
# MASTER OF COMPUTER APPLICATION

## SEMESTER 1

# PYTHON PROGRAMMING

# Unit 2

# Python Syntax and Variables

## Table of Contents

## 1. INTRODUCTION

Welcome to Unit 2 of our Python journey, where we delve deeper into the realm of Python syntax and variables, building upon the foundational knowledge you've acquired in Unit 1. In the introductory unit, you've familiarized yourself with Python's welcoming syntax, its object-oriented nature, and how Python treats everything as objects. You've explored the three main object characteristics: identity, type, and value, laying the groundwork for more sophisticated programming concepts.

Unit 2 is all about Python variables, identifiers, and the crucial process of declaring variables and assigning values. Variables are the basic storage units in programming, holding the data we manipulate to achieve desired outcomes. Understanding how to effectively name, declare, and assign values to variables is pivotal for writing clear and efficient code. Moreover, we'll unravel the intricacies of Python data types, offering you the tools to handle various kinds of data adeptly. From simple integers and strings to complex data structures like lists and dictionaries, this unit covers the gamut, ensuring you're well-equipped to store and process data in your Python endeavors.

To maximize your learning in this unit, engage with the material actively. Experiment with different variable names and data types in your code editor, observe Python's behavior with various assignments, and challenge yourself with exercises that test your understanding of Python's syntax and variables. Apply the "Analyze" action verb from Bloom's taxonomy by dissecting the examples provided, "Evaluate" the best practices for naming identifiers, and "Create" your own variables and data structures to solidify your grasp of these fundamental concepts. Remember, the best way to learn programming is by doing, so don't hesitate to get your hands dirty with code.

## 1.1. Learning Objectives

*At the end of this unit, you will be able to:*

- *Identify and apply the correct syntax for declaring and initializing variables in Python to store various types of data.*
- *Differentiate between valid and invalid identifiers based on Python's naming conventions and rules.*
- *Demonstrate the ability to assign values to variables, including performing multiple assignments and understanding the scope of variables (local vs. global).*
- *Construct simple Python scripts that utilize different data types and variable assignments to solve specific problems.*

## 2. PYTHON VARIABLES

Different programming languages have different mechanisms for storing or calling any data. You can retrieve data directly or indirectly from the computer storage. Languages such as C and C++ allow the user to access the memory directly. Whereas, there are some programming languages such as Java, Visual Basic, and more that follow such a mechanism that you can only retrieve data through indirect methods.

All data that is stored through a programming language needs a variable. Variable is a reserved memory location. It is created when the user assigns a value to it.

As mentioned in the previous unit as well, Python is an object-oriented language. That means Python uses objects to store and designate all types of data stored in the memory. What is an object? It can be defined as an entity of the memory that contains any value or data. In Python, data is defined either as an object or through its relation with an object.

An object has three characteristics. These are as explained below.

- Identity: Identity can be called the address of the memory where the object is to be stored. Each identity is unique so that all objects are defined differently.
- Type: Type refers to the operations that can be performed on the data that is stored in the object. In Python, the built-in function type () is used to define the type of the object.
- Value: The data contained in the object is called its value.

In Python, a variable is defined as the object that can store only one value and have a single attribute. The interpreter allocates memory to a variable depending upon its data type.

## 3. IDENTIFIERS RULES

Identifiers are names defined by the user to denote a variable. In the below example, we represented the variables by the names 'a' and 'b'. Thus, these are identifiers. Identifiers are the basic blocks of a Python program. We can use one identifier to represent one single variable only in a particular program.

Example:

num = 20

### 3.1. Rules to Follow While Naming Identifiers

You cannot use any word as an identifier. Because of the presence of keywords in Python, there are certain rules that you will have to follow while naming identifiers. These are explained below.

- An identifier can have lowercase letters, uppercase letters, digits, and or an underscore. Thus, identifiers such as Book65_a, variable_1, this_is_a_variable, are acceptable.
- They are case sensitive, if it is written as "identify", it has to be recalled as "identify" again; it will not accept if it is recalled as "Identify" it will take it as a new variable and throw an error.
- However, identifiers cannot begin with a digit. Thus, you cannot use 1_var or 8variable.
- Special symbols, including !, @, #, $, % cannot be used in an identifier.
- You cannot use a keyword as an identifier. We will study keywords in Python later in the unit.
- There is no particular limit to the length of the identifier. You can make it as long as you want but for sake of ease, keep it comprehensible.

### 3.2. Convention to be Followed While Naming Identifiers

These are not rules that you must follow while naming identifiers in Python. However, these conventions are followed so that a program can be understood by other users as well.

- Class names begin with an uppercase letter whereas all identifiers begin with a lowercase letter.

- Private identifiers should begin with an underscore (_).
- Using identifiers longer than one character is advised.
- One should keep in mind that Python is case-sensitive. Thus, variable and Variable will be considered differently.

## 4. DECLARING VARIABLE AND ASSIGNING VALUE

Now that you have understood the rules that need to be followed while naming an identifier, here are the ways that you can declare a variable in Python.

Number Variable: Variables that are used to save number values are numeric variables. Python supports three types of numbers integers, floating-point numbers, and complex numbers. Take the example provided below to understand how these variables are declared.

num = 20

float_num = 26.90

a = 3 + 4j

String Variable: Strings are a sequence of characters. These are declared under single or double-quotes. Some examples are given below.

str_1 = 'Example1'

str_2 = "Example2"

In Python, you do not have to declare the variable before you can assign a value to it. Once you assign the value, automatically the memory is reserved and the data is stored in the location. To assign a value to a variable, the equal sign OR Assignment Operator (=) is used.

For example:

>>> a = 10

>>> b = 20

>>> b-a

Output

10

Here, the operand to the left side of the equal sign is called variable. The operand on the right side is the data stored in that variable.

Along with numeric values, such as integers, decimals, etc., you can also assign string values to a variable. The string is a sequence of characters whose sequence cannot be altered. They are always written inside single inverted commas.

