Let us talk about these literal types one by one.

Integer Literals

Integer literals are whole numbers without any fractional part. The method of writing integer constants has been specified in the following rule :

An integer constant must have at least one digit and must not contain any decimal point. It may contain either (+) or (–) sign. A number with no sign is assumed to be positive. Commas cannot appear in an integer constant.

NOTE

Many programming languages such as C, C++, and even Python 2.x too have two types for integers : **int** (for small integers) and **long** (for big integers). But in **Python 3.x**, there is only one integer type <class 'int'> that works like long integers and can support all small and big integers.

Python allows *three* types of integer literals :

(*i*) ***Decimal Integer Literals.*** An integer literal consisting of a sequence of digits is taken to be decimal integer literal unless it begins with 0 (digit zero).

For instance, 1234, 41, +97, –17 are decimal integer literals.

(*ii*) ***Octal Integer Literals.*** A sequence of digits starting with 0o (digit zero followed by letter o) is taken to be an octal integer.

For instance, decimal integer 8 will be written as 0o10 as octal integer. ($8_{10} = 10_8$) and decimal integer 12 will be written as 0o14 as octal integer ($12_{10} = 14_8$).

An octal value can contain only digits 0-7 ; 8 and 9 are invalid digits in an octal number i.e., 0o28, 0o19, 0o987 etc., are examples of invalid octal numbers as they contain digits 8 and 9 in them.

(*iii*) ***Hexadecimal Integer Literals.*** A sequence of digits preceded by ***0x*** or ***0X*** is taken to be an hexadecimal integer.

For instance, decimal 12 will be written as 0XC as hexadecimal integer.

Thus, number 12 will be written either as 12 (as decimal), 0o14 (as octal) and 0XC (as hexadecimal).

A hexadecimal value can contain digits 0-9 and letters A-F only *i.e., 0XBK9, 0xPQR, 0x19AZ* etc., are examples of invalid hexadecimal numbers as they contain invalid letters, *i.e.,* letters other than A-F.

Floating Point Literals

Floating literals are also called real literals. Real literals are numbers having fractional parts. These may be written in one of the *two* forms called ***Fractional Form*** or the **Exponent Form.**

1. ***Fractional form.*** A real literal in Fractional Form consists of signed or unsigned digits including a decimal point between digits.

The rule for writing a real literal in fractional form is :

A real constant in fractional form must have at least one digit with the decimal point, either before or after. It may also have either + or – sign preceding it. A real constant with no sign is assumed to be positive.

The following are valid real literals in fractional form :

2.0, 17.5, – 13.0, – 0.00625,
.3 (*will represent* 0.3), 7. (*will represent* 7.0)

The following are invalid real literals :

7	(*No decimal point*)
+17 / 2	(*/-illegal symbol*)
17,250.26.2	(*Two decimal points*)
17,250.262	(*comma not allowed*)

2. ***Exponent form.*** A real literal in *Exponent form* consists of *two* parts : ***mantissa*** and ***exponent***. For instance, 5.8 can be written as $\mathbf{0.58 \times 10^1 = 0.58\ E01}$, where *mantissa* part is **0.58** (the part appearing before E) and *exponent* part is **1** (the part appearing after E). E01 represents 10^1. The rule for writing a real literal in exponent form is :

A real constant in exponent form has two parts : a ***mantissa*** *and an* ***exponent.*** *The mantissa must be either an integer or a proper real constant. The mantissa is followed by a letter E or e and the exponent. The exponent must be an integer.*

The following are the valid real literals in exponent form :

152E05, 1.52E07, 0.152E08, 152.0E08, 152E+8, 1520E04, –0.172E–3, 172.E3, .25E–4, 3.E3 (equivalent to 3.0E3)

To see Exponent Form in scan

(Even if there is no preceding or following digit of a decimal point, Python 3.x will consider it right)

The following are invalid real literals in exponent form :

1.7E	(*No digit specified for exponent*)
0.17E2.3	(*Exponent cannot have fractional part*)
17,225E02	(*No comma allowed*)

[Do read following discussion after it.]

