**Difference between Python and other programing languages**

Python is often compared to other interpreted languages such as Java, JavaScript, Perl, Tcl, or Smalltalk. Comparisons to C++, Common Lisp and Scheme can also be enlightening. In this section I will briefly compare Python to each of these languages. These comparisons concentrate on language issues only. In practice, the choice of a programming language is often dictated by other real-world constraints such as cost, availability, training, and prior investment, or even emotional attachment. Since these aspects are highly variable, it seems a waste of time to consider them much for this comparison.

## **Java**

Python programs are generally expected to run slower than Java programs, but they also take much less time to develop. Python programs are typically 3-5 times shorter than equivalent Java programs. This difference can be attributed to Python's built-in high-level data types and its dynamic typing. For example, a Python programmer wastes no time declaring the types of arguments or variables, and Python's powerful polymorphic list and dictionary types, for which rich syntactic support is built straight into the language, find a use in almost every Python program. Because of the run-time typing, Python's run time must work harder than Java's. For example, when evaluating the expression a+b, it must first inspect the objects a and b to find out their type, which is not known at compile time. It then invokes the appropriate addition operation, which may be an overloaded user-defined method. Java, on the other hand, can perform an efficient integer or floating point addition, but requires variable declarations for a and b, and does not allow overloading of the + operator for instances of user-defined classes.

For these reasons, Python is much better suited as a "glue" language, while Java is better characterized as a low-level implementation language. In fact, the two together make an excellent combination. Components can be developed in Java and combined to form applications in Python; Python can also be used to prototype components until their design can be "hardened" in a Java implementation. To support this type of development, a Python implementation written in Java is under development, which allows calling Python code from Java and vice versa. In this implementation, Python source code is translated to Java bytecode (with help from a run-time library to support Python's dynamic semantics).

## **Javascript**

Python's "object-based" subset is roughly equivalent to JavaScript. Like JavaScript (and unlike Java), Python supports a programming style that uses simple functions and variables without engaging in class definitions. However, for JavaScript, that's all there is. Python, on the other hand, supports writing much larger programs and better code reuse through a true object-oriented programming style, where classes and inheritance play an important role.

## **Perl**

Python and Perl come from a similar background (Unix scripting, which both have long outgrown), and sport many similar features, but have a different philosophy. Perl emphasizes support for common application-oriented tasks, e.g. by having built-in regular expressions, file scanning and report generating features. Python emphasizes support for common programming methodologies such as data structure design and object-oriented programming, and encourages programmers to write readable (and thus maintainable) code by providing an elegant but not overly cryptic notation. As a consequence, Python comes close to Perl but rarely beats it in its original application domain; however Python has an applicability well beyond Perl's niche.

## **Tcl**

Like Python, Tcl is usable as an application extension language, as well as a stand-alone programming language. However, Tcl, which traditionally stores all data as strings, is weak on data structures, and executes typical code much slower than Python. Tcl also lacks features needed for writing large programs, such as modular namespaces. Thus, while a "typical" large application using Tcl usually contains Tcl extensions written in C or C++ that are specific to that application, an equivalent Python application can often be written in "pure Python". Of course, pure Python development is much quicker than having to write and debug a C or C++ component. It has been said that Tcl's one redeeming quality is the Tk toolkit. Python has adopted an interface to Tk as its standard GUI component library.

Tcl 8.0 addresses the speed issuse by providing a bytecode compiler with limited data type support, and adds namespaces. However, it is still a much more cumbersome programming language.

## **Smalltalk**

Perhaps the biggest difference between Python and Smalltalk is Python's more "mainstream" syntax, which gives it a leg up on programmer training. Like Smalltalk, Python has dynamic typing and binding, and everything in Python is an object. However, Python distinguishes built-in object types from user-defined classes, and currently doesn't allow inheritance from built-in types. Smalltalk's standard library of collection data types is more refined, while Python's library has more facilities for dealing with Internet and WWW realities such as email, HTML and FTP.

Python has a different philosophy regarding the development environment and distribution of code. Where Smalltalk traditionally has a monolithic "system image" which comprises both the environment and the user's program, Python stores both standard modules and user modules in individual files which can easily be rearranged or distributed outside the system. One consequence is that there is more than one option for attaching a Graphical User Interface (GUI) to a Python program, since the GUI is not built into the system.

