Introduction

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to **object-oriented programming**. Python's elegant syntax and **dynamic typing**, together with its **interpreted** nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications.

The original and most-maintained implementation of Python is called **CPython**, written in C. New language features generally appear here first.

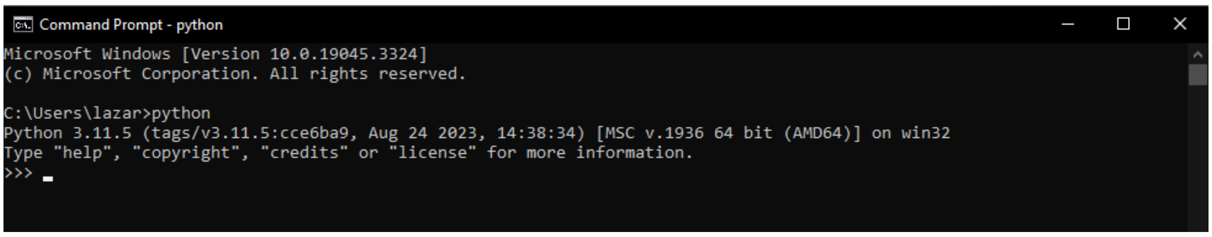
Python enables programs to be written compactly and readably. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons:

- the high-level data types allow you to express complex operations in a single statement;
- statement grouping is done by indentation instead of beginning and ending brackets;
- no variable or argument declarations are necessary.

Using the Python Interpreter

Invoking the Interpreter

The Python interpreter is usually installed in **C:/Users/{USER_NAME}/AppData/Local/Microsoft/WindowsApps/** on Windows machines where you have installed Python from the Microsoft Store.
Adding the previous value as an entry in the PATH environment variable will enable you to use the command directly in the terminal:



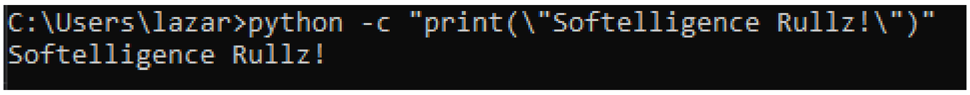
```
Microsoft Windows [Version 10.0.19045.3324]
(c) Microsoft Corporation. All rights reserved.

C:\Users\lazar>python
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Since the choice of the directory where the interpreter lives is an installation option, other places are possible.

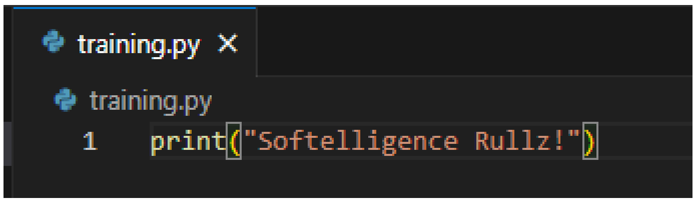
The interpreter operates somewhat like the Unix shell, or Windows cmd: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a script from that file.

A second way of starting the interpreter is **python -c command [arg] ...**, which executes the statement(s) in command, analogous to the shell's **-c** option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote the command in its entirety.

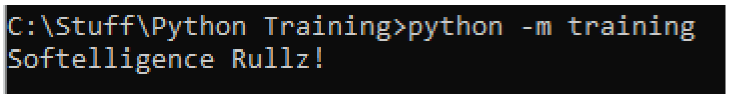


```
C:\Users\lazar>python -c "print(\"Softelligence Rullz!\")"
Softelligence Rullz!
```

Some Python modules are also useful as scripts. These can be invoked using **python -m module [arg] ...**, which executes the source file for the module as if you had spelled out its full name on the command line.