Numeric values with commas are not considered **int** or **float** value, rather Python treats them as a tuple. A tuples is a special type in Python that stores a *sequence of values*. (You will learn about tuples in coming chapters – for now just understand a tuple as *a sequence of values only*.)

The last invalid example value given above (17,225e02) asks for a special mention here.

Any numeric value with a comma in its mantissa will not be considered a legal *floating point number*, **BUT if you assign this value, Python won't give you an error**. The reason being is that Python will not consider that as a *floating point value* rather a **tuple**. Carefully have a look at the figure that illustrates it.

```
>>> a = 1,234
>>> b = 17,225E02
>>> type(a)
tuple
>>> type(b)
tuple
>>> a
(1, 234)
>>> b
(17,22500.0)
```

Python gives no error when you assign a numeric value with comma in it.

Python will not consider the numeric values with commas in them as numbers (int or float) BUT as a tuple – a sequence of values

We are not talking about Complex numbers here. These would be discussed later when the need arises.

6.3.3C Boolean Literals

A Boolean literal in Python is used to represent one of the two Boolean values *i.e.*, **True** (Boolean true) or **False** (Boolean false). A Boolean literal can either have value as *True* or as *False*.

NOTE

True and **False** are the only two Boolean literal values in Python. **None** literal represents absence of a value.

6.3.3D Special Literal None

Python has one special literal, which is **None**. The **None** literal is used to indicate absence of value. It is also used to indicate the end of lists in Python.

The **None** value in Python means *"There is no useful information"* or *"There's nothing here."* Python doesn't display anything when asked to display the value of a variable containing value as **None**. Printing with print statement, on the other hand, shows that the variable contains **None** (*see* figure below).

```
>>> Value1 = 10
>>> Value2 = None
>>> Value1
10
>>> Value2
>>> print(Value2)
None
```

Displaying a variable containing **None** does not show anything. However, with ***print( )***, it shows the value contained as **None**.

Python supports literal collections also such as tuples and lists etc. But covering these here would make the discussion too complex for the beginning. So, we'll take them at a later time.

NOTE

Boolean literals **True**, **False** and special literal None are some built-in constants/literals of Python.

Check Point 6.2

1. What are literals ? How many types of literals are available in Python ?
2. How many types of integer literals are allowed in Python ? How are they written ?
3. Why are characters \, ', " and tab typed using escape sequences ?
4. Which escape sequences represent the newline character and backspace character ? An escape sequence represents how many characters ?
5. What are string-literals in Python ? How many ways, can you create String literals in Python ? Are there any differences in them ?
6. What is meant by a floating-point literal in Python ? How many ways can a floating literal be represented into ?
7. Write the following real constants into exponent form :
   23.197, 7.214, 0.00005, 0.319
8. Write the following real constants into fractional form :
   0.13E04, 0.417E-04, 0.4E-5,
   .12E02, 12.E02
9. What are the two Boolean literals in Python ?
10. Name some built-in literals of Python.
11. Out of the following literals, determine their type whether decimal / octal / hexadecimal integer literal or a floating point literal in fractional or exponent form or string literal or other ?
    123, 0o124, 0xABC, 'abc', "ABC",
    12.34, 0.3E-01, '''ftghjkjl''',
    None, True, False
12. What kind of program elements are the following ?
    'a', 4.38925, "a", "main" ?
13. What will **var1** and **var2** store with statements : **var1** = 2,121E2 and **var2** = 0.2,121E2 ? What are the types of values stored in **var1** and **var2** ?

Pip BASICS ABOUT TOKENS Progress In Python 6.1

Start a Python IDE of your choice and do as directed.

⋮

>>>❖<<<

6.3.4 Operators

Operators are tokens that trigger some computation when applied to variables and other objects in an expression. Variables and objects to which the computation is applied, are called **operands**. So, an operator requires some *operands* to work upon.

The following list gives a brief description of the operators and their functions / operators, in details, will be covered in next chapter – *Data Handling*.

OPERATORS

Operators are tokens that trigger some computation / action when applied to variables and other objects in an expression.

