1 Variables

In Python, variables are declared and initialized by simply assigning a value to it. The type of the variable is determined by the value assigned to it. For example, the following code declares and initializes a variable named x to the value 5. The type of the variable x is an integer.

```
x = 5 # x is an integer
```

1.1 Rules of naming variables

Variables must be named according to the following rules:

- Variable names must start with a letter or an underscore.
- The remainder of your variable name may consist of letters, numbers and underscores.
- Names are case sensitive.
- Names must not be a reserved word.

Here are some examples of legal and illegal variable names:

```
# Examples of valid variable name
name = "Bob"
_my_name = "Bob"
myName = "Bob"
MYNAME = "Bob" # Note: myName and MYNAME are different variables
age = 23
height = 1.73
is_human = True
name2 = "Alice"

# Examples of invalid variable name
2name = "Bob"
my-name = "Bob"
my name = "Bob"
```

1.2 Reserved Words

The following words are reserved by Python and cannot be used as variable names:

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	

2 Python Data Types

Here are some basic data types in Python:

- Text Type: str
- Numeric Types: int, float
- Boolean Type: bool

2.1 Text Type

Type	Description	Example
str	String literals in python are surrounded by either single quotation marks, or double quotation marks.	'hello' or "hello"

2.2 Numeric Types

Type	Description	Example
int	Integers is a whole number, positive or negative, without decimals, of unlimited length.	x = 1
float	Float is a number, positive or negative, containing one or more decimals.	x = 1.0

2.3 Boolean Type

Type	Description	Example
bool	Booleans represent one of two values: True or False.	x = True

2.4 Type Conversion

You can convert from one type to another with the int(), float(), and str() methods.

Method	Description	Example	Result
int()	Converts a string to an integer.	x = int("1")	x = 1
float()	Converts a string to a float.	x = float("1.0")	x = 1.0
str()	Converts an integer to a string.	x = str(1)	x = "1"

3 Operators

In this section, x and y are assigned as follow:

```
x = 15
y = 2
```

3.1 Arithmetic Operator

Operator	Description	Example	Result
+	Addition	x + y	17
-	Subtraction	x - y	13
*	Multiplication	x * y	30
/	Division	x / y	7.5
%	Modulus	x % y	1
//	Floor Division	x // y	7
**	Exponentiation	x ** y	225

3.2 Assignment Operator

Operator	Description	Example	Result
=	Assign	x = 5	x = 5
+=	Add	x += 5	x = 20
-=	Subtract	x -= 5	x = 10
*=	Multiply	x *= 5	x = 75
/=	Divide	x /= 5	x = 3
%=	Modulus	x %= 5	x = 0
//=	Floor Division	x //= 5	x = 3
**=	Exponentiation	x **= 5	x = 759375

3.3 Comparison Operators

Comparison Operators (a.k.a Relational Operators) are used to compare values and create a conditional statement. It either returns True or False according to the condition.

Operator	Description	Example	Result
==	Equal	x == y	False
!=	Not Equal	x != y	True
>	Greater Than	x > y	True
<	Less Than	x < y	False
>=	Greater Than or Equal to	x >= y	True
<=	Less Than or Equal to	x <= y	False

3.4 Logical Operators

Logical operators are used to combine conditional statements. It either returns True or False according to the condition.

Operator	Description	Example	Result
and	Returns True if both statements are true	x < 5 and x < 10	True
or	Returns True if one of the statements is true	x < 5 or x < 4	True
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)	False

4 Conditions

4.1 If Statement

If Statement is used to check if a condition is true or false. If the condition is true, it will execute the code inside the if statement. If the condition is false, it will not execute the code inside the if statement.

Syntax

```
if condition:
    statement # The statement will execute if the condition is true
```

Example

```
A = 100
if A > 0:
    print("A is positive") # Note, This line is indented
```

4.2 If...Else Statement

If...Else Statement is used to check if a condition is true or false. If the condition is true, it will execute the code inside the if statement. If the condition is false, it will execute the code inside the else statement.

