Python basic concepts and programming

1.1 Python Keywords:

Python keywords are special reserved words that have specific meanings and purposes and can't be used for anything but those specific purposes. These keywords are always available you'll never have to import them into your code.

Python keywords are different from Python's built-in functions and types. The built-in functions and types are also always available, but they aren't as restrictive as the keywords in their usage.

As of Python 3.8, there are 35 keywords in Python. You can get a list of available keywords by using help():

```
>>> help("keywords")

Here is a list of the Python keywords.  Enter any keyword to get more help.

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

1.2 Variables, expressions and statements

1.2.1 Values and types

A **value** is one of the fundamental things — like a letter or a number — that a program manipulates. The values we have seen so far are 2 (the result when we added 1 + 1), 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.

```
>>> print 4
4
```

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

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type '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)
<type 'float'>
```

What about values like '17' and '3.2'? They look like numbers, but they are in quotation marks like strings.

```
>>> type('17')
<type 'str'>
>>> type('3.2')
<type '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 a legal expression:

```
>>> print 1,000,000
1 0 0
```

Well, that's not what we expected at all! Python interprets 1,000,000 as a comma-separated list of three integers, which it prints consecutively. 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.

1.2.2 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.

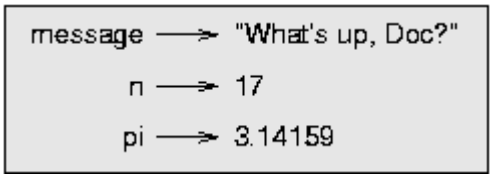
The **assignment statement** creates new variables and gives them values:

```
>>> message = "What's up, Doc?"
>>> n = 17
>>> pi = 3.14159
```

This example makes three assignments. The first assigns the string "What's up, Doc?" to a new variable named message. The second gives the integer 17 to n, and the third gives the floating-point number 3.14159 to pi.

Notice that the first statement uses double quotes to enclose the string. In general, single and double quotes do the same thing, but if the string contains a single quote (or an apostrophe, which is the same character), you have to use double quotes to enclose it.

A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a **state diagram** because it shows what state each of the variables is in (think of it as the variable's state of mind). This diagram shows the result of the assignment statements:



The print statement also works with variables.

```
>>> print message
What's up, Doc?
>>> print n
17
>>> print pi
3.14159
```

In each case the result is the value of the variable. Variables also have types; again, we can ask the interpreter what they are.

```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

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

1.2.3 Variable names and keywords

Programmers generally choose names for their variables that are meaningful — they document what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. Although it is legal to use uppercase letters, by convention we don't. If you do, remember that case matters. Bruce and bruce are different variables.

The underscore character (_) can appear in a name. It is often used in names with multiple words, such as my_name or price_of_tea_in_china.

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 = 'Computer Science 101'
SyntaxError: invalid syntax
```

76trombones is illegal because it does not begin with a letter. more$ is illegal because it contains an illegal character, the dollar sign. But what's wrong with class?

It turns out that class is one of the Python **keywords**. Keywords define the language's rules and structure, and they cannot be used as variable names.

Python has twenty-nine keywords:

```
and       def       exec      if        not       return
assert    del       finally   import    or        try
break     elif      for       in        pass      while
class     else      from      is        print     yield
continue  except    global    lambda    raise
```

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.

1.2.4 Statements

A statement is an instruction that the Python interpreter can execute. We have seen two kinds of statements: print and assignment.

When you type a statement on the command line, Python executes it and displays the result, if there is one. The result of a print statement is a value. Assignment statements don't produce a result.

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
```

Again, the assignment statement produces no output.

1.2.5 Evaluating expressions

An expression is a combination of values, variables, and operators. If you type an expression on the command line, the interpreter **evaluates** it and displays the result:

```
>>> 1 + 1
2
```

Although expressions contain values, variables, and operators, not every expression contains all of these elements. A value all by itself is considered an expression, and so is a variable.

```
>>> 17
17
>>> x
2
```

Confusingly, evaluating an expression is not quite the same thing as printing a value.

```
>>> message = 'Hello, World!'
>>> message
'Hello, World!'
>>> print message
Hello, World!
```

When the Python interpreter displays the value of an expression, it uses the same format you would use to enter a value. In the case of strings, that means that it includes the quotation marks. But if you use a print statement, Python displays the contents of the string without the quotation marks.

In a script, an expression all by itself is a legal statement, but it doesn't do anything. The script

```
17
3.2
'Hello, World!'
1 + 1
```

produces no output at all. How would you change the script to display the values of these four expressions?

1.2.6 Operators and operands

Operators are special symbols that represent computations like addition and multiplication. The values the operator uses are called **operands**.

The following are all legal Python expressions whose meaning is more or less clear:

```
20+32  hour-1  hour*60+minute  minute/60  5**2  (5+9)*(15-7)
```

The symbols +, -, and /, and the use of parenthesis for grouping, mean in Python what they mean in mathematics. The asterisk (*) is the symbol for multiplication, and ** is the symbol for exponentiation.

