Introduction

https://www.w3schools.com/python/python_intro.asp (https://www.w3schools.com/python/python_intro.asp)

Installation

https://www.w3schools.com/python/python_getstarted.asp (https://www.w3schools.com/python/python_getstarted.asp)

Scripts vs Modules

The main difference between a module and a script is that modules are meant to be imported, while scripts are made to be directly executed.

What is Python Interpreter?

Python is an excellent programming language that allows you to be productive in a wide variety of fields. Python is also a piece of software called an interpreter. The interpreter is the program you'll need to run Python code and scripts. Technically, the interpreter is a layer of software that works between your program and your computer hardware to get your code running. Depending on the Python implementation you use, the interpreter can be:

1. A program written in C, like CPython, which is the core implementation of the language
2. A program written in Java, like Jython
3. A program written in Python itself, like PyPy
4. A program implemented in .NET, like IronPython

Whatever form the interpreter takes, the code you write will always be run by this program. Therefore, the first condition to be able to run Python scripts is to have the interpreter correctly installed on your system.

The interpreter is able to run Python code in two different ways:

1. As a script or module

2. As a piece of code typed into an interactive session

How to Run Python Code Interactively

A widely used way to run Python code is through an interactive session. To start a Python interactive session, just open a command-line or terminal and then type in python, or python3 depending on your Python installation, and then hit Enter. The standard prompt for the interactive mode is >>>, so as soon as you see these characters, you'll know you are in.

Now, you can write and run Python code as you wish, with the only drawback being that when you **close the session, your code will be gone.**

When you work interactively, every expression and statement you type in is evaluated and executed immediately. An interactive session will allow you to test every piece of code you write, which makes it an awesome development tool and an excellent place to experiment with the language and test Python code on the fly.

To exit interactive mode, you can use one of the following options:

quit() or exit(), which are built-in functions The Ctrl+Z and Enter key combination on Windows, or just Ctrl+D on Unix-like systems

How Does the Interpreter Run Python Scripts?

When you try to run Python scripts, a multi-step process begins. In this process the interpreter will:

1. Process the statements of your script in a sequential fashion
2. Compile the source code to an intermediate format known as bytecode
3. This bytecode is a translation of the code into a lower-level language that's platform-independent. Its purpose is to optimize code execution. So, the next time the interpreter runs your code, it'll bypass this compilation step. Strictly speaking, this code optimization is only for modules (imported files), not for executable scripts.
4. Ship off the code for execution

At this point, something known as a Python Virtual Machine (PVM) comes into action. The PVM is the runtime engine of Python. It is a cycle that iterates over the instructions of your bytecode to run them one by one.

The PVM is not an isolated component of Python. It's just part of the Python system you've installed on your machine. Technically, the PVM is the last step of what is called the Python interpreter.

The whole process to run Python scripts is known as the Python Execution Model.

How to Run Python Scripts Using the Command-Line

How to Run Python Scripts Using the Command Line

A Python interactive session will allow you to write a lot of lines of code, but once you close the session, you lose everything you've written. That's why the usual way of writing Python programs is by using plain text files. By convention, those files will use the .py extension. (On Windows systems the extension can also be .pyw). Python code files can be created with any plain text editor.

Using the python Command: To run Python scripts with the python command, you need to open a command-line and type in the word python, or python3 if you have both versions, followed by the path to your script.

Redirecting the Output Sometimes it's useful to save the output of a script for later analysis. Here's how you can do that: **$ python3 hello.py > output.txt**

This operation redirects the output of your script to output.txt, rather than to the standard system output (stdout). The process is commonly known as stream redirection and is available on both Windows and Unix-like systems.

If output.txt doesn't exist, then it's automatically created. On the other hand, if the file already exists, then its contents will be replaced with the new output.

The output will be appended to the end of output.txt using: $ python3 hello.py >> output.txt

Dynamically Typed

Python is a dynamically typed language.We don't have to declare the type of variable while assigning a value to a variable in Python. Other languages like C, C++, Java, etc.., there is a strict declaration of variables before assigning values to them. Python don't have any problem even if we don't declare the type of variable. It states the kind of variable in the runtime of the program.

Strongly Typed Language

Python is a strongly-typed language which means that it is restrictive about how the data types can be intermingled. The interpreter keeps track of all variables types.