Syntax

```
if condition:
    statement # The statement will execute if the condition is true
else:
    statement # The statement will execute if the condition is false
```

Example

```
A = 100
if A > 0:
    print("A is positive") # Note: This line is indented
else:
    print("A is negative") # Note: This line is indented
```

4.3 If...Elif...Else Statement

If...Elif...Else Statement is used to check if a condition is true or false. If the condition is true, it will execute the code inside the if statement. If the condition is false, it will check the next condition. If the next condition is true, it will execute the code inside the elif statement. If the next condition is false, it will execute the code inside the else statement.

Syntax

```
if condition:
    statement # The statement will execute if the condition is true
elif condition:
    statement # The statement will execute if the condition is true but the
   first condition is false
else:
    statement # The statement will execute if the condition is false
```

Example

```
A = 100
if A > 0:
    print("A is positive") # Note: This line is indented
elif A == 0:
    print("A is zero") # Note: This line is indented
else:
    print("A is negative") # Note: This line is indented
```

4.4 Nested If Statement

A nested if statement contains an if statement inside another if statement. The inner if statement will be executed if the outer if statement is true.

Syntax

```
if condition:
    if condition:
        statement # The statement will execute if all the conditions are true
```

```
A = 100
if A > 0:
    print("A is positive")
    if A > 50:
        print("A is greater than 50")
```

4.5 Reminder

- The if statements are top down approach (if the first condition is true, it will not check the next condition)
- The condition of if/elif statements must not be empty
- All the statement must be indented
- The size of indentation does not matter, but all the indentations must be the same size
- The if statement can be used without else statement (but not vice versa)
- The elif statement can be used without else statement (but not vice versa)

5 User Input

5.1 Input Function

The input function is used to get input from the user. The input function will return a string value.

Syntax

```
variable = input(Message) # The message will be displayed to the user, but it is optional
```

Example

```
# This program is a simple example of how to use the input function
# to get the user's name and say hi with his/her name
name = input("What is your name? ")
print("Hello", name)
```

5.2 Multiple Input

There are different methods to get multiple inputs from the user. Here one of the methods.

Syntax

```
variable1, variable2 = input(Message).split() # The message will be displayed to the user, but it is optional
```

Example

```
# This program is an example of multiple input
name, age = input("What is your name and age? ").split()
print("Hello", name, "you are", age, "years old")
```

6 Loops

In python there are 2 types of loops, for loop and while loop. The for loop is used to iterate over a sequence (list, tuple, string) or other iterable objects. The while loop is used to iterate over a block of code as long as the test expression (condition) is true.

6.1 For Loop

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

6.1.1 Syntax

```
for val in sequence:
    Body of for
```

Usually we will use the for loop with the range() function to iterate over a sequence of numbers. The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

```
# This program prints all the numbers from 0 to 10
for i in range(11): # It only includes the ending parameter
    print(i)
```

If we want to print the numbers from 1 to 10, we can use the range() function with the starting and ending parameters.

Example

```
# This program prints all the numbers from 1 to 10
for i in range(1,11): # The starting parameter is 1 and the ending parameter is
    11
    print(i)
```

If we want to print the numbers from 1 to 10 with a step of 2, we can use the range() function with the starting, ending and step parameters.

Example

```
# This program prints all the numbers from 1 to 10 with a step of 2
for i in range(1,11,2): # The step is 2
    print(i)
```

We can also use the range() function with a negative step to print the numbers in reverse order.

Example

```
# This program prints all the numbers from 10 to 1
for i in range(10,0,-1): # The step is -1
    print(i)
```

Usually we use for loop when we know the number of times to iterate. But sometimes we don't know the number of times to iterate, in such cases we use while loop.

6.2 While Loop

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

6.2.1 Syntax

```
while test_expression:
    Body of while
```

Example

```
# This program prints all the numbers from 0 to 10
i = 0 # First we set the variable i to 0
while i < 11: # Then we check if i is less than 11
    print(i) # If it is, we print i
    i += 1 # Then we add 1 to i
```

6.3 Break and Continue

6.3.1 Break

The break statement is used to exit or terminate a loop.