When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed.

Addition, subtraction, multiplication, and exponentiation all do what you expect, but you might be surprised by division. The following operation has an unexpected result:

```
>>> minute = 59
>>> minute/60
0
```

The value of minute is 59, and in conventional arithmetic 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that Python is performing **integer division**.

When both of the operands are integers, the result must also be an integer, and by convention, integer division always rounds *down*, even in cases like this where the next integer is very close.

A possible solution to this problem is to calculate a percentage rather than a fraction:

```
>>> minute*100/60
98
```

Again the result is rounded down, but at least now the answer is approximately correct. Another alternative is to use floating-point division, which we get to in Chapter 3.

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**. Python follows the same precedence rules for its mathematical operators that mathematics does. The acronym **PEMDAS** is a useful way to remember the order of operations:

- **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 though it doesn't change the result.
- **E**xponentiation has the next highest precedence, so 2**1+1 is 3 and not 4, and 3*1**3 is 3 and 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 yields 5 rather than 4, and 2/3-1 is -1, not 1 (remember that in integer division, 2/3=0).
- Operators with the same precedence are evaluated from left to right. So in the expression minute*100/60, the multiplication happens first, yielding 5900/60, which in turn yields 98. If the operations had been evaluated from right to left, the result would have been 59*1, which is 59, which is wrong.

1.2.8 Operations on strings

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The following are illegal (assuming that message has type string):

```
message-1   'Hello'/123   message*'Hello'   '15'+2
```

Interestingly, the + operator does work with strings, although it does not do exactly what you might expect. For strings, the + operator represents **concatenation**, which means joining the two operands by linking them end-to-end. For example:

```
fruit = 'banana'
bakedGood = ' nut bread'
print fruit + bakedGood
```

The output of this program is banana nut bread. The space before the word nut is part of the string, and is necessary to produce the space between the concatenated strings.

The * operator also works on strings; it performs repetition. For example, 'Fun'*3 is 'FunFunFun'. One of the operands has to be a string; the other has to be an integer.

On one hand, this interpretation of + and * makes sense by analogy with addition and multiplication. Just as 4*3 is equivalent to 4+4+4, we expect 'Fun'*3 to be the same as 'Fun'+'Fun'+'Fun', and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition and multiplication have that string concatenation and repetition do not?

1.2.9 Composition

So far, we have looked at the elements of a program — variables, expressions, and statements — in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, we know how to add numbers and we know how to print; it turns out we can do both at the same time:

```
>>>  print 17 + 3
20
```

In reality, the addition has to happen before the printing, so the actions aren't actually happening at the same time. The point is that any expression involving numbers, strings, and variables can be used inside a print statement. You've already seen an example of this:

```
print 'Number of minutes since midnight: ', hour*60+minute
```

You can also put arbitrary expressions on the right-hand side of an assignment statement:

```
percentage = (minute * 100) / 60
```

This ability may not seem impressive now, but you will see other examples where composition makes it possible to express complex computations neatly and concisely.

Warning: There are limits on where you can use certain expressions. For example, the left-hand side of an assignment statement has to be a *variable* name, not an expression. So, the following is illegal: minute+1 = hour.

1.2.10 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 they are marked 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     # caution: integer division
```

Everything from the # to the end of the line is ignored — it has no effect on the program. The message is intended for the programmer or for future programmers who might use this code. In this case, it reminds the reader about the ever-surprising behavior of integer division.

This sort of comment is less necessary if you use the integer division operation, //. It has the same effect as the division operator * Note, but it signals that the effect is deliberate.

percentage = (minute * 100) // 60

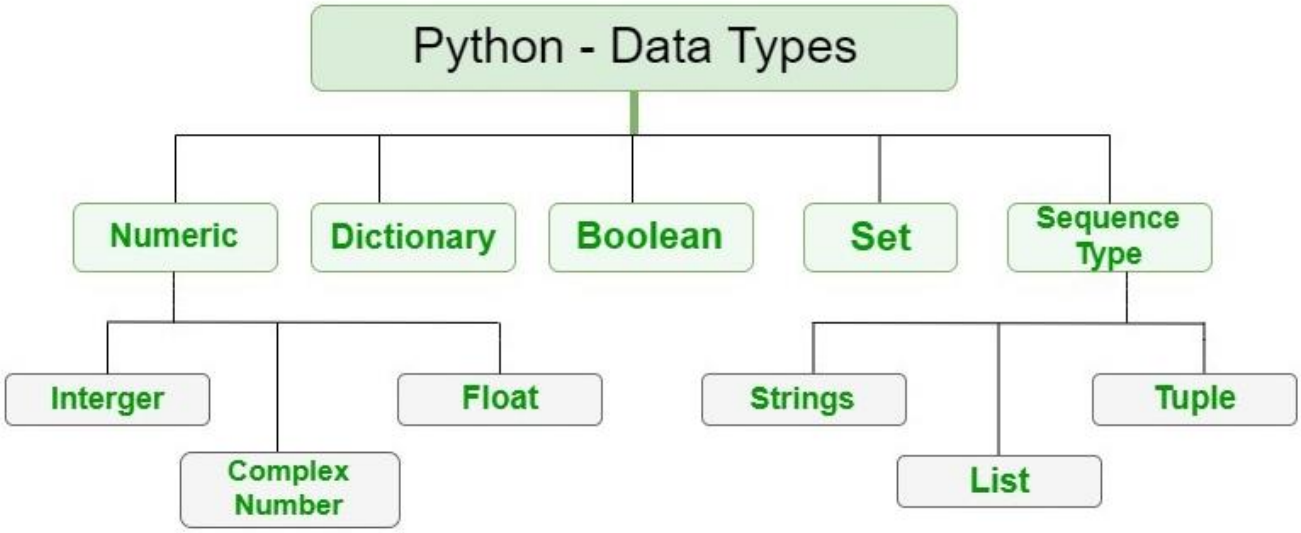
The integer division operator is like a comment that says, "I know this is integer division, and I like it that way!"

1.3 DATA TYPES

Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.

Following are the standard or built-in data type of Python:

- Numeric
- Sequence Type
- Boolean
- Set
- Dictionary



Numeric

In Python, numeric data type represents the data which has numeric value. Numeric value can be integer, floating number or even complex numbers. These values are defined as int, float and complex class in Python.

- **Integers** – This value is represented by int class. It contains positive or negative whole numbers (without fraction or decimal). In Python there is no limit to how long an integer value can be.

- **Float** – This value is represented by float class. It is a real number with floating point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation.
- **Complex Numbers** – Complex number is represented by complex class. It is specified as *(real part) + (imaginary part)j*. For example – 2+3j

Note – type() function is used to determine the type of data type.