```
print("Softelligence Rullz!")
```



```
C:\Stuff\Python Training>python -m training
Softelligence Rullz!
```

Argument passing

When known to the interpreter, the script name and additional arguments thereafter are turned into a list of strings and assigned to the **argv** variable in the **sys module**. You can access this list by executing import sys. The length of the list is at least one; when no script and no arguments are given, **sys.argv[0]** is an empty string. When the script name is given as **'-'** (meaning standard input), **sys.argv[0]** is set to **'-'**. When **-c** command is used, **sys.argv[0]** is set to **'-c'**. When **-m** module is used, **sys.argv[0]** is set to the full name of the located module. Options found after **-c** command or **-m** module are not consumed by the Python interpreter's option processing but left in sys.argv for the command or module to handle.

Example:

Consider the following script test.py:

```
import sys

print (f"Number of arguments: {len(sys.argv)} arguments.")
print (f"Argument List: {str(sys.argv)}")
```

Now run the above script as below:

```
python test.py arg1 arg2 arg3
```

The output will be the following:

```
Number of arguments: 4 arguments.
Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
```

Interactive Mode

When commands are read from a tty, the interpreter is said to be in interactive mode. In this mode it prompts for the next command with the primary prompt, usually three greater-than signs **(>>>)**; for continuation lines it prompts with the secondary prompt, by default three dots **(...)**. The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt (as seen in the first image above):

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this statement:

```
C:\Stuff\Python Training>python
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> training = True
>>> if training:
...     print("We are learning Python together.")
...
We are learning Python together.
>>>
```

Understanding Python's logic and structure

A Python program is read by a parser. Input to the parser is a stream of tokens, generated by the lexical analyzer. This chapter describes how the lexical analyzer breaks a file into tokens.

Python reads program text as **Unicode** code points; the encoding of a source file can be given by an encoding declaration and defaults to **UTF-8**, see **PEP 3120** for details. If the source file cannot be decoded, a **SyntaxError** is raised.

Comments

A comment starts with a hash character (**#**) that is not part of a string literal, and ends at the end of the physical line. A comment signifies the end of the logical line unless the implicit line joining rules are invoked. Comments are ignored by the syntax.

Indentation

Leading whitespace (**spaces** and **tabs**) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements.

Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important. Python uses indentation to indicate a block of code.

Indentation is rejected as inconsistent if a source file mixes tabs and spaces in a way that makes the meaning dependent on the worth of a tab in spaces; a **TabError** is raised in that case.

Identifiers

Identifiers (also referred to as names) are described by the following lexical definitions.

The syntax of identifiers in Python is based on the Unicode standard annex UAX-31, with elaboration and changes as defined below; see also PEP 3131 for further details.

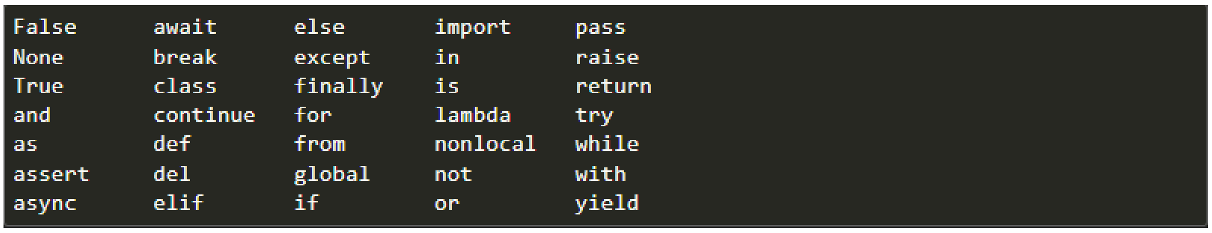
Within the ASCII range (U+0001..U+007F), the valid characters for identifiers are the same as in Python 2.x: the uppercase and lowercase letters A through Z, the underscore _ and, except for the first character, the digits 0 through 9.

Python 3.0 introduces additional characters from outside the ASCII range (see PEP 3131). For these characters, the classification uses the version of the Unicode Character Database as included in the unicodedata module.

Identifiers are unlimited in length. Case is significant.

Keywords

The following identifiers are used as reserved words, or keywords of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:



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

Operators and Data Types

Numbers

These are created by numeric literals and returned as results by arithmetic operators and arithmetic built-in functions. Numeric objects are **immutable**; once created their value never changes. Python numbers are of course strongly related to mathematical numbers, but subject to the limitations of numerical representation in computers.

Python distinguishes between integers, floating point numbers, and complex numbers:

1. numbers.Integral

These represent elements from the mathematical set of integers (positive and negative).

There are two types of integers:

> **Integers (int)**
> These represent numbers in an unlimited range, subject to available (virtual) memory only. For the purpose of shift and mask operations, a binary representation is assumed, and negative numbers are represented in a variant of 2’s complement which gives the illusion of an infinite string of sign bits extending to the left.
>
> **Booleans (bool)**
> These represent the truth values False and True. The two objects representing the values False and True are the only Boolean objects. The Boolean type is a subtype of the integer type, and Boolean values behave like the values 0 and 1, respectively, in almost all contexts, the exception being that when converted to a string, the strings "False" or "True" are returned, respectively.

2. numbers.Real (float)

These represent machine-level double precision floating point numbers. You are at the mercy of the underlying machine architecture (and C or Java implementation) for the accepted range and handling of overflow. Python does not support single-precision floating point numbers; the savings in processor and memory usage that are usually the reason for using these are dwarfed by the overhead of using objects in Python, so there is no reason to complicate the language with two kinds of floating point numbers.

3. numbers.Complex (complex)

These represent complex numbers as a pair of machine-level double precision floating point numbers. The same caveats apply as for floating point numbers. The real and imaginary parts of a complex number z can be retrieved through the read-only attributes z.real and z.imag.

Examples:

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators **+**, **-**, ***** and **/** can be used to perform arithmetic; parentheses **(())** can be used for grouping. For example:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating point number
1.6
```