For example,

>>> a = 'best'

>>> b = 'friend'

>>>a + b

Output

'bestfriend'

## 4.1. Multiple Assignment

You can assign one value to several variables at the same time. For example,

a = b = c = 1

You can also assign different values to different variables in the same command. For example,

a, b, c = 1, 2, 3

In this example, the interpreter will assign the value as per the sequence in which they are defined. Thus, a will be assigned with value 1, b with 2, and c with 3.

You can use the method to swap the values of two variables as well. For example,

var_1, var_2 = 3, 8

var_1, var2 = var_2, var_1

When you get the output of the program, var_1 will have the value 8 and var_2 will have 3.

## 4.2. Local Variables

Local variables are defined inside a function. A function is a set of commands that are arranged in a different block than the whole program.

The local variables can only be used inside the function. They have no meaning when called outside of it.

For example,

```
def f ():
    print (a)
    a = 44
a = 33
f ()
print (a)
```

The above program will fail to show output and give an error instead. This is because the variable a is local to the function f and cannot be declared or defined outside of it. Here is an example that will work:

```
def f ():
    a = 'Python'
    print (a)
f ()
```

## 4.3. Global Variables

Global variables are defined and declared outside a function. You can use and call them anywhere in the program, even inside a function.

For example,

```
def f ():
     print (a)
a = 64
f ()
```

Here, the output will be 64. As you can see, the variable 'a' is defined and declared outside the function and then called inside it. Thus, a is a global variable.

## 4.4. Delete A Variable

In a huge program where lots of variables are used, it is good practice to delete those which have served their purpose. This helps in maintaining the memory and free up space in RAM that can be used to perform and handle other tasks. To delete a variable, the keyword del is used in Python. the del keyword is used to delete an object in Python and since variables are objects as well, the keyword can be used in this case as well.

For example,

```
a = “hello”
del a
print (a)
```

In the output will, the interpreter will give the error that ‘a’ is not defined.

## 5. PYTHON DATATYPES

The data that is stored in the memory can be categorised into various types. Python's interpreter assigns space in the memory as per the requirement for the type. Each such type is called a data type. These data types can perform different operations.

The categorisation of data types is necessary. For example, when defining someone's name, you will need a string of characters whereas, for a student's roll number, you will need a numeric value.

Python has some data types built-in by default. These standard data types are also called primitive data types. The categories of data types in Python are as follows.

| Category | Data Type |
|---|---|
| **Text Type** | str (string) |
| **Numeric Type** | int (integers), float (decimal), complex (complex numbers) |
| **Sequence Type** | list, tuples, range |
| **Mapping Type** | dict (dictionary) |
| **Set Types** | set, frozenset |
| **Boolean Type** | bool |
| **Binary Type** | bytes, bytearray, memoryview |

### Strings

Strings are text type data types. These are a series of characters that are saved in the memory in the sequence that they are provided. They are enclosed in either single or double-quotes. However, while using strings, you should keep in mind that the quote that you have used to enclose the string cannot be used inside the string. For example, "won't" is valid. However, 'won't' will give you a syntax error. Here are some examples of string.

str = "Hello"

str_1 = 'World'

str_2 = "Goodbye!"

In Python, strings are immutable. Hence, once declared and assigned a value, they cannot be changed. When you assign a new value to it, another object is created.

When you need to save long sentences or paragraphs using string and need to divide the lines, you will have to use '\n' wherever you need a line break.

For example,

str = "I will learn Python. \n I want to learn to code"

When you print the string, it will look like this:

Output

I will learn Python.

I want to learn to code.

Another thing to keep in mind is the use of triple quotes, such as " " " or ' ' '. When using triple quotes, the new line will not escape the string and will be considered a part of it.

For example:

str = " " "I will learn to code with Python.

It is an easy programming language.

It is used to develop various web-based applications. " " "

The output of this string will be as follows:

Output

I will learn to code with Python.

It is an easy programming language.

It is used to develop various web-based applications.

To find the length of the string, that is the number of characters used in the string, you can use the function len (). For example,

str = “play with me”

len (str)

The output will be 12.

Another thing to note is that Python, unlike other programming languages, does not support character type. There is no way to store a single character other than using string. A single character can be extracted from a string using slicing. Slicing is a method through which the index of a character in a string is used to extract it from the complete string. Here is an example to make you understand better.

str = ‘playing’

print (str [0])

The output will be: p

Similarly, you can slice a substring as well. For this, you will have to use a colon in a way that is shown in the example.

print (str [0:3])

The output, in this case, will be: pla

If you do not specify the end of the index, then the range will be up to the end of the string.

For example,

print (str [3:])

This will give an output of **ying**

If you want to slice the string from the end, then you will have to use negative indexing. Note that the negative indexing, that is, from right to left, starts from -1.

For example,

print (str [-4:-2])

The output will be: yi

## 5.1. Numbers

Number data types are used to store numeric values. These are categorised as an immutable data type. An immutable data type is one in which changing the value of the data type results in a newly allocated object. You can create a number data type simply by assigning a value to it. For example,

var = 8

You can change the value stored in var by reassigning another value. For example,

var = 8

var_1 = 6

var = var_1

The output of the program, when you print var will give the value 6.

In Python, the number data type is further divided into four types. These are explained below.

**Integers**

These are the most commonly used data types. As defined in mathematics, integers are numbers ranging from negative to positive, including zero. However, due to constrictions of memory, in Python, the integers have a range from -2,147,483,648 (-2^31) through 2,147,483,647 (2^31 - 1).

Integers are represented in decimal format, that is, with base 10. However, one can also define them using the octal and hexadecimal format. If you are using integers in octal or hexadecimal format, then ensure that you use the prefix 0 and 0x respectively. To specify that the data type that you are using is an integer, you can use the constructor function provided in the example below.

