mnemonic variable names. Choosing them as variable names distracts from the meaning of the program. After a pretty short period of time, you will know the most common reserved words and you will start to see the reserved words jumping out at you: The parts of the code that are defined by Python (for, in, print, and :) are in bold and the programmer-chosen variables (word and words) are not in bold. Many text editors are aware of Python syntax and will color reserved words differently to give you clues to keep your variables and reserved words separate. After a while you will begin to read Python and quickly determine what is a variable and what is a reserved word.

**2.13 Debugging**

At this point, the syntax error you are most likely to make is an illegal variable name, like class and yield, which are keywords, or odd~job and US$, which contain illegal characters.

If you put a space in a variable name, Python thinks it is two operands without an operator:

>>> bad name = 5

SyntaxError: invalid syntax

>>> month = 09

File "<stdin>", line 1

month = 09

^

SyntaxError: invalid token

For syntax errors, the error messages don’t help much. The most common messages are SyntaxError: invalid syntax and SyntaxError: invalid token, neither of which is very informative. The runtime error you are most likely to make is a “use before def;” that is, trying to use a variable before you have assigned a value. This can happen if you spell a variable name wrong:

>>> principal = 327.68

>>> interest = principle \* rate

NameError: name 'principle' is not defined Variables names are case sensitive, so LaTeX is not the same as latex. At this point, the most likely cause of a semantic error is the order of operations.

For example, to evaluate 1*/*2*\_*, you might be tempted to write

>>> 1.0 / 2.0 \* pi

But the division happens first, so you would get *\_/*2, which is not the same thing! There is no way for Python to know what you meant to write, so in this case you don’t get an error message; you just get the wrong answer.

Assignment Questions:-

1.Write a python program to find the factorial of a number?

2. Write a Python program to add key to a dictionary.

3. Write a python program to find the intersection of elements from two list.

4. Write a Python program to sum all the items in a list.

5.  Write a Python program to get the smallest number from a list.

6.  Write a Python program to get a single string from two given strings, separated by a space and swap the first two characters of each string.

7. Write a Python program to test whether an input is an integer.

8. Write a Python program to check a list is empty or not.

9. Write a Python program to find the list of words that are longer than n from a given list of words

10. Write a Python program to remove duplicates from a list.

.