Python Reserved Words

In [1]:
```
1 import keyword
2 keyword.kwlist
```

Out[1]:
```
['False',
 'None',
 'True',
 'and',
 'as',
 'assert',
 'async',
 'await',
 'break',
 'class',
 'continue',
 'def',
 'del',
 'elif',
 'else',
 'except',
 'finally',
 'for',
 'from',
 'global',
 'if',
 'import',
 'in',
 'is',
 'lambda',
 'nonlocal',
 'not',
 'or',
 'pass',
 'raise',
 'return',
 'try',
 'while',
 'with',
 'yield']
```

Indentation

In [2]:
```
# Other languages: Only for readability
# Python: Indicate block of code

if 2>1:
    print("Yes")
else:
    print("No")
```

```
Yes
```

In [3]:
```
if 2>1:
    print("Yes")
    print("Done")
else:
    print("No")
```

```
Yes
Done
```

Comments

In [4]:
```
# This is a single-line comment
# Comment 1
# For multi-line comment, put a # at the start of evert line

print("hello")  # This is a comment

"""
Alternatively, we can also use multiline string for a multiline comment.
Since Python will ignore string literals that are not assigned to a variable,
you can add a multiline string (triple quotes) in your code, and place your comment inside it.
"""
print("World")

```

```
hello
World
```

Variables

- A variable is created the moment you first assign a value to it.
- Variables do not need to be declared with any particular type, and can even change type after they have been set.
- Rules for naming variable:
  - start with a letter or '_'
  - cannot start with a number
  - can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
  - Variable names are case sensitive

In [5]:
```
#type(variable_name)
x = 10
print(type(x))
print(x)
```

```
<class 'int'>
10
```

In [6]:
```
1 x = 10.5
2 print(type(x))
3 print(x)
```

```
<class 'float'>
10.5
```

In [7]:
```
1 x = "10"
2 print(type(x))
3 print(x)
```

```
<class 'str'>
10
```

In [8]:
```
1 x = "Hello"
2 print(type(x))
3 print(x)
4
5 x = 'World'
6 print(type(x))
7 print(x)
```

```
<class 'str'>
Hello
<class 'str'>
World
```

In [9]:
```
1 x = 10
2 print(type(x))
3 print(x)
4
5 x = "hi"
6 print(type(x))
7 print(x)
```

```
<class 'int'>
10
<class 'str'>
hi
```

Casting: If you want to specify the data type of a variable, this can be done with casting.

In [10]:
```
x = str(10)    # x will be '10'
print(type(x))
print(x)

y = int(10)    # y will be 10
print(type(y))
print(y)

z = float(10)  # z will be 10.0
print(type(z))
print(z)
```

```
<class 'str'>
10
<class 'int'>
10
<class 'float'>
10.0
```

Assign Multiple Values

In [11]:
```
x, y, z = "One", "Two", "Three"
print(x)
print(y)
print(z)
```

```
One
Two
Three
```

In [12]:
```
x = y = z = "Test"
print(x)
print(y)
print(z)
```

```
Test
Test
Test
```

In [13]:
```
# Unpacking a collection: extract the values into variables

names = ["Ram", 10, "Sita"]
x, y, z = names
print(x)
print(y)
print(z)
```

```
Ram
10
Sita
```

Output a variable

In [14]:
```
# printf("python is a %s",x);
x = "language"
print("Python is a " + x)
```

```
Python is a language
```

In [15]:
```
x = "Python is a "
y = "language"
z =  x + y
print(z)
```

```
Python is a language
```

In [16]:
```
# For numbers, the + character works as a mathematical operator:

x = 1
y = 2
print(x + y)
```

```
3
```

In [17]:
```
# Using + operator to combine a string and a number will give error

x = 10
y = "Hello"
print(x + y)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-17-6a6cf4f6bf31> in <module>
      3 x = 10
      4 y = "Hello"
----> 5 print(x + y)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Global Variables

In [18]:
```
x = 10

def fun():
  print(x)

fun()
```

```
10
```

Data Types

- **Text Types:** str
- **Numeric Types:** int, float, complex
- **Sequence Types:** list, tuple, range
- **Mapping Type:** dict
- **Set Types:** set, frozenset
- **Boolean Type:** bool
- **Binary Types:** bytes, bytearray, memoryview

Numbers

- int: whole number, positive or negative, without decimals, of unlimited length.
- float: decimal, exponential numbers
- complex: 5j

Type Conversion

In [19]:
```
# We cannot convert complex number to any type

x = 1    # int
y = 2.8  # float
z = 1j   # complex

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)

#convert from int to complex:
c = complex(x)

print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```

```
1.0
2
(1+0j)
<class 'float'>
<class 'int'>
<class 'complex'>
```

Random Number

Python does not have a random() function to make a random number, but Python has a built-in module called random that can be used to make random numbers:

In [20]:
```
import random
print(random.randrange(10, 20))
```

```
10
```

input() Function

The input() function allows user input.