For example:

a = int (60)

**Long Integers**

Long integers are a subtype of integers. These are when you need a number that is greater than 231 or lesser than -231. Long integers in Python do not have a range and are limited as per the virtual memory of your computer system. The memory should be large enough to accommodate and save a long integer. While defining a long integer, you should use the suffix 'l' or 'L'. These can also be defined in the form of decimal, octal, and/or hexadecimal.

For example, 100, -856, 0X35 are examples of integers.

Whereas 826353738L, 0X52462354L and -6253902628L are the examples of long integers.

Study Note: To represent long integers, you can use both 'l' and 'L' but the uppercase 'L' is preferred to avoid confusion of lowercase 'l' with '1' (number 1).

**Floating Point Number**

Floating point numbers or as commonly known as, float, are numbers with a decimal point. They have two parts; one is the decimal point part and the other is the exponent part. The latter is optional and it is mandatory to define it while using a float. They are assigned 8 bytes of memory. The 52 bits are taken up by mantissa, 11 by the exponent, and the one remixing bit is for the negative or positive sign.

The exponent part is denoted by an uppercase or lowercase 'e' or 'E'. The sign mentioned just before 'e' is the sign of the exponent. The absence of mention of sign, in any case, will mean that it is positive.

Some examples of floating-point real numbers are 9.0, 55.8977, -658.99, 52.66-E37, -88.223e6, etc.

To explicitly define that a variable stores a floating data type, you can use the function float ().

**Complex Numbers**

Complex numbers in Python are represented by two floating numbers taken in an order of a + bj. Here, a represents the real part and b is the imaginary part of the complex number. The imaginary part is always followed by a lowercase 'j' or uppercase 'J'. (Usually, 'j' in lowercase is used). Some examples of complex numbers are 7 + 4j, -8.9 +5j, -876j, 2e4j, etc.

In some cases, if you want to print or extract only the imaginary or real part of the complex number, then you can do so by using its data attributes. This has been shown through an example provided below. A variable can be specified to store a complex number with the help of the function complex (ij)

```
complex = 44.9 + 89e4j

print (complex.real)

print (complex.imag)
```

The output will be like this:

```
44.9

89e4
```

You can also get the conjugate of the complex number. If a complex number is defined as a + ib, then its conjugate will be a – ib. Using the attribute complex.conjugate will print the conjugate of the complex number in Python.

## 5.2. Sequence Type

The sequence data type is another essential data type that is commonly used in Python to store a series of data together. These are further divided into the following types.

**Lists:**

Lists work similarly as arrays in C. The one difference is that in lists, you can store values of different data types together. The items in a list are enclosed inside a square bracket []. Each value is separated with a comma. The values saved in a list are indexed and ordered. Thus, when you print a list, the order of the values stored in it does not change.

A list is alterable. You can add items to the list, remove them, delete the list, etc., once it has been created. You can also store the same values in a list. As they have a different index, the user will be able to differ between the same data.

For example,

list_1 = [23, 56, 766, “passed”]

As mentioned, the indexing in the list starts from 0 as well. Thus, in the above example, element 23 has an index of 0, 56 has 1, and so on.

Here are the methods that you can implement in the list. These are built-in and can be used using the functions that are listed in the table below.

**TABLE 1: METHODS FOR LISTS**

| Method | Use |
|---|---|
| **insert ()** | To add an element to the list at a specified position or index |
| **extend ()** | To add the elements of a list to another list |
| **count ()** | To count the number of elements, present in the list. |
| **append ()** | To add an element at the end of the list |
| **clear ()** | To remove all the elements of the list. |
| **pop ()** | To remove an element from the list at a specific index. |
| **reverse ()** | To reverse the order of the list |
| **sort ()** | To sort the list |

## Tuples

Tuple functions are similar to lists but with one difference. They are immutable. Values stored in a list can be changed and altered even after the list has been created. Such a feature may become an issue in some programs. Thus, in such cases, tuples can be used. Tuples are enclosed in parentheses with each element separated by a comma. They are ordered as well and allow repeating a single value.

For example,

tuple1 = (3, 6, 9)

To join two tuples together, you can use the + operator. It is shown in the example below.

tuple1 = (3, 6, 9)

tuple2 = (“a”, “b”, “c”)

tuple3 = tuple1 + tuple2

The output will be (3, 6, 9, “a”, “b”, “c”)

## 5.3. BOOLEAN

Boolean type is used to store one of the two values: True or False. Bool is used to test whether the result of an expression is true or false. This can be used when you are comparing two values. If you run such a conditional statement as shown in the example, Python will give True or False as a result.

For example,

print (7 == 8)

This statement will result in an output showing False.

The bool () function is used to verify whether a value is true or false. Here, being a true value means that it is acceptable by Python.

For example,

print (bool (89))

This will return the value as True in the output window.

## 5.4. Set

Just like lists and tuples, sets are used to store a collection of data. Through set, you can store multiple items in a single variable. The collection is unordered and unindexed. That is, the

data stored in a set is not saved in a sequence. Hence, you cannot surely say in which order the sets will appear or get printed. They are immutable as well. That means, once a collection of data is stored in a set, it cannot be changed. Also, no two sets can have the same value.

You cannot completely change the values stored in a set but you can add new data. And, example of a set is provided below.

set_1 = {“pencil”, “eraser”, “ruler”}

Here, note that all the data in the set is enclosed in curly braces {}.

To add new items to a set, you can use the function add (). The method is shown in the example below.

set_1 = {“pencil”, “eraser”, “ruler”}

set_1.add (“pen”)

When you print the set, you will find four values in it, namely, pencil, eraser, ruler, and pen arranged in random order.

You can add the values of one set to another using the function update (). For example,

set_1 = {“pencil”, “eraser”, “ruler”}

set_2 = {“protractor”, “divider”}

set_1.update (set_2)

The output will be: pencil, eraser, ruler, protractor, and divider.

The update () function can be used to add any iterable object such as lists, tuples, etc.