```
# Python program to

# demonstrate numeric value

a = 5

print("Type of a: ", type(a))

b = 5.0

print("\nType of b: ", type(b))

c = 2 + 4j

print("\nType of c: ", type(c))
```

OUTPUT

```
Type of a:  <class 'int'>

Type of b:  <class 'float'>

Type of c:  <class 'complex'>
```

Sequence Type

In Python, sequence is the ordered collection of similar or different data types. Sequences allows to store multiple values in an organized and efficient fashion. There are several sequence types in Python –

- String
- List
- Tuple

1) String

In Python, Strings are arrays of bytes representing Unicode characters. A string is a collection of one or more characters put in a single quote, double-quote or triple quote. In python there is no character data type, a character is a string of length one. It is represented by str class.

Creating String

Strings in Python can be created using single quotes or double quotes or even triple quotes.

```
# Python Program for

# Creation of String

# Creating a String

# with single Quotes

String1 = 'Welcome to the Geeks World'
```

```
print("String with the use of Single Quotes: ")

print(String1)

# Creating a String

# with double Quotes

String1 = "I'm a Geek"

print("\nString with the use of Double Quotes: ")

print(String1)

print(type(String1))

# Creating a String

# with triple Quotes

String1 = '''I'm a Geek and I live in a world of "Geeks"'''

print("\nString with the use of Triple Quotes: ")

print(String1)

print(type(String1))

# Creating String with triple

# Quotes allows multiple lines

String1 = '''Geeks

        For

        Life'''

print("\nCreating a multiline String: ")

print(String1)
```

OUTPUT

```
String with the use of Single Quotes:
Welcome to the Geeks World
String with the use of Double Quotes:
I'm a Geek
<class 'str'>
String with the use of Triple Quotes:
I'm a Geek and I live in a world of "Geeks"
<class 'str'>
```

Creating a multiline String:

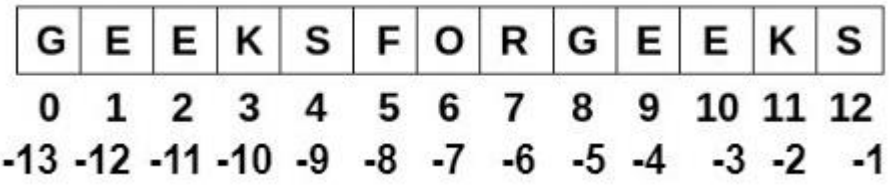
Geeks

For

Life

Accessing elements of String

In Python, individual characters of a String can be accessed by using the method of Indexing. Indexing allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character and so on.




```
# Python Program to Access

# Characters of String

String1 = "GeeksForGeeks"

print("Initial String: ")

print(String1)

# Printing First character

print("\nFirst character of String is: ")

print(String1[0])

# Printing Last character

print("\nLast character of String is: ")

print(String1[-1])
```

OUTPUT

```
Initial String:

GeeksForGeeks

First character of String is:
G
Last character of String is:
S
```

2) List

Lists are just like the arrays, declared in other languages which is a ordered collection of data. It is very flexible as the items in a list do not need to be of the same type.

Creating List

Lists in Python can be created by just placing the sequence inside the square brackets[].