## **C++**

Almost everything said for Java also applies for C++, just more so: where Python code is typically 3-5 times shorter than equivalent Java code, it is often 5-10 times shorter than equivalent C++ code! Anecdotal evidence suggests that one Python programmer can finish in two months what two C++ programmers can't complete in a year. Python shines as a glue language, used to combine components written in C++.

## **Common Lisp and Scheme**

These languages are close to Python in their dynamic semantics, but so different in their approach to syntax that a comparison becomes almost a religious argument: is Lisp's lack of syntax an advantage or a disadvantage? It should be noted that Python has introspective capabilities similar to those of Lisp, and Python programs can construct and execute program fragments on the fly. Usually, real-world properties are decisive: Common Lisp is big (in every sense), and the Scheme world is fragmented between many incompatible versions, where Python has a single, free, compact implementation.

Python Data Types

Variables can hold values, and every value has a data-type. Python is a dynamically typed language; hence we do not need to define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.

1. a = 5

The variable **a** holds integer value five and we did not define its type. Python interpreter will automatically interpret variables **a** as an integer type.

Python enables us to check the type of the variable used in the program. Python provides us the **type ()** function, which returns the type of the variable passed.

Consider the following example to define the values of different data types and checking its type.

1. a=10
2. b="Hi Python"
3. c = 10.5
4. **print**(type(a))
5. **print**(type(b))
6. **print**(type(c))

**Output:**

<type 'int'>

<type 'str'>

<type 'float'>

## **Standard data types**

A variable can hold different types of values. For example, a person's name must be stored as a string whereas its id must be stored as an integer.

Python provides various standard data types that define the storage method on each of them. The data types defined in Python are given below.

[Numbers](https://www.javatpoint.com/python-data-types" \l "numbers)

[Sequence Type](https://www.javatpoint.com/python-data-types" \l "SequenceType)

[Boolean](https://www.javatpoint.com/python-data-types" \l "Boolean)

[Set](https://www.javatpoint.com/python-data-types" \l "Set)

[Dictionary](https://www.javatpoint.com/python-data-types" \l "dictionary)



### **Numbers**

Number stores numeric values. The integer, float, and complex values belong to a Python Numbers data-type. Python provides the **type()** function to know the data-type of the variable. Similarly, the **isinstance()** function is used to check an object belongs to a particular class.

Python creates Number objects when a number is assigned to a variable. For example;

1. a = 5
2. **print**("The type of a", type(a))
4. b = 40.5
5. **print**("The type of b", type(b))
7. c = 1+3j
8. **print**("The type of c", type(c))
9. **print**(" c is a complex number", isinstance(1+3j,complex))

**Output:**

The type of a <class 'int'>

The type of b <class 'float'>

The type of c <class 'complex'>

c is complex number: True

Python supports three types of numeric data.

1. **Int -** Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to **int**
2. **Float -** Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate upto 15 decimal points.
3. **complex -** A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts, respectively. The complex numbers like 2.14j, 2.0 + 2.3j, etc.

### **Sequence Type**

### **String**

The string can be defined as the sequence of characters represented in the quotation marks. In Python, we can use single, double, or triple quotes to define a string.

String handling in Python is a straightforward task since Python provides built-in functions and operators to perform operations in the string.

In the case of string handling, the operator + is used to concatenate two strings as the operation "hello"+" python" returns "hello python".

The operator \* is known as a repetition operator as the operation "Python" \*2 returns 'Python Python'.

The following example illustrates the string in Python.

**Example - 1**

1. str = "string using double quotes"
2. **print**(str)
3. s = '''''A multiline
4. string'''
5. **print**(s)

**Output:**

string using double quotes

A multiline

string

Consider the following example of string handling.

**Example - 2**

1. str1 = 'hello java' #string str1
2. str2 = ' how are you' #string str2
3. **print** (str1[0:2]) #printing first two character using slice operator
4. **print** (str1[4]) #printing 4th character of the string
5. **print** (str1\*2) #printing the string twice
6. **print** (str1 + str2) #printing the concatenation of str1 and str2

**Output:**

he

o

hello javahello java

hello java how are you

### **List**

Python Lists are similar to arrays in C. However, the list can contain data of different types. The items stored in the list are separated with a comma (,) and enclosed within square brackets [].

We can use slice [:] operators to access the data of the list. The concatenation operator (+) and repetition operator (\*) works with the list in the same way as they were working with the strings.

Consider the following example.