The function remove () or discard () can be used to remove a value from the set. For example,

set_1 = {“pencil”, “eraser”, “ruler”}

set_1.remove (“ruler”)

This way, you can remove the value “ruler” from the set.

To empty the complete set, you can use the method clear ().

For example,

set_1.clear ()

will remove all the values from the set.

The del keyword will delete the set completely.

For example,

del set_1

## 5.5. Dictionary

Imagine you want to look up a number in the logs of your mobile phone. You do not remember the number, and surely you would not remember the index at which it is saved. How will you search for it then? Using the name of the person whose number you are searching for. The name of the person is called key whereas the phone number will be the value.

In Python, you can save the pair of key: value together in the form of dictionary data type. In a dictionary, one key is associated with only one value. It is enclosed in curly braces {} and each pair of key: value is separated by a comma.

For example:

```
dict_1 = {
“name”: “Aakash”,
“birthyear”: 1998,
“rollnumber”: 891
}
```

The values in a dictionary are stored in order. No duplication of the value of a key is allowed in a dictionary.

Here are the functions and methods that you can use to create, modify, and add to a dictionary.

| TABLE 2: METHODS FOR DICTIONARY | |
|---|---|
| **Method** | Use |
| **clear ()** | To remove all the elements of a dictionary |
| **copy ()** | To return a copy of the dictionary |
| **get ()** | To return the value of a specific key |
| **keys ()** | To get a list of the keys included in a dictionary |
| **update ()** | To update the dictionary by adding key:value pairs |
| **values ()** | To get a list of values present in the dictionary |
| **pop ()** | To remove the element of the specified key |

## 6. PYTHON KEYWORDS

Every programming language has a definite syntax and rules that need to be followed while writing a program. Python reserves some of the names that cannot be used as programmer-defined identifiers. These predefined identifiers are called keywords. Here is a list of keywords used in Python.

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

- try
- print
- yield

Till now, we have learnt to use global (to define global variables) and print (to print a variable or value on the output screen).

# 7. TYPE CONVERSION

## 7.1. Introduction

### Understanding Type Conversion

Type conversion, often referred to as type casting, is a pivotal concept in Python programming, enabling developers to manipulate variables across different data types more effectively. This process allows for greater flexibility in arithmetic operations, function arguments, and data storage, ensuring compatibility and preventing runtime errors due to type mismatches.

### The Significance of Type Conversion

In Python, a dynamically typed language, the type of a variable is determined at runtime and can change based on the value assigned to it. While this offers a high degree of flexibility, it also introduces scenarios where explicit type conversion becomes necessary. For instance, when performing operations between integers and floating-point numbers, Python needs to reconcile these types to execute the operation successfully. Similarly, when receiving input from a user or a file, the data often comes as a string, necessitating conversion to the appropriate type for numerical operations or other processing tasks.

### Data Types in Python

Python's rich set of built-in data types forms the backbone of its type system, with each type serving distinct purposes:

- Integers (int): Used to represent whole numbers without a fractional component. Ideal for countable items, indexing, and situations requiring discrete values.
- Floating-point Numbers (float): Represent real numbers and include a decimal point. Suitable for measurements, scientific calculations, and any context requiring fractional values.
- Strings (str): A sequence of characters used to store textual data. Strings are immutable and can include letters, digits, symbols, and whitespace.

- Lists (list): Ordered, mutable collections that can hold items of mixed types, including other lists. Lists are versatile and widely used for data storage and manipulation.
- Tuples (tuple): Ordered, immutable collections similar to lists but cannot be altered once created. Tuples are typically used for fixed collections of items.
- Dictionaries (dict): Unordered collections of key-value pairs. Dictionaries are mutable and indexed by unique keys, making them ideal for associative data storage.

**Implicit vs. Explicit Conversion**

**Python supports two main types of type conversion:**

- Implicit Conversion: Automatically performed by Python when an operation involves mixed data types. For example, adding an integer to a floating-point number automatically converts the integer to a float to preserve the decimal component.
- Explicit Conversion: Manually executed by the programmer using built-in functions like int(), float(), and str(). This form of conversion is necessary when the desired type conversion cannot be inferred by Python or when precise control over the conversion process is required.

**Real-world Application**

Consider a scenario where you're developing a web application that accepts user input for numerical data, such as age or price. The input received from a web form is typically in string format. To perform any numerical calculations or store this data in a numerical field in a database, explicit type conversion from string to integer or float is essential. This conversion ensures data integrity and prevents type-related errors in your application.

Type conversion is a foundational aspect of Python programming, bridging the gap between different data types and enabling seamless data manipulation and processing. Whether implicitly handled by Python or explicitly executed by the programmer, understanding type conversion is crucial for writing robust and error-free code.

## 7.2. Implicit Type Conversion

**Understanding Implicit Type Conversion**

Implicit type conversion, also known as type coercion, is a process where Python automatically converts one data type to another without any explicit instruction from the programmer. This feature is designed to prevent data loss and to ensure that operations are performed on compatible data types, thereby maintaining the integrity and predictability of arithmetic operations and expressions.

**Mechanism of Implicit Conversion**

The Python interpreter follows a set of predefined rules to determine how and when to apply implicit type conversion. These rules are based on a type hierarchy or "type promotion" in which certain data types are automatically converted to others to avoid losing information. For example, in an operation involving both integers (int) and floating-point numbers (float), the integer is seamlessly converted to a float, preserving the decimal component of the calculation.

**Common Scenarios for Implicit Conversion**

1. Arithmetic Operations: When performing arithmetic operations with mixed data types, Python converts the operands to a common type to execute the operation successfully.

   ```
   # Integer and float mixed in an operation

   result = 10 + 2.5  # 10 (int) is implicitly converted to 10.0 (float)
   ```

2. Boolean in Arithmetic: Booleans (bool) are treated as integers (int) in arithmetic operations, with True being 1 and False being 0.

   ```
   # Using boolean in arithmetic

   total = 100 + True  # True is implicitly converted to 1
   ```

3. Function Arguments: When passing arguments to functions that expect a specific type, implicit conversion may occur if it does not lead to data loss or ambiguity.