In [21]:
```
1 x = input('Enter your name: ')
```

```
Enter your name: Shyam
```

In [22]:
```
1 print(x)
```

```
Shyam
```

In [23]:
```
1 print('Hello, ' + x)
```

```
Hello, Shyam
```

In [24]:
```
1 a = input('Enter number: ')
2 print(a)
3 print(type(a))
```

```
Enter number: 5
5
<class 'str'>
```

In [25]:
```
1 a = input('Enter number: ')
2 a = int(a)
3 print(type(a))
```

```
Enter number: 5
<class 'int'>
```

In [26]:
```
1 a = int(input("Enter number: "))
2 print(type(a))
```

```
Enter number: 5
<class 'int'>
```

In [27]:
```
# a = int(a)
```

In [28]:
```
a = int(input('Enter number: '))
print(a)
print(type(a))
```

```
Enter number: 5
5
<class 'int'>
```

Some more Built-in functions in Python: https://www.w3schools.com/python/python_ref_functions.asp (https://www.w3schools.com/python/python_ref_functions.asp)

Output Formatting

The print() function

Syntax: print(object(s), sep=separator, end=end, file=file, flush=flush)

- object(s): Any object, and as many as you like.
  - Will be converted to string before printed
- sep='separator': Optional. Specify how to separate the objects, if there is more than one.
  - Default is ' '
- end='end': Optional. Specify what to print at the end.
  - Default is '\n' (line feed)
- file: Optional. An object with a write method.
  - Default is sys.stdout
- flush: Optional. A Boolean, specifying if the output is flushed (True) or buffered (False).
  - Default is False

In [29]:
```
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

In [30]:
```
print("Hello" + "World")
```

```
HelloWorld
```

In [31]:
```
print("Hello")
print("World")
```

```
Hello
World
```

In [32]:
```
print("Hello....World")
```

```
Hello....World
```

In [33]:
```
print("Hello", end = "#")
print("World", end = "*")
print("Hi")
print("Hey")
```

```
Hello#World*Hi
Hey
```

In [34]:
```
1 print("Hello", "World", "10")
```

```
Hello World 10
```

In [35]:
```
1 print("Hello", "World", sep=" ")
```

```
Hello World
```

In [36]:
```
1 'Ram' + 'SHyam'
```

Out[36]:
```
'RamSHyam'
```

In [37]:
```
1 'Ram' + str(10)
```

Out[37]:
```
'Ram10'
```

In [38]:
```
1 age = 21
2 name = 'Ram'
```

In [39]:
```
1 # Approach 1
2 print('my age is '+ str(age))
```

```
my age is 21
```

In [40]:
```
1 # Approach 2
2 print('my age is',age,'and my name is',name)
```

```
my age is 21 and my name is Ram
```

In [41]:
```
1 # Approach 3: using f-string
2 print(f'my age is {age} and my name is {name}')
```

```
my age is 21 and my name is Ram
```

In [42]:
```
age = 10
name = "Ram"
print('hello, ',name,' how are you?',age)
```

```
hello,  Ram  how are you? 10
```

In [43]:
```
for i in range(10):
    print(i, end= '*')
```

```
0*1*2*3*4*5*6*7*8*9*
```

In [44]:
```
# Using join method:
print(' '.join(["Joe is", 42, "years old"]))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-44-16ecd124c7e3> in <module>
      1 # Using join method:
----> 2 print(' '.join(["Joe is", 42, "years old"]))

TypeError: sequence item 1: expected str instance, int found
```

In [45]:
```
# It’s safer to just unpack the sequence with the star operator (*) and let print() handle type
print(*[10, 20, 30], sep="#")
```

```
10#20#30
```

In [46]:
```
# When two non-Boolean values are joined by 'and' or 'or', the value of the expression is one of
# operands, not True or False.

a=100
b=200
a or b
```

Out[46]:
```
100
```

In [47]:
```
x = 0.1 + 0.2
print(x)
x

# Explanation: floating-point numbers are implemented in computer hardware as binary fractions a
# computer only understands binary (0 and 1). Due to this reason, most of the decimal fractions
# cannot be accurately stored in our computer. Let's take an example. We cannot represent the fr
# as a decimal number. This will give 0.33333333... which is infinitely long, and we can only ap
# Unfortunately, most decimal fractions cannot be represented exactly as binary fractions.