The integer numbers (e.g. 2, 4, 20) have type **int**, whereas the ones with a fractional part (e.g. 5.0, 1.6) have type **float**.

Division (**/**) always returns a **float**. To do **floor division** and get an **integer** result you can use the **//** operator; to calculate the remainder you can use **%**:

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # floored quotient * divisor + remainder
17
```

With Python, it is possible to use the ****** operator to calculate powers:

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

The equal sign (**=**) is used to assign a value to a **variable**. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

If a variable is not **"defined"** (assigned a value), trying to use it will give you an error:

```
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>> 4 * 3.75 - 1
14.0
```

In interactive mode, the last printed expression is assigned to the variable **_**. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

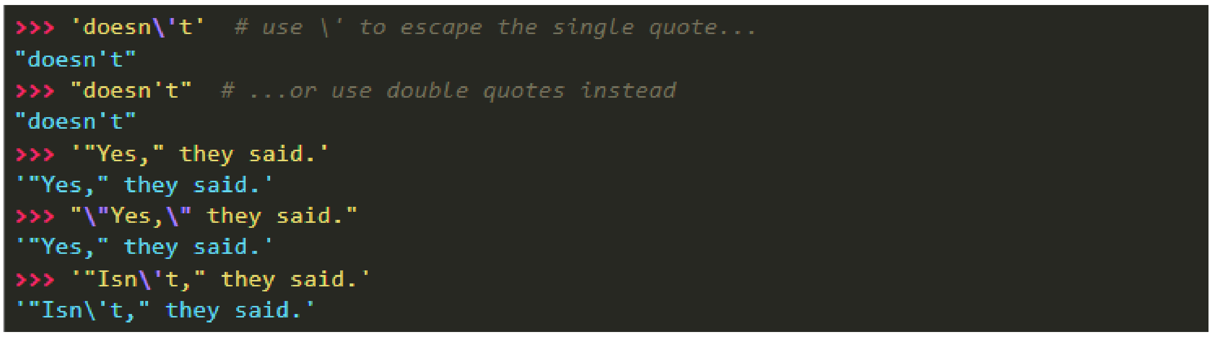
This variable should be treated as read-only by the user. Don't explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

Text

Python can manipulate text (represented by type **str**, so-called “strings”) as well as numbers. This includes characters “!”, words “rabbit”, names “Paris”, sentences “Got your back.”, etc. “Yay! :)”. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result

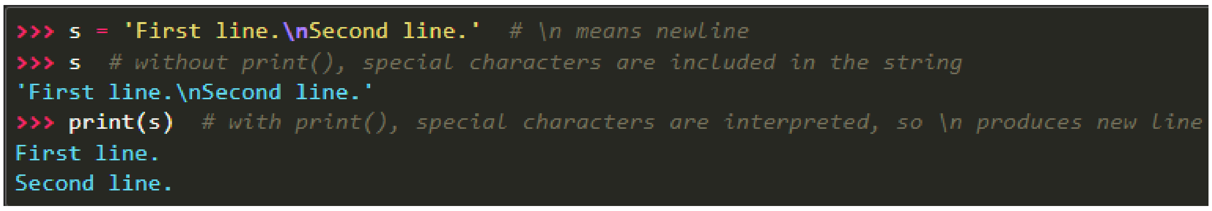
Examples:

To quote a quote, we need to **“escape”** it, by preceding it with \. Alternatively, we can use the other type of quotation marks:



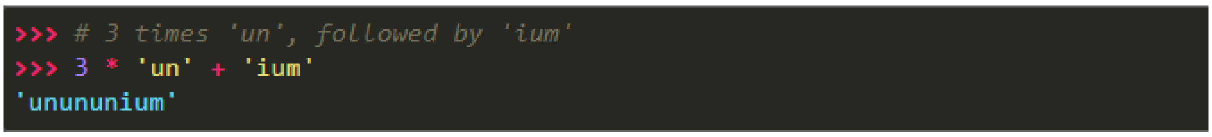
```
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