```
# Python program to demonstrate

# Creation of List

# Creating a List

List = []

print("Initial blank List: ")

print(List)

# Creating a List with

# the use of a String

List = ['GeeksForGeeks']

print("\nList with the use of String: ")

print(List)

# Creating a List with

# the use of multiple values

List = ["Geeks", "For", "Geeks"]

print("\nList containing multiple values: ")

print(List[0])

print(List[2])

# Creating a Multi-Dimensional List

# (By Nesting a list inside a List)

List = [['Geeks', 'For'], ['Geeks']]

print("\nMulti-Dimensional List: ")

print(List)
```

OUTPUT

```
Initial blank List:

[]

List with the use of String:

['GeeksForGeeks']

List containing multiple values:

Geeks

Geeks

Multi-Dimensional List:

[['Geeks', 'For'], ['Geeks']]
```

Accessing elements of List

In order to access the list items refer to the index number. Use the index operator [ ] to access an item in a list. In Python, negative sequence indexes represent positions from the end of the array. Instead of having to compute the offset as in List[len(List)-3], it is enough to just write List[-3]. Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second-last item, etc.

```
# Python program to demonstrate

# accessing of element from list

# Creating a List with

# the use of multiple values

List = ["Geeks", "For", "Geeks"]

# accessing a element from the

# list using index number

print("Accessing element from the list")

print(List[0])

print(List[2])

# accessing a element using

# negative indexing

print("Accessing element using negative indexing")

# print the last element of list

print(List[-1])

# print the third last element of list

print(List[-3])
```

OUTPUT

```
Accessing element from the list

Geeks

Geeks

Accessing element using negative indexing

Geeks

Geeks
```

3) Tuple

Just like list, tuple is also an ordered collection of Python objects. The only difference between tuple and list is that tuples are immutable i.e. tuples cannot be modified after it is created. It is represented by tuple class.

Creating Tuple

In Python, tuples are created by placing a sequence of values separated by 'comma' with or without the use of parentheses for grouping of the data sequence. Tuples can contain any number of elements and of any datatype (like strings, integers, list, etc.).

Note: Tuples can also be created with a single element, but it is a bit tricky. Having one element in the parentheses is not sufficient; there must be a trailing 'comma' to make it a tuple.

```
# Python program to demonstrate

# creation of Set

# Creating an empty tuple

Tuple1 = ()

print("Initial empty Tuple: ")

print (Tuple1)

# Creating a Tuple with

# the use of Strings

Tuple1 = ('Geeks', 'For')

print("\nTuple with the use of String: ")

print(Tuple1)

# Creating a Tuple with

# the use of list

list1 = [1, 2, 4, 5, 6]

print("\nTuple using List: ")

print(tuple(list1))

# Creating a Tuple with the

# use of built-in function

Tuple1 = tuple('Geeks')

print("\nTuple with the use of function: ")

print(Tuple1)

# Creating a Tuple

# with nested tuples

Tuple1 = (0, 1, 2, 3)

Tuple2 = ('python', 'geek')
```

```
Tuple3 = (Tuple1, Tuple2)

print("\nTuple with nested tuples: ")

print(Tuple3)
```

OUTPUT

```
Initial empty Tuple:
()
Tuple with the use of String:
('Geeks', 'For')
Tuple using List:
(1, 2, 4, 5, 6)
Tuple with the use of function:
('G', 'e', 'e', 'k', 's')
Tuple with nested tuples:
((0, 1, 2, 3), ('python', 'geek'))
```

Note – Creation of Python tuple without the use of parentheses is known as Tuple Packing.

Accessing elements of Tuple

In order to access the tuple items refer to the index number. Use the index operator[ ] to access an item in a tuple. The index must be an integer. Nested tuples are accessed using nested indexing.

```
# Python program to

# demonstrate accessing tuple

tuple1 = tuple([1, 2, 3, 4, 5])

# Accessing element using indexing

print("First element of tuple")

print(tuple1[0])

# Accessing element from last

# negative indexing

print("\nLast element of tuple")

print(tuple1[-1])

print("\nThird last element of tuple")

print(tuple1[-3])
```

OUTPUT

```
First element of tuple
1
Last element of tuple
5
Third last element of tuple 3
```

Boolean

Data type with one of the two built-in values, True or False. Boolean objects that are equal to True are truthy (true), and those equal to False are falsy (false). But non-Boolean objects can be evaluated in Boolean context as well and determined to be true or false. It is denoted by the class bool.

Note – True and False with capital 'T' and 'F' are valid booleans otherwise python will throw an error.