Unary Operators

Unary operators are those operators that require one operand to operate upon. Following are some unary operators :

+	Unary plus	–	Unary minus
~	Bitwise complement	not	logical negation

Binary Operators

Binary operators are those operators that require two operands to operate upon. Following are some binary operators :

Arithmetic operators

+	Addition	–	Subtraction
*	Multiplication	/	Division
%	Remainder/ Modulus		
**	exponent (raise to power)		
//	Floor division		

Bitwise operators

&	Bitwise AND
^	Bitwise exclusive OR (XOR)
|	Bitwise OR

Shift operators

<<	shift left	>>	shift right

Identity operators

is	is the identity same ?
is not	is the identity not same ?

Relational operators

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Assignment operators

=	Assignment
/=	Assign quotient
+=	Assign sum
*=	Assign product
%=	Assign remainder
-=	Assign difference
**=	Assign Exponent
//=	Assign Floor division

Logical operators

and	Logical AND
or	Logical OR

Membership operators

in	whether variable in sequence
not in	whether variable not in sequence

More about these operators you will learn in the due course. Giving descriptions and examples is not feasible and possible right here at the moment.

6.3.5 Punctuators

Punctuators are symbols that are used in programming languages to organize sentence structures, and indicate the rhythm and emphasis of expressions, statements, and program structure.

Most common punctuators of Python programming language are :

```
' " # \ ( ) [ ] { } @ , : . ` =
```

The usage of these punctuators will be discussed when the need arises along with normal topic discussions.

PUNCTUATORS

Punctuators are symbols that are used in programming languages to organize programming-sentence structures, and indicate the rhythm and emphasis of expressions, statements, and program structure.