# A consequence is that, in general, the decimal floating-point numbers you enter are only appro
# the binary floating-point numbers actually stored in the machine.

# Using Python's decimal class would give you better results.

# Documentation link: https://docs.python.org/3.3/tutorial/floatingpoint.html
```

```
0.30000000000000004
```

Out[47]:
```
0.30000000000000004
```

In [48]:
```
# Solution: Use decimal class when we want to carry out decimal calculations as we learned in sc

from decimal import Decimal as D
print(D('0.1') + D('0.2'))
```

```
0.3
```

In [49]:
```
threshold = 0.00001
x = 1.1 + 2.2
print(x)
print(x - 3.3)
abs(x - 3.3) < threshold
```

```
3.3000000000000003
4.440892098500626e-16
```

Out[49]:
```
True
```

Formatting output using String modulo operator(%)

In [50]:
```
1 val = 'Hi'
2 name = 'Joseph'
3
4 print("%s, my name is %s." % (val,name))
5
6 n = 50
7 print("The number is: %d and %f" % (n, 120.20))
```

```
Hi, my name is Joseph.
The number is: 50 and 120.200000
```

The format() Method

- added in Python 2.6
- The brackets and characters within them are called format fields

In [51]:
```
1 # using format() method
2
3 print('I am a {}'.format('Programmer'))
4 print('The number is {}'.format(n))
```

```
I am a Programmer
The number is 50
```

In [52]:
```
1 # using format() method and refering to a position of the object
2 print('{0} and {1}'.format('Hello', 'World'))
3
```

```
Hello and World
```

In [53]:
```
1 print('{1}  and {0}'.format('Hello', 'World'))
```

```
World  and Hello
```

Formatted String literals

To create an f-string, prefix the string with the letter " f ". The string itself can be formatted in much the same way that you would with str.format().

In [54]:
```
# the above formatting can also be done by using f-Strings. Works only with python 3.6 or above.
print(f"{val}, my name is {name}.")
```

```
Hi, my name is Joseph.
```

Operators

https://www.w3schools.com/python/python_operators.asp (https://www.w3schools.com/python/python_operators.asp)

Booleans

https://www.w3schools.com/python/python_booleans.asp (https://www.w3schools.com/python/python_booleans.asp)

if-else

In [55]:
```
# C language:
# if
#     this
# else
#     this
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

```
b is greater than a
```

In [56]:
```
a = 30
b = 200

# a = 300
# b = 200

# a = 30
# b = 30

if b > a:
    print("b is greater than a")
elif b<a:
    print("b is less than a")
else:
    print("b is equal to a")
```

```
b is greater than a
```

In [57]:
```
# Short-hand

# In c: condition ? value_if_true : value_if_false
# Eg.: c = (a < b) ? a : b;

a = 30
b = 200
if b>a: print("Yes")

# a = 20
# b = 30
# print("A") if a > b else print("B")

# This is also called Ternany Operators or Conditional Expressions.
```

```
Yes
```

In [58]:
```
# Nested if-else

x = 41

if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

```
Above ten,
and also above 20!
```

Pass Statement

if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

In [59]:
```
a = 33
b = 200

if b > a:

else:
    print("Test")
```

```
  File "<ipython-input-59-e74a16484162>", line 6
    else:
    ^
IndentationError: expected an indented block
```

In [60]:
```
a = 33
b = 200

if b > a:
    pass
else:
    print("Test")
```

While loop

In [61]:
```
# C language:
# i=1;
# while(i<6)
# {
#     printf(""%d",i));
#     i++;
# }
i = 1
while i < 6:
  print(i)
  i += 1
```

```
1
2
3
4
5
```

else in While loop

In [62]:
```
# With the else statement we can run a block of code once when the condition no longer is true:

i=1
while i < 6:
    print(i)
    i += 1
else:
    print("i is no longer less than 6")
```