**Examples and Use Cases**

1. Combining strings with numbers in a print statement:

```
age = 30  # int

print("You are " + str(age) + " years old.")  # Without explicit conversion, this would raise an error
```

In the above example, although implicit conversion doesn't directly apply, it demonstrates a scenario where understanding conversion is crucial. Python does not implicitly convert integers to strings in concatenations, necessitating explicit conversion.

2. Calculating the average of a list containing both integers and floats:

```
numbers = [1, 2.5, 3, 4.5]  # mix of int and float

average = sum(numbers) / len(numbers)  # int values are implicitly converted to floats during the sum operation
```

**Limitations and Considerations**

While implicit type conversion adds convenience and flexibility to Python programming, it also introduces areas where careful consideration is required:

- Precision Loss: In some cases, implicit conversion, especially from float to int, can lead to precision loss. Python, however, does not implicitly convert floats to integers to avoid this issue.
- Type Errors: Not all data types can be implicitly converted. For example, attempting to concatenate a string and an integer directly will result in a TypeError.

Implicit type conversion in Python plays a vital role in facilitating operations across different data types, enhancing the language's flexibility. By adhering to a set of predefined rules, Python ensures that data integrity is maintained during these conversions. However, programmers must be vigilant about where and how these conversions occur to avoid unintended consequences in their code.

## 7.3. Explicit Type Conversion

The Essence of Explicit Conversion

Explicit type conversion, also known as type casting, is a pivotal aspect of Python programming where the programmer intentionally converts data from one type to another. This process is essential when Python's implicit conversion isn't applicable or when precise control over data types is necessary for specific operations.

**Key Functions for Explicit Conversion**

Python provides several built-in functions to facilitate explicit type conversion, catering to various data types and scenarios:

- int(x[, base]): Converts x to an integer. The base specifies the numeral system for conversion, defaulting to 10.

  integer = int("123") # Converts string to integer

- float(x): Converts x to a floating-point number.

  floating_point = float("123.456") # Converts string to float

- str(x): Converts x to a string.

  string = str(123) # Converts integer to string

- bool(x): Converts x to a Boolean (True or False), following the truthiness rules in Python.

  boolean = bool(1) # Non-zero numbers are converted to True

**Conversion Between Collection Types**

- list(x): Converts x to a list, where x can be any iterable like tuples, sets, or dictionaries.

  list_from_tuple = list((1, 2, 3)) # Converts tuple to list

- set(x): Converts x to a set, useful for removing duplicates from a sequence.

  set_from_list = set([1, 2, 2, 3]) # Converts list to set, removing duplicates

- dict(x): Converts x to a dictionary. x must be an iterable of key-value pairs.

```
dict_from_list = dict([(1, 'one'), (2, 'two')])  # Converts list of tuples to dictionary
```

**Practical Applications and Considerations**

Explicit type conversion is commonly used in scenarios where data types need to be standardized for processing, such as user input handling, data serialization, or when interfacing with databases. It allows for the conversion of data received in one format (e.g., from user input or a file) into a more suitable format for computation or storage.

- User Input: All user input received from functions like input() is treated as a string and often needs to be explicitly converted for numerical operations.

  ```
  age = int(input("Enter your age: "))  # Convert user input to integer
  ```

- Data Serialization: When reading or writing data to files, especially in formats like JSON or CSV, explicit conversion ensures that data types are preserved accurately.

**Best Practices and Error Handling**

While explicit type conversion offers flexibility, it also requires careful error handling to manage situations where conversion might fail, typically due to incompatible data formats.

- Try-Except Blocks: Use try-except blocks to catch and handle exceptions that arise from failed type conversions, providing a graceful fallback or user feedback.

  ```
  try:
      number = int(input("Enter a number: "))
  except ValueError:
      print("That's not a valid number!")
  ```

- Data Validation: Before attempting a conversion, especially with user-provided data, validate the data to ensure it meets the expected format or criteria.

  ```
  user_input = input("Enter a positive number: ")
  if user_input.isdigit():
      number = int(user_input)
  else:
      print("Invalid input!")
  ```

Explicit type conversion in Python empowers programmers to precisely control the data types of their variables, ensuring compatibility and correctness in operations that involve disparate types. Through the judicious use of built-in conversion functions and mindful error handling, developers can navigate Python's dynamic typing system effectively, making their code more robust and adaptable to varying data inputs.

## 7.4. Advanced Topics in Python Type Conversion

Expanding on the foundational knowledge of explicit type conversion, let's explore some advanced topics that highlight the versatility and power of type conversion in Python. These include handling complex data structures, working with binary data, and leveraging type hints for improved code clarity and type checking.

**Handling Complex Data Structures**

Complex data structures, such as nested lists, dictionaries, or combinations thereof, often require careful consideration during type conversion, especially when preparing data for serialization or communication between different parts of an application or with external systems.

Nested Structures: When dealing with nested lists or dictionaries, recursive functions can be used to convert all elements to a desired type, ensuring uniformity across the data structure.

```
def convert_all_items_to_str(data):
    if isinstance(data, list):
        return [convert_all_items_to_str(item) for item in data]
    elif isinstance(data, dict):
        return {convert_all_items_to_str(key): convert_all_items_to_str(value) for key, value in data.items()}
    else:
        return str(data)

# Example usage
nested_list = [1, [2, 3], 4]
converted_list = convert_all_items_to_str(nested_list)
```

Working with Binary Data

Binary data manipulation is another area where explicit type conversion is crucial, particularly when dealing with files, network communication, or low-level data processing.

Bytes and Bytearray: Python provides bytes and bytearray types for handling binary data. Converting between these types and strings or integers involves explicit conversion, often with consideration for the encoding used.

```
# Converting a string to bytes
my_string = "Hello, World!"
my_bytes = my_string.encode('utf-8')

# Converting bytes to a string
converted_string = my_bytes.decode('utf-8')
```

**Leveraging Type Hints and Static Type Checking**

Type hints, introduced in Python 3.5 through PEP 484, allow developers to annotate variables, function parameters, and return types with explicit types. While these annotations do not enforce type conversion at runtime, they enable static type checking, improving code clarity and catching potential type-related errors during development.