LET US REVISE

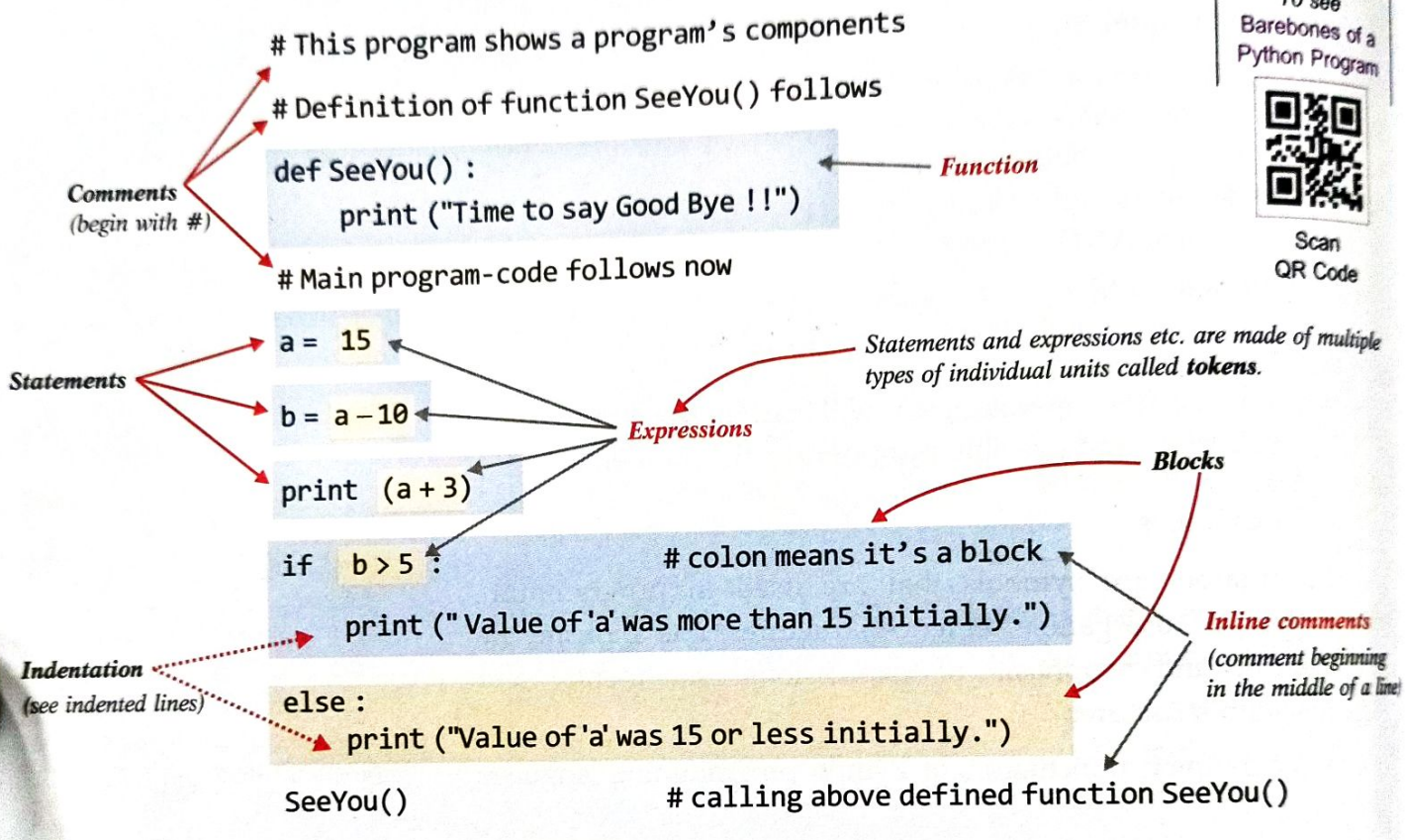
- *A token is the smallest individual unit in a program.*
- *Python provides following tokens : keywords, identifiers (names), Values (literals), punctuators, operators and comments.*
- *A keyword is a reserved word carrying special meaning and purpose.*
- *Identifiers are the user-defined names for different parts of the program.*
- *In Python, an identifier may contain **letters** (a-z, A-Z), **digits** (0-9) and a symbol underscore (_). However, an identifier must begin with a letter or underscore ; all letters/digits in an identifier are significant.*
- *Literals are the fixed values.*
- *Python allows following literals : string literal, numeric (integer, floating-point literals, Boolean literals, special literal **None** and literal collections).*
- *Operators are tokens that trigger some computation / action when applied to variables and other objects in an expression.*
- *Punctuators are symbols used to organize programing-sentence structures and indicate the rhythm and emphasis of expressions, statements and program-structure.*

6.4 Barebones of a Python Program

Let us take our discussion further. Now we are going to talk about the basic structure of a Python program – what all it can contain. Before we proceed, have a look at following sample code. Look at the code and then proceed to the discussion that follows. Don't worry if the things are not clear to you right now. They'll become clear when the discussion proceeds.

To see Barebones of a Python Program

Scan QR Code



As you can see that the above sample program contains various components like :

- expressions
- statements
- comments
- function
- blocks and indentation

Let us now discuss various components shown in above sample code.

(i) Expressions

An ***expression*** is any legal combination of symbols that ***represents a value.*** An expression represents something, which **Python evaluates** and which then produces a *value*.

Some examples of expressions are

```
15
2.9
```
} *expressions that are values only*

```
a + 5
(3 + 5) / 4
```
} *complex expressions that produce a value when evaluated.*

EXPRESSIONS

An ***expression*** is any legal combination of symbols that ***represents a value.***

Now from the above sample code, can you pick out all expressions ?

These are : 15, $a-10$, $a+3$, $b>5$

(ii) Statement

While an expression represents something, a statement is a programming instruction that does something *i.e.*, some action takes place.

Following are some examples of statements :

```
print ("Hello")   # this statement calls print function
if b > 5 :
    ⋮
```

STATEMENT

A statement is a programming instruction that does something *i.e.*, some action takes place.

While an expression is evaluated, a statement is executed *i.e.*, some action takes place. And it is **not necessary that a statement results in a value** ; it may or may not yield a value.

Some statements from the above sample code are :

```
a = 15
b = a – 10
print (a + 3)
if b < 5 :
    ⋮
```

NOTE

A statement executes and may or may not yield a value.