```
1
2
3
4
5
i is no longer less than 6
```

For Loop

In [63]:
```
# C language:
# for(i=0;i<10;i++)
# {
#     xyz;
# }
```

In [64]:
```
# Looping Through a String

for x in "Universe":
    print(x)
```

```
U
n
i
v
e
r
s
e
```

In [65]:
```
# Looping through a list

fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

```
apple
banana
cherry
```

break in for loop

In [66]:
```

for x in "Universe":
    print(x)
    if x == "v":
        break
```

```
U
n
i
v
```

continue in for loop

In [67]:
```
for x in "Universe":
    print(x)
    if x == "v":
        continue

# for x in "Universe":
#     if x == "v":
#         continue
#     print(x)
```

```
U
n
i
v
e
r
s
e
```

The range() function

To loop through a set of code a specified number of times, we can use the range() function, The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

In [68]:
```
range?
```

In [69]:
```
for x in range(10):
    print(x)
```

```
0
1
2
3
4
5
6
7
8
9
```

In [70]:
```
for x in range(3, 10):
    print(x)
```

```
3
4
5
6
7
8
9
```

In [71]:
```
for x in range(3, 30, 2):
    print(x)
```

```
3
5
7
9
11
13
15
17
19
21
23
25
27
29
```

else in For loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

In [72]:
```
for x in range(6):
    print(x)
else:
    print("Finished!")
```

```
0
1
2
3
4
5
Finished!
```

In [73]:
```
# Nested for loop

adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
    for y in fruits:
        print(x, y)
```

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

The pass statement

for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

In [74]:
```
1 for x in [0, 1, 2]:
2
```

```
  File "<ipython-input-74-6d1e5fbf3f8c>", line 2

    ^
SyntaxError: unexpected EOF while parsing
```

In [75]:
```
1 for x in [0, 1, 2]:
2     pass
```

Strings

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters. However, Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string.

In [76]:
```
1 # Assign String to a Variable
2 x = "Hello"
3 print(x)
```

```
Hello
```