In the Python shell, the string definition and output string can look different. The **print(**) function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:



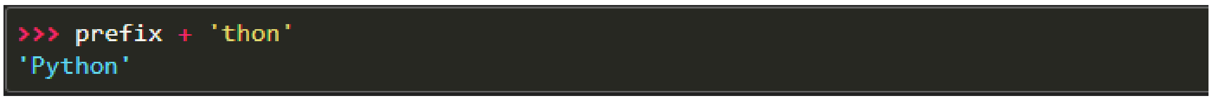
```
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), special characters are included in the string
'First line.\nSecond line.'
>>> print(s)  # with print(), special characters are interpreted, so \n produces new line
First line.
Second line.
```

Strings can be concatenated (glued together) with the **+** operator, and repeated with *****:



```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

If you want to concatenate variables or a variable and a literal, use **+**:



```
>>> prefix + 'thon'
'Python'
```

Strings can be **indexed** (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

In addition to indexing, **slicing** is also supported. While indexing is used to obtain individual characters, slicing allows you to obtain substring:

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

One way to remember how slices work is to think of the indices as pointing **between** characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of **n** characters has index **n**, for example:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0...6 in the string; the second row gives the corresponding negative indices. The slice from i to j consists of all characters between the edges labeled i and j, respectively.

Attempting to use an index that is too large will result in an error:

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Python strings cannot be changed — they are **immutable**. Therefore, assigning to an indexed position in the string results in an error

Type Casting

There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.

Casting in python is therefore done using constructor functions:

- int() - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)
- float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Implicit type conversion:

```
# Python program to demonstrate
# implicit type Casting

# Python automatically converts
# a to int
a = 7
print(type(a))

# Python automatically converts
# b to float
b = 3.0
print(type(b))

# Python automatically converts
# c to float as it is a float addition
c = a + b
print(c)
print(type(c))

# Python automatically converts
# d to float as it is a float multiplication
d = a * b
print(d)
print(type(d))
```

Output:

```
<class 'int'>

<class 'float'>

10.0

<class 'float'>

21.0

<class 'float'>
```

Explicit type conversion:

Int to Float:

```python
# Python program to demonstrate
# type Casting: int to float

# int variable
a = 5

# typecast to float
n = float(a)

print(n)
print(type(n))
```

Output:

```
5.0

<class 'float'>
```

Float to Int:

```python
# Python program to demonstrate
# type Casting: float to int

# int variable
a = 5.9
```

```
# typecast to int
n = int(a)

print(n)
print(type(n))
```

Output:

```
5

<class 'int'>
```

Int to String:

```
# Python program to demonstrate
# type Casting: int to str

# int variable
a = 5

# typecast to str
n = str(a)

print(n)
print(type(n))
```

Output:

```
5

<class 'str'>
```