(iii) Comments

Comments are the additional readable information, which is read by the programmers but ignored by Python interpreter. In Python, comments begin with symbol # (Pound or hash character) and end with the end of physical line.

COMMENTS

Comments are the additional readable information to clarify the source code.

Comments in Python begin with symbol # and generally end with end of the physical line.

In the above code, you can see *four* comments :

(*i*) The physical lines beginning with # are the **full line comments**. There are *three* full line comments in the above program are :

```
# This program shows a program's components
# Definition of function SeeYou( ) follows
# Main program code follows now
```

(*ii*) The fourth comment is an ***inline comment*** as it starts in the middle of a physical line, after Python code (see below)

```
if b < 5 : # colon means it requires a block
```

NOTE

A *Physical line* is the one complete line that you see on a computer whereas a *logical line* is the one that Python sees as one full statement.

Multi-line Comments

What if you want to enter a ***multi-line comment*** or a ***block comment*** ?. You can enter a multi-line comment in Python code in *two* ways :

(*i*) Add a # symbol in the beginning of every physical line part of the multi-line comments, *e.g.*,

```
# Multi-line comments are useful for detailed additional information.
# Related to the program in question.
# It helps clarify certain important things.
```

(ii) Type comment as a triple-quoted multi-line string *e.g.*,

```
'''  Multi-line comments are useful for detailed additional
     information related to the program in question.
     It helps clarify certain important things
'''
```

This type of multi-line comment is also known as ***docstring***. You can either use triple-apostrophe (''') or triple quotes (""") to write *docstrings*. The docstrings are very useful in documentation – and you'll learn about their usage later.

NOTE

Comments enclosed in triple quotes (" " ") or triple apostrophe (' ' ') are called **docstrings**.

(iv) Functions

A function is a code that has a name and it can be reused (executed again) by specifying its name in the program, where needed.

In the above sample program, there is one function namely **SeeYou( )**. The statements indented below its **def** statement are part of the function. [All statements indented at the same level below *def SeeYou( )* are part of *SeeYou( )*.] This function is executed in main code through following statement (Refer to sample program code given above)

FUNCTIONS

A **function** is a code that has a name and it can be reused (executed again) by specifying its name in the program, where needed.

```
SeeYou()      # function-call statement
```

Calling of a function becomes a statement *e.g.*, **print** is a function but when you call **print( )** to print something, then that function call becomes a statement.

For now, only this much introduction of functions is sufficient. You will learn about functions in details in Class 12.

(v) Blocks and Indentation

Sometimes a group of statements is part of another statement or function. Such a group of one or more statements is called **block** or **code-block** or **suite**. *For example,*

Four spaces together mark the next indent-level

```
if b < 5 :
    print ("Value of 'b' is less than 5.")
    print ("Thank you.")
```

This is a block with all its statements at same indentation level.

Many languages such as C, C++, Java etc., use symbols like curly brackets to show blocks but Python does not use any symbol for it, rather it uses indentation.

Consider the following example :

```
if b < a :
    tmp = a
    a = b
    b = tmp
print ("Thank you")
```

*This is a block, part of **if** statement. Notice, all statements in same block have same indentation*

This statement is not part of if's block as it is at different indentation level.

BLOCK OR CODE-BLOCK OR SUITE

A group of statements which are part of another statement or a function are called *block* or *code-block* or *suite* in Python.

A group of individual statements which make a single *code-block* is also called a **suite** in Python. Consider some more examples showing indentation to create blocks :

```
def check() :
    c = a + b
    if c < 50 :
        print ('Less than 50')
        b = b * 2
        a = a + 10
    else :
        print ('>= 50')
        a = a * 2
        b = b + 10
```

Two different indentation-levels inside this code.

Block inside function check( )

Block / suite inside if statement

Block / suite inside else statement

NOTE

Python uses indentation to create blocks of code. Statements at same indentation level are part of same block/suite.

Statements requiring suite/ code-block have a colon (:) at their end.

You cannot unnecessarily indent a statement ; Python will raise error for that.