Example

```
# This program prints all the numbers from 0 to 5
i = 0 # First we set the variable i to 0
while i < 11: # Then we check if i is less than 11
    print(i) # If it is, we print i
    if i == 5: # If i is equal to 5
        break # We break the loop
    i += 1 # Then we add 1 to i
```

6.3.2 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

```
# This program prints all the numbers from 0 to 10 except 5
i = 0 # First we set the variable i to 0
while i < 11: # Then we check if i is less than 11
    i += 1 # Then we add 1 to i
    if i == 5: # If i is equal to 5
        continue # We continue to the next iteration
    print(i) # If i is not equal to 5, we print i
```

7 String

Before dive deep in Python string, we need to have a concept of array. An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array). In python, strings are arrays of bytes representing unicode characters. However, python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string.

```
a = "Hello, World!"
print(a[1]) # e
```

In the above example, the word "Hello, World!" is stored in the variable a. The first character of the string is accessed using the array index 1. The index starts from 0. So, the character of index 1 is "e". The index must be an integer. We can not use float or other types, this will result into errors. For a better visulization of the string, we can refer to the following figure.

0	1	2	3	4	5	6	7	8	9	10	11	12
H	e	l	l	o	,		W	o	r	l	d	!

7.1 String Functions

7.1.1 len()

The `len()` function returns the length of a string.

```
a = "Hello, World!"
print(len(a)) # 13
```

7.1.2 strip()

The `strip()` method removes any whitespace from the beginning or the end.

```
a = " Hello, World! "
print(a.strip()) # "Hello, World!"
```

7.1.3 lower()

The `lower()` method returns the string in lower case.

```
a = "Hello, World!"
print(a.lower()) # "hello, world!"
```

7.1.4 upper()

The `upper()` method returns the string in upper case.

```
a = "Hello, World!"
print(a.upper()) # "HELLO, WORLD!"
```

7.1.5 replace()

The `replace()` method replaces a string with another string.

```
a = "Hello, World!"
print(a.replace("H", "J")) # "Jello, World!"
```

7.1.6 split()

The `split()` method splits the string into substrings if it finds instances of the separator. The default separator is any whitespace.

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

7.1.7 format()

The `format()` method takes the passed arguments, formats them, and places them in the string where the placeholders {} are.

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age)) # My name is John, and I am 36
```

7.1.8 check string

To check if a certain phrase or character is present in a string, we can use the keywords `in` or `not in`.

```
txt = "The rain in Spain stays mainly in the plain"
x = "ain" in txt
print(x) # True
```

7.1.9 concatenation

To concatenate, or combine, two strings you can use the `+` operator.

```
a = "Hello"
b = "World"
c = a + b
print(c) # HelloWorld
```

To add a space between them, add a ” ”:

```
a = "Hello"
b = "World"
c = a + " " + b
print(c) # Hello World
```

7.2 string slicing

You can return a range of characters by using the slice syntax. Specify the start index and the end index, separated by a colon, to return a part of the string.

```
a = "Hello, World!"
print(a[2:5]) # llo
```

7.2.1 string slicing with negative index

Specify negative indexes if you want to start the slice from the end of the string.

```
a = "Hello, World!"
print(a[-5:-2]) # orl
```

7.2.2 string slicing with step

Specify the step of the slice. The step specifies the increment between the indexes to slice.

```
a = "Hello, World!"
print(a[::2]) # HloWrd
```

7.2.3 string slicing with negative step

Specify negative step to slice the string in reverse order.

```
a = "Hello, World!"
print(a[::-1]) # !dlroW ,olleH
```

7.2.4 string slicing with negative step and negative index

Specify negative step to slice the string in reverse order.