- Function Annotations: Adding type hints to functions clarifies the expected argument and return types, enhancing readability and maintainability.

  ```
  def greet(name: str) -> str:
      return "Hello, " + name
  ```

- Static Type Checkers: Tools like mypy can be used to perform static type checking based on the provided type hints, identifying type inconsistencies without executing the code.

  ```
  mypy script.py
  ```

Best Practices for Advanced Type Conversion

- Consistency and Clarity: Ensure that type conversions and annotations are used consistently throughout the codebase to maintain clarity and prevent confusion.

- Error Handling: Advanced type conversions, especially when dealing with external data sources or complex structures, should include comprehensive error handling to gracefully manage conversion failures.
- Performance Considerations: Be mindful of the performance implications of type conversions in critical code paths, particularly in data-intensive applications. Profiling and optimization may be necessary to maintain efficiency.

## 7.5. Best Practices in Type Conversion

Adhering to best practices in type conversion is crucial for writing clear, efficient, and error-resistant Python code. Below are key guidelines to follow when performing type conversions, ensuring your codebase remains robust and maintainable.

1. **Explicit is Better Than Implicit**

   In line with the Zen of Python, it's preferable to make type conversions explicit rather than relying on Python's implicit coercion. This approach enhances code readability and reduces the likelihood of unexpected behaviors or bugs.

```
# Explicit conversion of string to integer
age = "30"
age_int = int(age)  # Explicitly converting string to int
```

2. **Validate Data Before Conversion**

   Always validate data before attempting a type conversion, especially when dealing with user input or data from external sources. This step helps prevent runtime errors like ValueError and ensures that the data conforms to the expected format.

```
user_input = input("Enter a number: ")
if user_input.isdigit():
    number = int(user_input)
else:
    print("Invalid input. Please enter a numeric value.")
```

3. **Use Try-Except Blocks for Error Handling**

When performing explicit type conversions, wrap the conversion in a try-except block to gracefully handle potential conversion errors. This practice is particularly important for conversions that might fail, such as converting strings to integers or floats.

```
try:
    number = int(user_input)
except ValueError:
    print("Conversion error: Input is not a valid integer.")
```

4. **Prefer str.format() or f-Strings for String Interpolation**

When incorporating variables into strings, use str.format() or f-strings for clarity and flexibility. These methods are more readable and powerful compared to old-style % formatting.

```
# Using str.format()
greeting = "Hello, {}. Welcome to Python programming!".format(name)

# Using f-strings (Python 3.6+)
greeting = f"Hello, {name}. Welcome to Python programming!"
```

5. **Be Mindful of Numeric Type Conversion**

When converting between numeric types (int to float, float to int), be aware of the implications, such as the loss of precision or changes in value. Use rounding functions if necessary to achieve the desired results.

```
# Converting float to int (loss of decimal part)
floating_point = 3.14
integer = int(floating_point)  # integer is 3
# Rounding before conversion for more precise control
rounded_integer = int(round(floating_point))  # rounded_integer is 3
```

6. **Handle Binary Data with Care**

When dealing with binary data (bytes and bytearray), ensure that you understand the encoding and decoding processes. Incorrect handling can lead to data corruption or loss.

```
# Correctly converting string to bytes and back
original_string = "Hello, world!"
bytes_data = original_string.encode('utf-8')  # Encoding string to bytes
decoded_string = bytes_data.decode('utf-8')  # Decoding bytes back to string
```

7. **Leverage Python's Standard Library**

   Python's standard library offers numerous modules (json, pickle, struct, etc.) for converting between Python data types and formats suitable for storage or transmission. Use these modules to handle common data serialization and deserialization tasks.

```
import json
# Converting a dictionary to a JSON string
data_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
json_string = json.dumps(data_dict)
# Converting JSON string back to dictionary
decoded_dict = json.loads(json_string)
```

8. **Consistently Use Type Hints for Clarity**

   Incorporate type hints in your function definitions to clarify the expected types of arguments and return values. While type hints do not enforce type conversion, they serve as valuable documentation and aid in static type checking.

```
def calculate_area(radius: float) -> float:
    return 3.14159 * radius ** 2
```

Adhering to these best practices in type conversion not only enhances the robustness and reliability of your Python code but also contributes to its clarity and maintainability. By validating data, handling errors gracefully, and using the most suitable conversion methods, you can effectively manage the dynamic nature of Python's type system and prevent common pitfalls associated with type conversion.

## 8. SUMMARY

In this unit on Python Syntax and Variables, we embarked on an explorative journey through the fundamental aspects of Python programming that set the stage for more complex and dynamic code writing. Starting with the basics, we delved into the intricacies of Python variables, a cornerstone of programming that allows us to store, retrieve, and manipulate data within our programs. We learned that variables in Python are more than just names for values; they are references to objects stored in memory, each with a unique identity, type, and value.

We also navigated through the rules and conventions for naming variables, known as identifiers, understanding the importance of clear and descriptive names for readability and maintainability of code. The discussion highlighted the constraints and best practices in naming, including the use of lowercase letters for variables and uppercase letters for class names, and the significance of avoiding reserved keywords in Python.

Furthermore, we explored the process of declaring variables and assigning values, uncovering Python's flexibility in data type assignment and the convenience of multiple assignments. This led us to a comprehensive overview of Python's data types, from primitive types like integers, strings, and floats, to complex structures like lists, tuples, and dictionaries, each serving unique purposes in data manipulation and storage.

Through practical examples and exercises, this unit not only strengthened our understanding of Python syntax and variables but also emphasized the dynamic and object-oriented nature of Python, preparing us for the subsequent units where these concepts will be applied in more complex programming scenarios. As we move forward, the foundational knowledge gained here will be instrumental in writing efficient, readable, and robust Python code.