```
# Python program to
# demonstrate boolean type
print(type(True))
print(type(False))
print(type(true))
```

OUTPUT

```
<class 'bool'>

<class 'bool'>

Traceback (most recent call last):

  File "/home/7e8862763fb66153d70824099d4f5fb7.py", line 8, in

    print(type(true))

NameError: name 'true' is not defined
```

Set

In Python, Set is an unordered collection of data type that is iterable, mutable and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements.

Creating Sets

Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by 'comma'. Type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

```
# Python program to demonstrate
# Creation of Set in Python
# Creating a Set
set1 = set()
print("Initial blank Set: ")
print(set1)

# Creating a Set with
# the use of a String
set1 = set("GeeksForGeeks")
print("\nSet with the use of String: ")
print(set1)

# Creating a Set with
# the use of a List
set1 = set(["Geeks", "For", "Geeks"])
print("\nSet with the use of List: ")
print(set1)
```

```
# Creating a Set with
# a mixed type of values
# (Having numbers and strings)
set1 = set([1, 2, 'Geeks', 4, 'For', 6, 'Geeks'])
print("\nSet with the use of Mixed Values")
print(set1)
```

OUTPUT
Initial blank Set:
set()
Set with the use of String:
{'F', 'o', 'G', 's', 'r', 'k', 'e'}
Set with the use of List:
{'Geeks', 'For'}
Set with the use of Mixed Values
{1, 2, 4, 6, 'Geeks', 'For'}

Accessing elements of Sets

Set items cannot be accessed by referring to an index, since sets are unordered the items has no index. But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

```
# Python program to demonstrate
# Accessing of elements in a set
# Creating a set
set1 = set(["Geeks", "For", "Geeks"])
print("\nInitial set")
print(set1)
# Accessing element using
# for loop
print("\nElements of set: ")
for i in set1:
        print(i, end =" ")
# Checking the element
# using in keyword
print("Geeks" in set1)
```

OUTPUT
Initial set:

{'Geeks', 'For'}

Elements of set:

Geeks For

True

Dictionary

Dictionary in Python is an unordered collection of data values, used to store data values like a map, which unlike other Data Types that hold only single value as an element, Dictionary holds key:value pair. Key-value is provided in the dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon :, whereas each key is separated by a ‘comma’.

Creating Dictionary

In Python, a Dictionary can be created by placing a sequence of elements within curly {} braces, separated by 'comma'. Values in a dictionary can be of any datatype and can be duplicated, whereas keys can't be repeated and must be immutable. Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing it to curly braces{}.

Note – Dictionary keys are case sensitive, same name but different cases of Key will be treated distinctly.

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)
# Creating a Dictionary
# with Integer Keys
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)
# Creating a Dictionary
# with Mixed keys
Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)
# Creating a Dictionary
# with dict() method
Dict = dict({1: 'Geeks', 2: 'For', 3:'Geeks'})
print("\nDictionary with the use of dict(): ")
print(Dict)
# Creating a Dictionary
# with each item as a Pair
Dict = dict([(1, 'Geeks'), (2, 'For')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

OUTPUT

Empty Dictionary:

{}

Dictionary with the use of Integer Keys:

{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Dictionary with the use of Mixed Keys:

{1: [1, 2, 3, 4], 'Name': 'Geeks'}

Dictionary with the use of dict():

{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Dictionary with each item as a pair:

{1: 'Geeks', 2: 'For'}

Accessing elements of Dictionary

In order to access the items of a dictionary refer to its key name. Key can be used inside square brackets. There is also a method called get() that will also help in accessing the element from a dictionary.

```
# Python program to demonstrate
```

```
# accessing a element from a Dictionary

# Creating a Dictionary

Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# accessing a element using key

print("Accessing a element using key:")

print(Dict['name'])

# accessing a element using get()

# method

print("Accessing a element using get:")

print(Dict.get(3))
```

OUTPUT

Accessing a element using key:

For

Accessing a element using get:

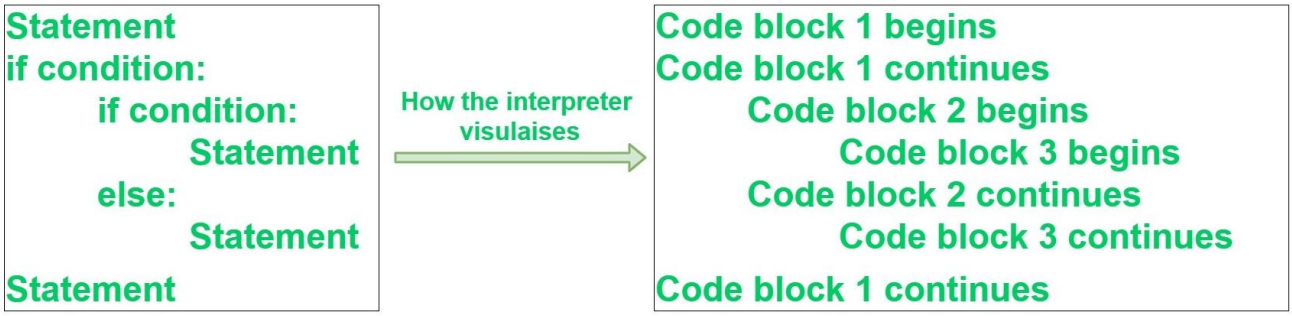
Geeks

1.4 INDENTATION

Indentation is a very important concept of Python because without properly indenting the Python code, you will end up seeing IndentationError and the code will not get compiled.

Python Indentation

Python indentation refers to adding white space before a statement to a particular block of code. In another word, all the statements with the same space to the right, belong to the same code block.



Example 1