```
a = "Hello, World!"
print(a[-5:-2:-1]) #
```

8 List

In Python a list is a collection of items in a particular order. You can make a list that includes the letters of the alphabet, the digits from 0-9, or the names of all the people in your family. You can put anything you want into a list, and the items in your list don't have to be related in any particular way. Because a list usually contains more than one element, it's a good idea to make the name of your list plural, such as letters, digits, or names. In Python, square brackets [] indicate a list, and individual elements in the list are separated by commas.

Example

```
bicycles = ["trek", "cannondale", "redline", "specialized"] # list of bicycles,
    each element is a string
print(bicycles)
```

The above example creates a list of bicycles and it elements are all strings. However, you can store any type of data in a list. For example, you can store numbers in a list, and you can even mix different types of data in a list. For example, you can store a string, a number, and a Boolean value in the same list. The following example shows a list that includes several different types of data:

```
mix = ["hello", 1, True]
print(mix) # ['hello', 1, True]
```

8.1 Creating a List

You can create a list by using square brackets [] and separating the elements in the list with commas. The following example shows how to create a list:

```
bicycles = ["trek", "cannondale", "redline", "specialized"]
print(bicycles) # ['trek', 'cannondale', 'redline', 'specialized']
```

You can also create an empty list and add elements to it later. The following example shows how to create an empty list and add elements to it later:

```
bicycles = [] # create an empty list
bicycles.append("trek") # add an element to the list
bicycles.append("cannondale")
bicycles.append("redline")
bicycles.append("specialized")
print(bicycles) # ['trek', 'cannondale', 'redline', 'specialized']
```

8.2 Accessing Elements in a List

Similar to string, you can access elements in a list by using the index of the element. The index of the first element in a list is 0, the index of the second element is 1, and so on. The following example shows how to access elements in a list:

```
bicycles = ["trek", "cannondale", "redline", "specialized"]
print(bicycles[0]) # trek
```

You can also use negative indices to access elements in a list. The index -1 refers to the last item in a list, the index -2 refers to the second last item in a list, and so on. The following example shows how to access elements in a list using negative indices:

```
bicycles = ["trek", "cannondale", "redline", "specialized"]
print(bicycles[-1]) # specialized
```

Slice a List

You can also use a range of indices to access a subset of elements in a list. The following example shows how to access a subset of elements in a list:

```
bicycles = ["trek", "cannondale", "redline", "specialized"]
print(bicycles[0:2]) # ['trek', 'cannondale']
```

8.3 Changing, Adding, and Removing Elements

You can modify elements in a list, add new elements to a list, and remove elements from a list. You can modify elements in a list by using the index of the element you want to change.

Modifying Elements in a List

```
bicycles = ["trek", "cannondale", "redline", "specialized"]
bicycles[0] = "ducati" # change the first element in the list
print(bicycles) # ['ducati', 'cannondale', 'redline', 'specialized']
```

Adding Elements to a List

You can add elements to the end of a list by using the append() method. The following example shows how to add elements to the end of a list:

```
bicycles = ["trek", "cannondale", "redline", "specialized"]
bicycles.append("ducati") # add ducati to the end of the list
print(bicycles) # ['trek', 'cannondale', 'redline', 'specialized', 'ducati']
```

Inserting Elements into a List

You can add a new element at any position in your list by using the insert() method. You do this by specifying the index of the new element and the value of the new item. The following example shows how to insert an element into a list:

```
bicycles = ["trek", "cannondale", "redline", "specialized"]
bicycles.insert(0, "ducati") # insert ducati at the beginning of the list
print(bicycles) # ['ducati', 'trek', 'cannondale', 'redline', 'specialized']
```

Difference of append() and insert()

The difference between append() and insert() is that append() adds a new element to the end of a list, whereas insert() adds a new element at any position in your list. The following example shows the difference between append() and insert():

```
bicycles = ["trek", "cannondale", "redline", "specialized"]
bicycles.append("ducati") # add ducati to the end of the list
bicycles.insert(0, "ducati") # insert ducati at the beginning of the list
print(bicycles) # ['ducati', 'trek', 'cannondale', 'redline', 'specialized', '
    ducati']
```

Removing Elements from a List

You can remove an item according to its position in the list by using the del statement, if you know the position of the item you want to remove. The following example shows how to remove an element from a list:

```
bicycles = ["trek", "cannondale", "redline", "specialized"]
del bicycles[0] # remove the first element in the list
print(bicycles) # ['cannondale', 'redline', 'specialized']
```

Removing an Item Using the pop() Method

You can also remove an item according to its position in the list by using the pop() method. When you use the pop() method, the item you want to remove is no longer stored in the list. The pop() method removes the last item in a list, but it lets you work with that item after removing it. The following example shows how to remove an element from a list using the pop() method:

```
bicycles = ["trek", "cannondale", "redline", "specialized"]
popped_bicycle = bicycles.pop() # remove the last element in the list
print(bicycles) # ['trek', 'cannondale', 'redline']
print(popped_bicycle) # specialized
```

Popping Items from any Position in a List

You can use the pop() method to remove an item in a list at any position by including the index of the item you want to remove in parentheses. The following example shows how to remove an element from a list using the pop() method:

```
bicycles = ["trek", "cannondale", "redline", "specialized"]
first_owned = bicycles.pop(0) # remove the first element in the list
print(bicycles) # ['cannondale', 'redline', 'specialized']
print(first_owned) # trek
```

Removing an Item by Value

You can remove an item according to its value by using the remove() method. The following example shows how to remove an element from a list using the remove() method:

```
bicycles = ["trek", "cannondale", "redline", "specialized"]
bicycles.remove("trek") # remove the element with value "trek"
print(bicycles) # ['cannondale', 'redline', 'specialized']
```

Note that the remove() method deletes only the first occurrence of the value you specify. If there's a possibility the value appears more than once in the list, you'll need to use a loop to make sure all occurrences of the value are removed.

```
bicycles = ["trek", "cannondale", "redline", "specialized", "trek"]
while "trek" in bicycles: # remove all occurrences of "trek"
    bicycles.remove("trek")
print(bicycles) # ['cannondale', 'redline', 'specialized']
```

8.4 Organizing a List

You can organize a list in alphabetical order, reverse alphabetical order, or in the order you added items to the list. You can also reverse the order of a list permanently.

Sorting a List Permanently with the sort() Method

The sort() method changes the order of a list permanently. If the list is a number list, the sort() method will sort the list in ascending order. If the list is a string list, the sort() method will sort the list in alphabetical order. The following example shows how to sort a number list and a string list using the sort() method:

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
numbers.sort() # sort the list in ascending order
print(numbers) # [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]
cars = ["bmw", "audi", "toyota", "subaru"]
cars.sort() # sort the list in alphabetical order
print(cars) # ['audi', 'bmw', 'subaru', 'toyota']
```

Sorting a List in Reverse Alphabetical Order

You can also sort a list in reverse alphabetical order by passing the argument reverse=True to the sort() method. The following example shows how to sort a list in reverse alphabetical order using the sort() method:

```
cars = ["bmw", "audi", "toyota", "subaru"]
cars.sort(reverse=True) # sort the list in reverse alphabetical order
print(cars) # ['toyota', 'subaru', 'bmw', 'audi']
```

Sorting a List Temporarily with the sorted() Function

The sorted() function lets you display your list in a particular order but doesn't affect the actual order of the list. The following example shows how to sort a list temporarily using the sorted() function:

```
cars = ["bmw", "audi", "toyota", "subaru"]
print("Here is the original list:")
print(cars) # ['bmw', 'audi', 'toyota', 'subaru']
print("\nHere is the sorted list:")
print(sorted(cars)) # ['audi', 'bmw', 'subaru', 'toyota']
print("\nHere is the original list again:")
print(cars) # ['bmw', 'audi', 'toyota', 'subaru']
```