## 9. GLOSSARY

- Variable: A named location in memory used to store data. In Python, variables are dynamically typed and do not need explicit declaration.
- Object: In Python, everything is an object. An object is an instance of a class, encapsulating data (attributes) and behaviors (methods).
- Identity: A unique identifier for an object, often equivalent to the object's memory address. In Python, the id() function returns an object's identity.
- Type: Defines the category of a value and determines the operations that can be performed on it. Python's type() function reveals an object's type.
- Value: The data contained within an object. In Python, variables refer to values (or objects).
- Identifier: A name given to entities like variables, functions, or classes. Identifiers must start with a letter (a-z, A-Z) or an underscore (_), followed by letters, underscores, or digits (0-9).
- Keywords: Reserved words in Python that have special syntactic significance and cannot be used as identifiers. Examples include def, class, if, else, etc.
- Declaration: The act of introducing a new name in the program. In Python, variables are created at the moment a value is first assigned to them.
- Assignment: The process of binding a value to a variable using the assignment operator =.
- Numeric Variables: Variables that store numeric values, including integers (int), floating-point numbers (float), and complex numbers (complex).
- String Variable: Variables that store sequences of characters, enclosed in quotes. Strings in Python are immutable.
- Multiple Assignment: A feature in Python that allows assigning multiple variables in a single statement, e.g., a, b, c = 1, 2, "python".
- Local Variables: Variables declared within a function, accessible only within that function's scope.
- Global Variables: Variables declared outside any function, accessible throughout the program.

- Deleting a Variable: Removing a variable from the namespace using the del keyword, e.g., del variable_name.
- Data Types: Categories of values that determine what operations can be performed on them. Python's standard (primitive) data types include int, float, str, list, tuple, dict, set, bool, etc.
- Strings: A data type in Python for sequences of characters. Strings are immutable and can be defined using single, double, or triple quotes.
- Numbers: Represent numeric values in Python and include int, float, and complex.
- Sequence Type: Include list, tuple, and range, which represent ordered collections of items.
- Mapping Type: Represented by dict, which stores key-value pairs.
- Set Types: Include set and frozenset, which represent unordered collections of unique items.
- Boolean Type: Represents truth values, True and False.
- Binary Types: Include bytes, bytearray, and memoryview, used for binary data manipulation.

## 10. SELF-ASSESSMENT QUESTIONS

1. What is a variable in Python?
   A) A fixed memory location
   B) A reserved memory location that can store a value
   C) A type of Python keyword
   D) A function in Python
2. In Python, when is a variable created?
   A) At the start of the program
   B) When a function is called
   C) When a value is first assigned to it
   D) After the program executes
3. Which of the following is true about Python variables?
   A) They must be declared with a specific type
   B) They can change type after they have been set
   C) They can only store numeric values
   D) They are immutable
4. Which of the following is a valid identifier in Python?
   A) 1_variable
   B) variable_name
   C) variable-name
   D) None of the above
5. Identifiers in Python cannot start with:
   A) An underscore
   B) A lowercase letter
   C) A digit
   D) An uppercase letter
6. Which of the following is NOT allowed in Python identifiers?
   A) Underscores
   B) Digits
   C) Special symbols like @, $, %
   D) Uppercase letters

7. How do you assign the same value to multiple variables in a single statement?
   A) a = b = c = 1
   B) a, b, c = 1
   C) a = 1; b = 1; c = 1;
   D) a, b, c = 1, 1, 1
8. What is the output of the following code snippet?

```
a, b = 10, 20

b, a = a, b

print(a)
```

   A) 10
   B) 20
   C) (10, 20)
   D) Error
9. Which statement is true about local variables in Python?
   A) They can be accessed anywhere in the program
   B) They are declared outside functions
   C) They are accessible only inside the function they are declared
   D) They do not need to be initialized
10. Which of the following is a mutable data type?
   A) String
   B) Tuple
   C) List
   D) Integer
11. What data type is the result of: type(3.5)?
   A) int
   B) float
   C) complex
   D) str
12. How do you create a dictionary with the following pairs: 1:"one", 2:"two"?

A) {1="one", 2="two"}

B) {1:"one", 2:"two"}

C) dict(1="one", 2="two")

D) dict(1:"one", 2:"two")

13. Which of the following is a Python keyword?

A) function

B) elif

C) begin

D) var

14. The del keyword is used to:

A) Declare a variable

B) Modify a variable

C) Delete a variable

D) Debug a variable

15. Which keyword is used to define a function in Python?

A) func

B) define

C) function

D) def

## 11. TERMINAL QUESTIONS

1. Explain the dynamic typing feature of Python variables.
2. Describe the process and significance of memory allocation for variables in Python.
3. How does Python manage variable identities, and how can you verify an object's identity?
4. Outline the rules for naming identifiers in Python and provide examples of valid and invalid identifiers.
5. Discuss the importance of following naming conventions for identifiers in Python.
6. How does Python's case sensitivity affect the naming and usage of identifiers?
7. Illustrate how to perform multiple assignments in Python and provide a use case.
8. Explain the difference between local and global variables in Python with examples.
9. How can you delete a variable in Python, and why might you want to do this?
10. Compare and contrast mutable and immutable data types in Python with examples.
11. Provide examples of sequence data types in Python and discuss their characteristics.
12. Explain the concept of dictionary data types in Python and how they differ from lists and tuples.
13. What role do keywords play in Python syntax, and can you use a keyword as an identifier?
14. Describe how the del keyword is used in Python with an example.
15. Provide examples of how the def and return keywords are used in Python functions.
16. Differentiate between implicit and explicit type conversion in Python with examples.
17. Discuss the use of type conversion functions like int(), float(), and str() in Python with scenarios where each would be necessary.
18. Explain how to handle errors during explicit type conversion in Python.
19. Describe how to work with nested data structures in Python and provide an example of converting all elements of a nested list to strings.
20. Discuss the use and importance of type hints in Python, providing examples of how they can be used in function definitions.