```
# Python program showing indentation

site = 'gfg'
if site == 'gfg':
        print('Logging on to geeksforgeeks...')
else:
        print('retype the URL.')
print('All set !')
```

EXAMPLE 2

```
j = 1

while(j<= 5):
        print(j)
        j = j + 1
```

OUTPUT

```
1

2

3

4

5
```

1.5 TYPE CONVERSIONS

Python defines type conversion functions to directly convert one data type to another which is useful in day-to-day and competitive programming. This article is aimed at providing information about certain conversion functions.

There are two types of Type Conversion in Python:

1. Implicit Type Conversion
2. Explicit Type Conversion

Implicit Type Conversion

In Implicit type conversion of data types in Python, the Python interpreter automatically converts one data type to another without any user involvement. To get a more clear view of the topic see the below examples.

```
x = 10
print("x is of type:",type(x))
y = 10.6
print("y is of type:",type(y))
z = x + y
print(z)
print("z is of type:",type(z))
```

OUTPUT

```
x is of type: <class 'int'>
y is of type: <class 'float'>
20.6
z is of type: <class 'float'>
```

Explicit Type Conversion

In Explicit Type Conversion in Python, the data type is manually changed by the user as per their requirement. With explicit type conversion, there is a risk of data loss since we are forcing an expression to be changed in some specific data type. Various forms of explicit type conversion are explained below:

1. int(a, base): This function converts **any data type to integer**. 'Base' specifies the **base in which string is** if the data type is a string.
2. float(): This function is used to convert **any data type to a** floating-point **number.**

```
# Python code to demonstrate Type conversion
# using int(), float()
# initializing string
s = "10010"

# printing string converting to int base 2
c = int(s,2)
print ("After converting to integer base 2 : ", end="")
print (c)

# printing string converting to float
e = float(s)
print ("After converting to float : ", end="")
print (e)
```

OUTPUT

```
After converting to integer base 2 : 18
After converting to float : 10010.0
```

3. ord() : This function is used to convert a **character to integer.**
4. hex() : This function is to convert **integer to hexadecimal string**.
5. oct() : This function is to convert **integer to octal string**.

```
# Python code to demonstrate Type conversion
# using ord(), hex(), oct()
# initializing integer
s = '4'
# printing character converting to integer
c = ord(s)
print ("After converting character to integer : ",end="")
print (c)

# printing integer converting to hexadecimal string
c = hex(56)
print ("After converting 56 to hexadecimal string : ",end="")
print (c)

# printing integer converting to octal string
c = oct(56)
print ("After converting 56 to octal string : ",end="")
print (c)
```

OUTPUT

```
After converting character to integer : 52

After converting 56 to hexadecimal string : 0x38

After converting 56 to octal string : 0o70
```

6. tuple() : This function is used to **convert to a tuple**.
7. set() : This function returns the **type after converting to set**.
8. list() : This function is used to convert **any data type to a list type**.

```
# Python code to demonstrate Type conversion
# using tuple(), set(), list()
# initializing string
s = 'geeks'
# printing string converting to tuple
c = tuple(s)
print ("After converting string to tuple : ",end="")
print (c)

# printing string converting to set
c = set(s)
print ("After converting string to set : ",end="")
print (c)

# printing string converting to list
c = list(s)
print ("After converting string to list : ",end="")
print (c)
```

OUTPUT

After converting string to tuple : ('g', 'e', 'e', 'k', 's')

After converting string to set : {'k', 'e', 's', 'g'}

After converting string to list : ['g', 'e', 'e', 'k', 's']

9. dict() : This function is used to **convert a tuple of order (key,value) into a dictionary**.
10. str() : Used to **convert integer into a string.**
11. complex(real,imag) : This function **converts real numbers to complex(real,imag) number.**

```
# Python code to demonstrate Type conversion
# using dict(), complex(), str()
# initializing integers
a = 1
b = 2

# initializing tuple
tup = (('a', 1) ,('f', 2), ('g', 3))

# printing integer converting to complex number
c = complex(1,2)
print ("After converting integer to complex number : ",end="")
print (c)

# printing integer converting to string
c = str(a)
print ("After converting integer to string : ",end="")
print (c)

# printing tuple converting to expression dictionary
c = dict(tup)
print ("After converting tuple to dictionary : ",end="")
print (c)
```

OUTPUT

After converting integer to complex number : (1+2j)

After converting integer to string : 1

After converting tuple to dictionary : {'a': 1, 'f': 2, 'g': 3}

12. chr(number): This function **converts number to its corresponding ASCII character.**

```
# Convert ASCII value to characters
a = chr(76)
b = chr(77)

print(a)
print(b)
```

OUTPUT

L

M

1.6 Type( ) function

Type() method returns class type of the argument(object) passed as parameter in Python.

Syntax of type() function

Syntax: *type(object, bases, dict)*

Parameters :

- **object:** Required. If only one parameter is specified, the type() function returns the type of this object
- **bases :** tuple of classes from which the current class derives. Later corresponds to the __bases__ attribute.
- **dict :** a dictionary that holds the namespaces for the class. Later corresponds to the __dict__ attribute.

Return: returns a new type class or essentially a metaclass.