Python Style Rules and Conventions

While working in Python, one should keep in mind certain style rules and conventions. In the following lines, we are giving some very elementary and basic style rules :

Statement Termination Python does not use any symbol to terminate a statement. When you end a physical code-line by pressing Enter key, the statement is considered terminated by default.

Maximum Line Length Line length should be maximum 79 characters.

Lines and Indentation Blocks of code are denoted by line indentation, which is enforced through *4 spaces* (not tabs) per indentation level.

Blank Lines Use *two* blank lines between top-level definitions, *one* blank line between method/function definitions.

Functions and methods should be separated with *two* blank lines and Class definitions with *three* blank lines.

Avoid multiple statements on one line Although you can combine more than one statements in one line using symbol semicolon (;) between two statements, but it is not recommended.

Whitespace You should always have whitespace around operators and after punctuation but not with parentheses. Python considers these 6 characters as whitespace : ' ' (space), '\n' (newline), '\t' (horizontal tab), '\v' (vertical tab), '\f' (formfeed) and '\r' (carriage return)

Case Sensitive Python is case sensitive, so case of statements is very important. Be careful while typing code and identifier-names.

Docstring Convention Conventionally triple double quotes (""") are used for docstrings.

Identifier Naming You may use underscores to separate words in an identifier *e.g.*, **loan_amount** or use *CamelCase* by capitalizing first letter of the each word *e.g.*, LoanAmount or loanAmount.

Check Point 6.3

1. What is an expression in Python ?
2. What is a statement in Python ? How is a statement different from expression ?
3. What is a comment ? In how many ways can you create comments in Python ?
4. How would you create multi-line comment in Python ?
5. What is the difference between full-line comment and inline comment ?
6. What is a block or suite in Python ? How is indentation related to it ?

⋮

This is another program with different components

⋮

```
def First5Multiples( ) :
    ⋮
# main code
    ⋮
```

Fill the appropriate components of program from the above code.

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 6.2 under Chapter 6 after practically doing it on the computer.

>>>❖<<<

6.5 Variables and Assignments

A variable in Python represents named location that refers to a value and whose values can be used and processed during program run. For instance, to store name of a student and marks of a student during a program run, we require some labels to refer to these *marks* so that these can be distinguished easily. *Variables*, called as *symbolic variables*, serve the purpose. The variables are called symbolic variables because these are named labels. *For instance*, the following statement creates a variable namely ***marks*** of **Numeric** type :

```
marks = 70
```

VARIABLES

Named labels, whose values can be used and processed during program run, are called *Variables*.

6.5.1 Creating a Variable

Recall the statement we used just now to create the variable **marks** :

```
marks = 70
```

As you can see, that we just assigned the value of numeric type to an identifier name and Python created the variable of the type similar to the type of value assigned. In short, after the above statement, we can say that ***marks*** is a *numeric* variable.

So, creating variables was just that simple only ? Yes, you are right. In Python, to create a variable, just assign to its name the value of appropriate type. *For example*, to create a variable namely ***student*** to hold student's name and variable ***age*** to hold student's age, you just need to write somewhat similar to what is shown below :

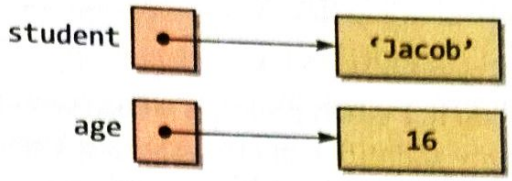
```
student = 'Jacob'
age = 16
```

NOTE

Python variables are created by assigning value of desired type to them, *e.g.*, to create a numeric variable, assign a numeric value to *variable_name* ; to create a string variable, assign a string value to *variable_name*, and so on.

You cannot unnecessarily indent a statement ; Python will raise error for that.

Python will internally create **labels referring to these values** as shown below :



Isn't it simple?? ☺ Okay, let's now create some more variables.

```
trainNo = 'T#1234'          # variable created of String type
balance = 23456.75          # variable created of Numeric(floating point) type
rollNo = 105                # variable create of Numeric(integer) type
```

Same way, you can create as many variables as you need for your program.