## 12. ANSWERS

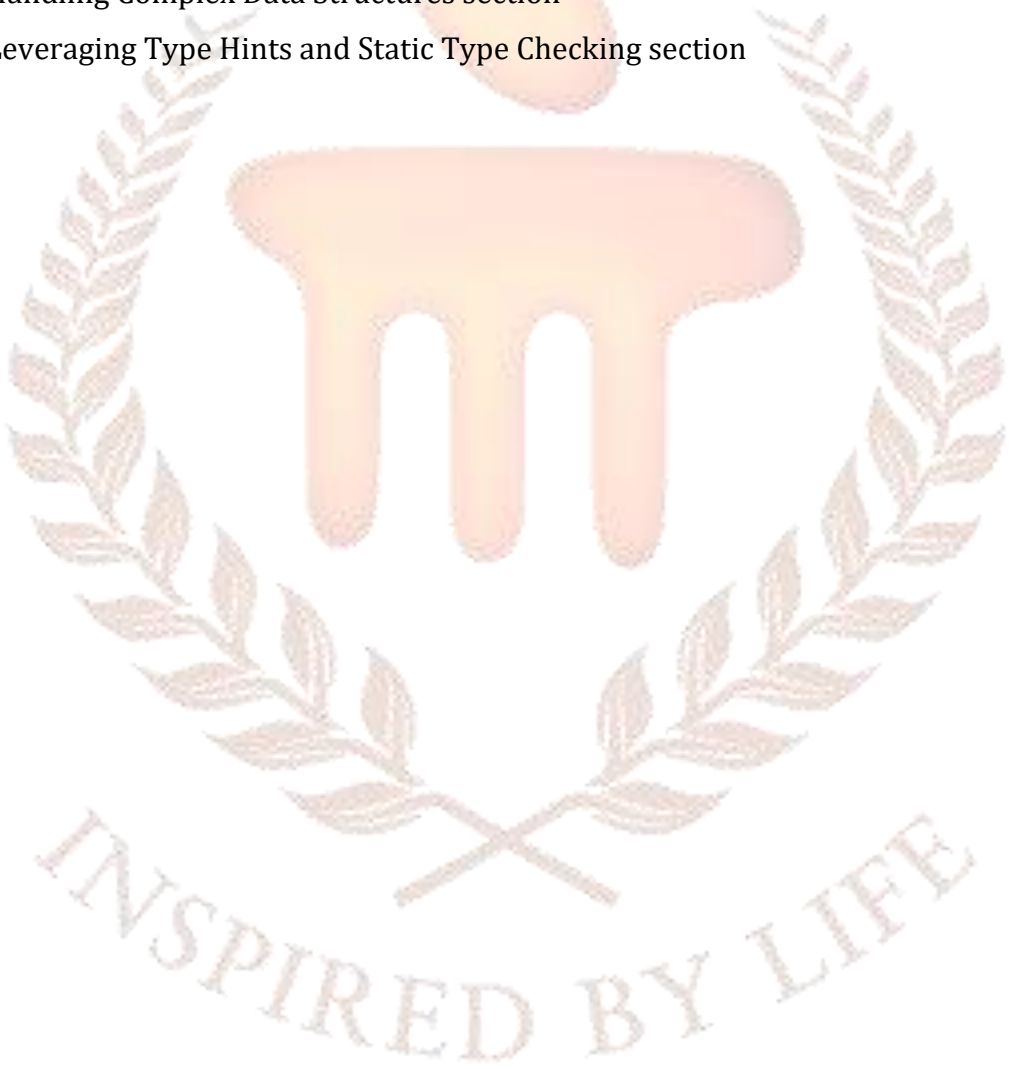
**Self-Assessment Questions**

1. Answer: B) A reserved memory location that can store a value
2. Answer: C) When a value is first assigned to it
3. Answer: B) They can change type after they have been set
4. Answer: B) variable_name
5. Answer: C) A digit
6. Answer: C) Special symbols like @, $, %
7. Answer: A) a = b = c = 1
8. Answer: B) 20
9. Answer: C) They are accessible only inside the function they are declared
10. Answer: C) List
11. Answer: B) float
12. Answer: B) {1:"one", 2:"two"}
13. Answer: B) elif
14. Answer: D) def

**Terminal Questions**

1. Python Variables section
2. Python Variables section
3. Python Variables section
4. Identifiers Rules section
5. Identifiers Rules section
6. Identifiers Rules section
7. Declaring Variable And Assigning Value section
8. Declaring Variable And Assigning Value section
9. Declaring Variable And Assigning Value section
10. Python Datatypes section
11. Python Datatypes section
12. Python Datatypes section

13. Python Keywords section
14. Python Keywords section
15. Python Keywords section
16. Type Conversion section
17. Explicit Type Conversion section
18. Best Practices in Type Conversion section
19. Handling Complex Data Structures section
20. Leveraging Type Hints and Static Type Checking section



## 13. SUGGESTED BOOKS AND E-REFERENCES

**Books:**


- Eric Matthes (2016), Python Crash Course: A Hands-On, Project-Based Introduction to Programming.
- John M. Zelle (2009), Python Programming: An Introduction to Computer Science (Preliminary Second Edition).
- Mark Lutz (2011), Python Programming: A Powerful Object-Oriented Programming (Fourth Edition).
- Sebastian Raschka (2017), Python Machine Learning - Machine Learning and Deep Learning with Python (Edition 2).


**E-REFERENCES:**


- Python Programming Certification Training Course, last viewed on March 23, 2021, <https://www.edureka.co/python-programming-certification-training>
- Python Tutorials and Sample Programs, last viewed on March 23, 2021, <https://www.w3schools.com/python/>
- History of Python, last viewed on March 23, 2021, <https://en.wikipedia.org/wiki/History_of_Python>