Printing a List in Reverse Order

You can reverse the original order of a list permanently by using the reverse() method. The following example shows how to reverse a list permanently using the reverse() method:

```
cars = ["bmw", "audi", "toyota", "subaru"]
cars.reverse() # reverse the list permanently
print(cars) # ['subaru', 'toyota', 'audi', 'bmw']
```

Finding the Length of a List

Same as strings, you can find the length of a list by using the len() function. The following example shows how to find the length of a list using the len() function:

```
cars = ["bmw", "audi", "toyota", "subaru"]
print(len(cars)) # 4
```

8.5 Avoiding Index Errors When Working with Lists

If you try to access an item in a list that doesn't exist, you'll get an index error. For example, if you try to access the third item in a list that has only two items, you'll get an index error. The following example shows how to avoid index errors when working with lists:

```
motorcycles = ["honda", "yamaha", "suzuki"]
print(motorcycles[3]) # IndexError: list index out of range
```

8.6 Looping Through an Entire List

You can loop through the entire list by using a for loop. The following example shows how to loop through an entire list using a for loop:

```
magicians = ["alice", "david", "carolina"]
for magician in magicians: # The in keyword tells Python to pull a value from
    the list magicians and store it in the variable magician.
    print(magician)
```

9 Functions

In Python a function is a block of code that is executed when it is called. A function can take parameters and return a value. A function is defined using the def keyword. The syntax of a function is as follows:

```
def function_name(parameters):
    """docstring"""
    statement(s)
    return [expression]
```

9.1 Why use functions?

Functions are used to break down a large program into small manageable and organized chunks. This makes program easy to understand, maintain and debug. Furthermore, it avoids repetition and makes the code reusable. Imagine you have to write a program that calculates the area of a rectangle. You can write a function that calculates the area of a rectangle and use it whenever you want to calculate the area of a rectangle. This saves you from writing the same code again and again. You can also use the same function to calculate the area of a square.

Example

Let us write a function that calculates the area of a rectangle. The function takes two parameters, length and width, and returns the area of the rectangle.

```
def area(length, width):
    """This function calculates the area of a rectangle"""
    return length * width

print(area(5, 6)) # 30
print(area(2, 3)) # 6
```

The function definition starts with the def keyword. It is followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

9.2 Parameters

Information can be passed to functions as parameters. Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

```
def greet(name, msg):
    """This function greets to
    the person with the provided message."""
    print("Hello", name + ', ' + msg)

greet("Monica", "Good morning!") # Hello Monica, Good morning!
```

9.3 Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The default value is specified after the parameter name in the function definition. The default value is evaluated at the point of function definition in the defining scope.

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

greet("Monica", "Good morning!") # Hello Monica, Good morning!
greet("John") # Hello John, Good morning!
```

9.4 Return values

A function can return a value. The return statement is used to exit a function and go back to the place from where it was called. The statement consists of the return keyword followed by an expression that is evaluated and returned. If the return statement is without any expression, then the special value None is returned.

```
def greet(name, msg):
    """This function greets to
    the person with the provided message."""
    print("Hello", name + ', ' + msg)
    return

greet("Monica", "Good morning!") # Hello Monica, Good morning!
```

9.5 Scope of variables

The scope of a variable is the portion of the program where the variable is recognized. There are two basic scopes of variables in Python.

- Global variables
- Local variables

9.5.1 Global variables

A variable that is defined outside of a function or in global scope is a global variable. Global variables can be accessed inside or outside of the function.

```
x = "global"

def foo():
    print("x inside:", x) # x inside: global

foo()
print("x outside:", x) # x outside: global
```

9.5.2 Local variables

A variable that is defined inside a function's body or in the local scope is a local variable. Local variables can only be accessed inside the function.

```
x = "global" # global variable

def foo():
    x = "local" # local variable
    print("x inside:", x) # x inside: local

foo()
print("x outside:", x) # x outside: global
```

9.6 Recursion

A function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions. In a recursive function, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

```
def factorial(x):
    if x == 1: # base case
        return 1
    else: # recursive case
        return x * factorial(x-1)

print(factorial(5)) # 120
```