IMPORTANT – Variables are Not Storage Containers in Python

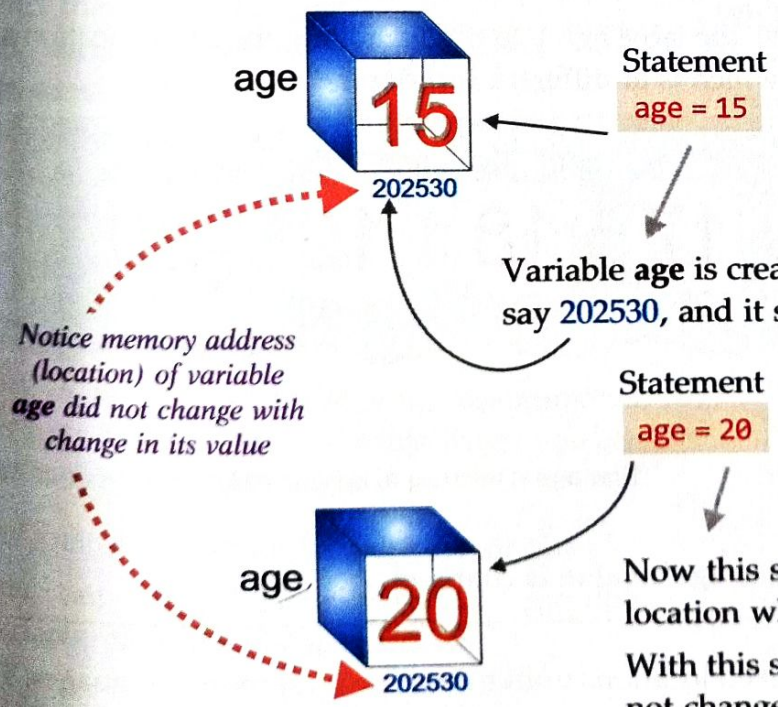
If you have an earlier exposure to programming, you must be having an idea of variables. BUT PYTHON VARIABLES ARE NOT CREATED IN THE FORM MOST OTHER PROGRAMMING LANGUAGES DO. Most programming languages create variables as storage containers *e.g.*,

Consider this :

```
age = 15
age = 20
```

Firstly value 15 is assigned to variable **age** and then value 20 is assigned to it.

Traditional Programming Languages' Variables



[Variable is a container in traditional programming languages like C, C++, *i.e.*, it is a named storage location that stores a value in it.]

Now this statement will change the contents of the location where variable **age** is created, *i.e.*,

With this statement, the location of the variable did not change, only its contents changed.

This is how traditionally variables were created in programming languages like C, C++, Java etc.