```
a = ("Geeks", "for", "Geeks")
b = ["Geeks", "for", "Geeks"]
c = {"Geeks": 1, "for":2, "Geeks":3}
d = "Hello World"
e = 10.23
f = 11.22

print(type(a))
print(type(b))
print(type(c))
print(type(d))
print(type(e))
print(type(f))
```

OUTPUT

<class 'tuple'>

<class 'list'>

<class 'dict'>

<class 'str'>

<class 'float'>

<class 'float'>

What is a type() function in Python?

The type() function is mostly used for debugging purposes. Two different types of arguments can be passed to type() function, single and three arguments. If a single argument type(obj) is passed, it returns the type of the given object. If three arguments type(object, bases, dict) is passed, it returns a new type object.

Applications:

- **type()** function is basically used for debugging purposes. When using other string functions like .upper(), .lower(), .split() with text extracted from a web crawler, it might not work because they might be of different type which doesn't support string functions. And as a result, it will keep throwing errors, which are very difficult to debug [Consider the error as Generator Type has no attribute lower()].
- **type()** function can be used at that point to determine the type of text extracted and then change it to other forms of string before we use string functions or any other operations on it.
- **type()** with three arguments can be used to dynamically initialize classes or existing classes with attributes. It is also used to register database tables with SQL.

1.7 DECISION CONTROL FLOW STATEMENTS

Decision-making statements in programming languages decide the direction of the flow of program execution.

In Python, if-else elif statement is used for decision making.

if statement
if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

example:
```
# python program to illustrate If statement

i = 10
if (i > 15):
        print("10 is less than 15")
print("I am Not in if")
```

OUTPUT
```
I am Not in if
```

if-else
The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the *else* statement. We can use the *else* statement with *if* statement to execute a block of code when the condition is false.

Example:
```
# python program to illustrate If else statement
#!/usr/bin/python
```

```
i = 20
if (i < 15):
        print("i is smaller than 15")
        print("i'm in if Block")
else:
        print("i is greater than 15")
        print("i'm in else Block")
print("i'm not in if and not in else Block")
```

OUTPUT

```
i is greater than 15
i'm in else Block
i'm not in if and not in else Block
```

nested-if

A nested if is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement. Yes, Python allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

```
# python program to illustrate nested If statement
#!/usr/bin/python
i = 10
if (i == 10):

        # First if statement
        if (i < 15):
                print("i is smaller than 15")

        # Nested - if statement
        # Will only be executed if statement above
        # it is true
        if (i < 12):
                print("i is smaller than 12 too")
        else:
                print("i is greater than 15")
```

OUTPUT:

```
i is smaller than 15

i is smaller than 12 too
```

if-elif-else ladder

Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

```
# Python program to illustrate if-elif-else ladder
#!/usr/bin/python
i = 20
if (i == 10):
        print("i is 10")
```

```
elif (i == 15):
        print("i is 15")
elif (i == 20):
        print("i is 20")
else:
        print("i is not present")
```

OUTPUT

i is 20

1.8 WHILE LOOP

While Loop is used to execute a block of statements repeatedly until a given condition is satisfied. And when the condition becomes false, the line immediately after the loop in the program is executed.

Syntax:

while expression:

  statement(s)

Example 1:

```
# Python program to illustrate
# while loop
count = 0
while (count < 3):
        count = count + 1
        print("Hello Geek")
```

OUTPUT

Hello Geek

Hello Geek

Hello Geek

Example 2:

```
# checks if list still
# contains any element
a = [1, 2, 3, 4]

while a:
        print(a.pop())
```

OUTPUT

4

3

2

1

1.9 FOR LOOP

For loops are used for sequential traversal. For example: traversing a list or string or array etc. In Python, there is no C style for loop, i.e., for (i=0; i<n; i++). There is "for in" loop which is similar to for each loop in other languages. Let us learn how to use for in loop for sequential traversals.

Syntax:

```
for iterator_var in sequence:
    statements(s)
```

Example 1:

```
# Python program to illustrate
# Iterating over range 0 to n-1
n = 4
for i in range(0, n):
        print(i)
```

OUTPUT:

```
0
1
2
3
```

Example 2:

```
# Python program to illustrate
# Iterating over a list
print("List Iteration")
l = ["geeks", "for", "geeks"]
for i in l:
        print(i)

# Iterating over a tuple (immutable)
print("\nTuple Iteration")
t = ("geeks", "for", "geeks")
for i in t:
        print(i)

# Iterating over a String
print("\nString Iteration")
s = "Geeks"
for i in s:
        print(i)

# Iterating over dictionary
print("\nDictionary Iteration")
d = dict()
d['xyz'] = 123
d['abc'] = 345
for i in d:
        print("%s %d" % (i, d[i]))
```