1. list1 = [1, "hi", "Python", 2]
2. #Checking type of given list
3. **print**(type(list1))
5. #Printing the list1
6. **print** (list1)
8. # List slicing
9. **print** (list1[3:])
11. # List slicing
12. **print** (list1[0:2])
14. # List Concatenation using + operator
15. **print** (list1 + list1)
17. # List repetation using \* operator
18. **print** (list1 \* 3)

**Output:**

[1, 'hi', 'Python', 2]

[2]

[1, 'hi']

[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]

[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]

### **Tuple**

A tuple is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma (,) and enclosed in parentheses ().

A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.

Let's see a simple example of the tuple.

1. tup  = ("hi", "Python", 2)
2. # Checking type of tup
3. **print** (type(tup))
5. #Printing the tuple
6. **print** (tup)
8. # Tuple slicing
9. **print** (tup[1:])
10. **print** (tup[0:1])
12. # Tuple concatenation using + operator
13. **print** (tup + tup)
15. # Tuple repatation using \* operator
16. **print** (tup \* 3)
18. # Adding value to tup. It will throw an error.
19. t[2] = "hi"

**Output:**

<class 'tuple'>

('hi', 'Python', 2)

('Python', 2)

('hi',)

('hi', 'Python', 2, 'hi', 'Python', 2)

('hi', 'Python', 2, 'hi', 'Python', 2, 'hi', 'Python', 2)

Traceback (most recent call last):

File "main.py", line 14, in <module>

t[2] = "hi";

TypeError: 'tuple' object does not support item assignment

### **Dictionary**

Dictionary is an unordered set of a key-value pair of items. It is like an associative array or a hash table where each key stores a specific value. Key can hold any primitive data type, whereas value is an arbitrary Python object.

The items in the dictionary are separated with the comma (,) and enclosed in the curly braces {}.

Consider the following example.

1. d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'}
3. # Printing dictionary
4. **print** (d)
6. # Accesing value using keys
7. **print**("1st name is "+d[1])
8. **print**("2nd name is "+ d[4])
10. **print** (d.keys())
11. **print** (d.values())

**Output:**

1st name is Jimmy

2nd name is mike

{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}

dict\_keys([1, 2, 3, 4])

dict\_values(['Jimmy', 'Alex', 'john', 'mike'])

### **Boolean**

Boolean type provides two built-in values, True and False. These values are used to determine the given statement true or false. It denotes by the class bool. True can be represented by any non-zero value or 'T' whereas false can be represented by the 0 or 'F'. Consider the following example.

1. # Python program to check the boolean type
2. **print**(type(True))
3. **print**(type(False))
4. **print**(false)

**Output:**

<class 'bool'>

<class 'bool'>

NameError: name 'false' is not defined

### **Set**

Python Set is the unordered collection of the data type. It is iterable, mutable (can modify after creation), and has unique elements. In set, the order of the elements is undefined; it may return the changed sequence of the element. The set is created by using a built-in function **set(),** or a sequence of elements is passed in the curly braces and separated by the comma. It can contain various types of values.

**Properties of set()**

* No parameters are passed to create the empty set
* Dictionary can also be created using set, but only keys remain after conversion, values are lost.

Consider the following example.

1. # Creating Empty set
2. set1 = set()
4. set2 = {'James', 2, 3,'Python'}
6. #Printing Set value
7. **print**(set2)
9. # Adding element to the set
11. set2.add(10)
12. **print**(set2)
14. #Removing element from the set
15. set2.remove(2)
16. **print**(set2)

**Output:**

{3, 'Python', 'James', 2}

{'Python', 'James', 3, 2, 10}

{'Python', 'James', 3, 10}

# Type Conversion in Python

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.

**Example:**

* Python3

|  |
| --- |
| x **=** 10    **print**("x is of type:", type(x))    y **=** 10.6  **print**("y is of type:" type(y))    x **=** x **+** y    print(x)  print("x is of type:",type(x)) |

**Output:**

x is of type: <class 'int'>

y is of type: <class 'float'>

20.6

x is of type: <class 'float'>

As we can see the type of ‘x’ got automatically changed to the “float” type from the “integer” type. this is a simple case of Implicit type conversion in python.

## **Explicit Type Conversion**

In Explicit Type Conversion in Python, the data type is manually changed by the user as per their requirement. 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**.

* Python3

|  |
| --- |
| # 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.**

* Python3

|  |
| --- |
| # 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.**

* Python3

|  |
| --- |
| # Convert ASCII value to characters  a **=** chr(76)  b **=** chr(77)    print(a)  print(b) |

**Output:**

L

M

References: https://www.geeksforgeeks.org/type-conversion-python/