Python Variables in Memory

To see Variables in Memory in action

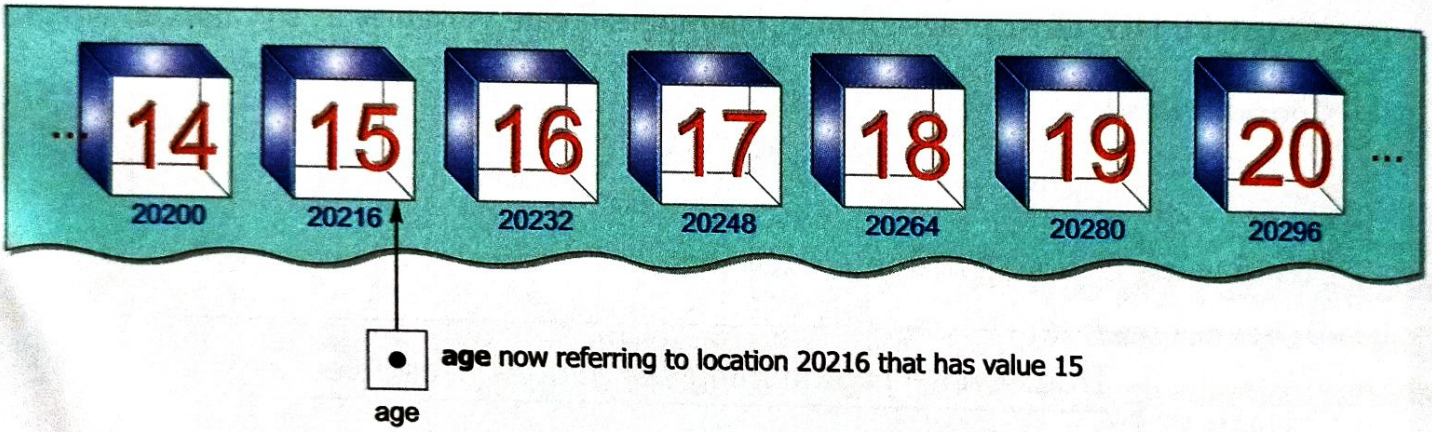
Scan QR Code

BUT PYTHON DOES THIS DIFFERENTLY

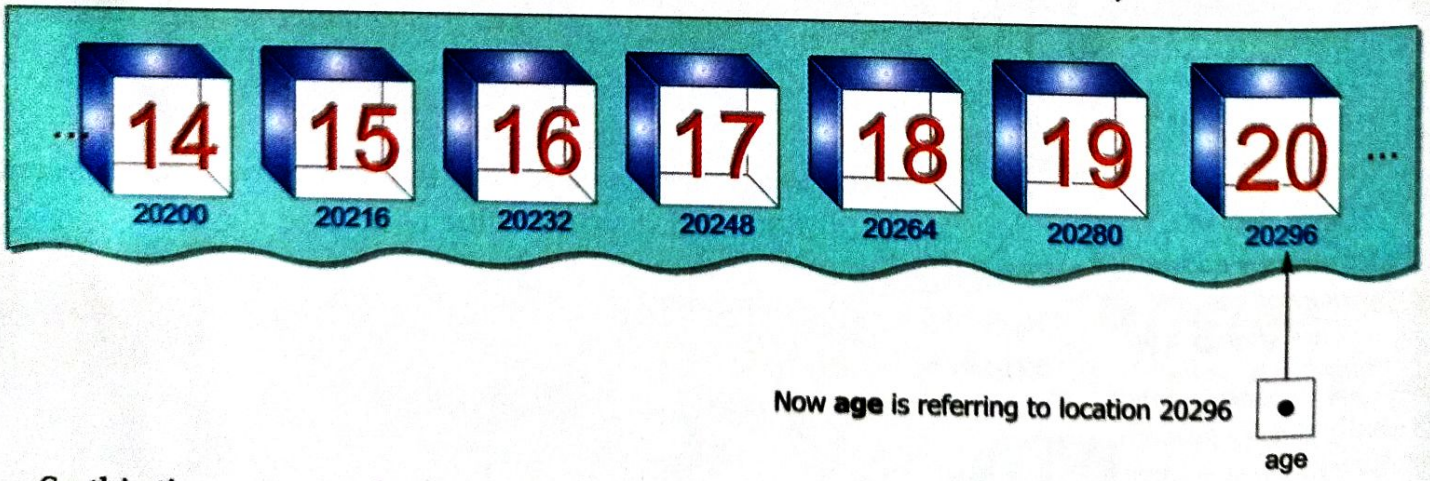
Let us see how Python will do it. Python preloads some commonly used values (even before any identifier is created) in an area of memory. We can refer to this area as *front-loaded dataspace*.

The dataspace memory has literals/values at defined memory locations, and each memory location has a memory address.

When you give statement **age = 15**, variable **age** will be created as a label pointing to memory location where value **15** is stored *i.e.*, as.



And when you give statement **age = 20**, the label **age** will not be having the same location as earlier. It will now refer to value 20, which is at different location *i.e.*,



So this time memory location of variable **age's** value is changed.

Thus variables in Python do not have fixed locations unlike other programming languages. The location they refer to changes everytime their values change (**This rule is not for all types of variables, though**). It will become clear to you in the next chapter, where we talk about *Mutable and Immutable types*.