```
# Iterating over a set
print("\nSet Iteration")
set1 = {1, 2, 3, 4, 5, 6}
for i in set1:
        print(i),
```

OUTPUT

```
List Iteration
geeks
for
geeks

Tuple Iteration
geeks
for
geeks

String Iteration
G
e
e
k
s

Dictionary Iteration
xyz  123
abc  345
```

Example 3:

```
# Python program to illustrate

# Iterating by index

list = ["geeks", "for", "geeks"]

for index in range(len(list)):

    print list[index]
```

OUTPUT

```
geeks

for

geeks
```

1.10 The Continue and break statements

Break:

- The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

- If the break statement is inside a nested loop (loop inside another loop), the break statement will terminate the innermost loop.

Example :

```
for val in "string":

    if val == "i":

        break

    print(val)

print("The end")
```

OUTPUT

s

t

r

The end

Continue:

- The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

Example:

```
for val in "string":

    if val == "i":

        continue

    print(val)

print("The end")
```

OUTPUT

s

t

r

n

g

The end

1.11 Built-in Functions

Python has some built in functions

Function	Description
abs()	Returns the absolute value of a number
all()	Returns True if all items in an iterable object are true
any()	Returns True if any item in an iterable object is true
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin()	Returns the binary version of a number
bool()	Returns the boolean value of the specified object
bytearray()	Returns an array of bytes
bytes()	Returns a bytes object
callable()	Returns True if the specified object is callable, otherwise False
chr()	Returns a character from the specified Unicode code.

classmethod()	Converts a method into a class method
compile()	Returns the specified source as an object, ready to be executed
complex()	Returns a complex number
delattr()	Deletes the specified attribute (property or method) from the specified object
dict()	Returns a dictionary (Array)
dir()	Returns a list of the specified object's properties and methods
divmod()	Returns the quotient and the remainder when argument1 is divided by argument2
enumerate()	Takes a collection (e.g. a tuple) and returns it as an enumerate object
eval()	Evaluates and executes an expression
exec()	Executes the specified code (or object)
filter()	Use a filter function to exclude items in an iterable object
float()	Returns a floating point number

format()	Formats a specified value
frozenset()	Returns a frozenset object
getattr()	Returns the value of the specified attribute (property or method)
globals()	Returns the current global symbol table as a dictionary
hasattr()	Returns True if the specified object has the specified attribute (property/method)
hash()	Returns the hash value of a specified object
help()	Executes the built-in help system
hex()	Converts a number into a hexadecimal value
id()	Returns the id of an object
input()	Allowing user input
int()	Returns an integer number
isinstance()	Returns True if a specified object is an instance of a specified object

issubclass()	Returns True if a specified class is a subclass of a specified object
iter()	Returns an iterator object
len()	Returns the length of an object
list()	Returns a list
locals()	Returns an updated dictionary of the current local symbol table
map()	Returns the specified iterator with the specified function applied to each item
max()	Returns the largest item in an iterable
memoryview()	Returns a memory view object
min()	Returns the smallest item in an iterable
next()	Returns the next item in an iterable
object()	Returns a new object
oct()	Converts a number into an octal

open()	Opens a file and returns a file object
ord()	Convert an integer representing the Unicode of the specified character
pow()	Returns the value of x to the power of y
print()	Prints to the standard output device
property()	Gets, sets, deletes a property
range()	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
repr()	Returns a readable version of an object
reversed()	Returns a reversed iterator
round()	Rounds a numbers
set()	Returns a new set object
setattr()	Sets an attribute (property/method) of an object
slice()	Returns a slice object

sorted()	Returns a sorted list
staticmethod()	Converts a method into a static method
str()	Returns a string object
sum()	Sums the items of an iterator
super()	Returns an object that represents the parent class
tuple()	Returns a tuple
type()	Returns the type of an object
vars()	Returns the __dict__ property of an object
zip()	Returns an iterator, from two or more iterators

1.12 Function definition and calling the function

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Creating a Function

In Python a function is defined using the def keyword:

Example

```
def my_function():
  print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example

```
def my_function():
  print("Hello from a function")
my_function()
```

OUTPUT

```
Hello from a function
```

Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

OUTPUT:

```
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

With No. of arguments:

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

OUTPUT

```
Emil Refsnes
```

1.13 Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

Example

If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

OUTPUT

```
The youngest child is Linus
```

1.14 Keyword Arguments

You can also send arguments with the *key* = *value* syntax.

This way the order of the arguments does not matter.

Example

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

OUTPUT

```
The youngest child is Linus
```

1.15 Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

This way the function will receive a *dictionary* of arguments, and can access the items accordingly:

Example

If the number of keyword arguments is unknown, add a double ** before the parameter name:

```
def my_function(**kid):
  print("His last name is " + kid["lname"])


my_function(fname = "Tobias", lname = "Refsnes")
```

OUTPUT:

```
His last name is Refsnes
```

****