In [77]:
```
1 a = "Hello World!"
```

In [78]:
```
1 # Strings as arrays
2 print(a[1])
3 print(a[6])
```

```
e
W
```

In [79]:
```
# Looping Through a String
for i in a:
    print(i)
```

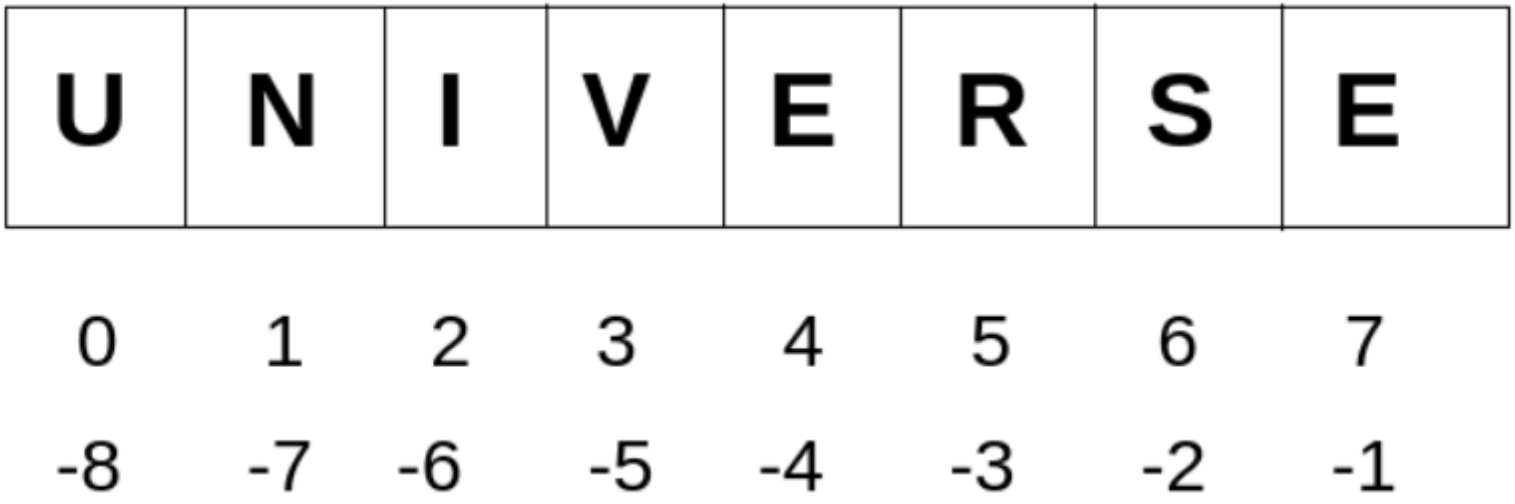
```
H
e
l
l
o

W
o
r
l
d
!
```

In [80]:
```
# String Length
print(len(a))
```

```
12
```

Misc Functions: https://www.w3schools.com/python/python_strings.asp (https://www.w3schools.com/python/python_strings.asp)

String Slicing



In [81]:
```
1 print(a[2:5])
```

```
llo
```

In [82]:
```
1 print(a[:5])
```

```
Hello
```

In [83]:
```
1 print(a[2:])
```

```
llo World!
```

Negative Indexing

In [84]:
```
1 print(a[-5:-2])
```

```
orl
```

In [85]:
```
# Upper Case
print(a.upper())
```

```
HELLO WORLD!
```

In [86]:
```
# Lower Case
print(a.lower())
```

```
hello world!
```

In [87]:
```
# Remove Whitespace: strip()
a = "   Hello, World!     "
print(a.strip())
```

```
Hello, World!
```

In [88]:
```
# Replace a string: returns a list
a = "Hello, World!"
print(a.split(","))
```

```
['Hello', ' World!']
```

Other methods: https://www.w3schools.com/python/python_ref_string.asp (https://www.w3schools.com/python/python_ref_string.asp)

String Concatenation

In [89]:
```
a = "Hello"
b = "World"
c = a + b
print(c)
```

```
HelloWorld
```

In [90]:
```
a = "Hello"
b = "World"
c = a + " " + b
print(c)
```

```
Hello World
```

String Formatting: Done Above

Escape Character

To insert characters that are illegal in a string, use an escape character. An escape character is a backslash \ followed by the character you want to insert. An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

In [91]:
```
txt = "Hello, My name is "Ram" and I am a programmer."
```

```
  File "<ipython-input-91-937e8c6def83>", line 1
    txt = "Hello, My name is "Ram" and I am a programmer."
                             ^
SyntaxError: invalid syntax
```

In [92]:
```
txt = "Hello, My name is \"Ram\" and I am a programmer."
print(txt)
```

```
Hello, My name is "Ram" and I am a programmer.
```

Other escape characters: https://www.w3schools.com/python/python_strings_escape.asp (https://www.w3schools.com/python/python_strings_escape.asp)

is vs ==

is is used for identity comparison, while == is used for equality comparison.

If you care about equality (the two strings should contain the same characters), then 'is' operator is simply wrong and you should be using == instead.

Difference Between == and is: is checks whether the variables are referring to the same object in memory, while == checks whether the variables have the same value.

The reason 'is' works interactively is that (most) string literals are interned by default.

Without interning, checking that two different strings are equal involves examining every character of both strings. This is slow for several reasons:

- it is inherently O(n) in the length of the strings
- it typically requires reads from several regions of memory, which take time
- the reads fills up the processor cache, meaning there is less cache available for other needs.

So, when you have two string literals (words that are literally typed into your program source code, surrounded by quotation marks) in your program that have the same value, the Python compiler will automatically intern the strings, making them both stored at the same memory location. (Note that this doesn't always happen, and the rules for when this happens are quite convoluted, so please don't rely on this behavior in production code!). This means that, when we create two strings with the same value - instead of allocating memory for both of them, only one string is actually committed to memory. The other one just points to that same memory location.

Since in your interactive session both strings are actually **stored in the same memory location, they have the same identity**, so the is operator works as expected.

But if you construct a string by some other method (even if that string contains exactly the same characters), then the string may be equal, but it is not the same string -- that is, it has a different identity, because it is stored in a different place in memory.

Let us understand this more clearly with the help of an example:

When string a is created, the compiler checks if Hello World is present in interned memory. Since it is the first occurrence of this string value, Python creates an object and caches this string in memory and points a to this reference.

When b is created, Hello World is found by the compiler in the interned memory so instead of creating another string, b simply points to the previously allocated memory.

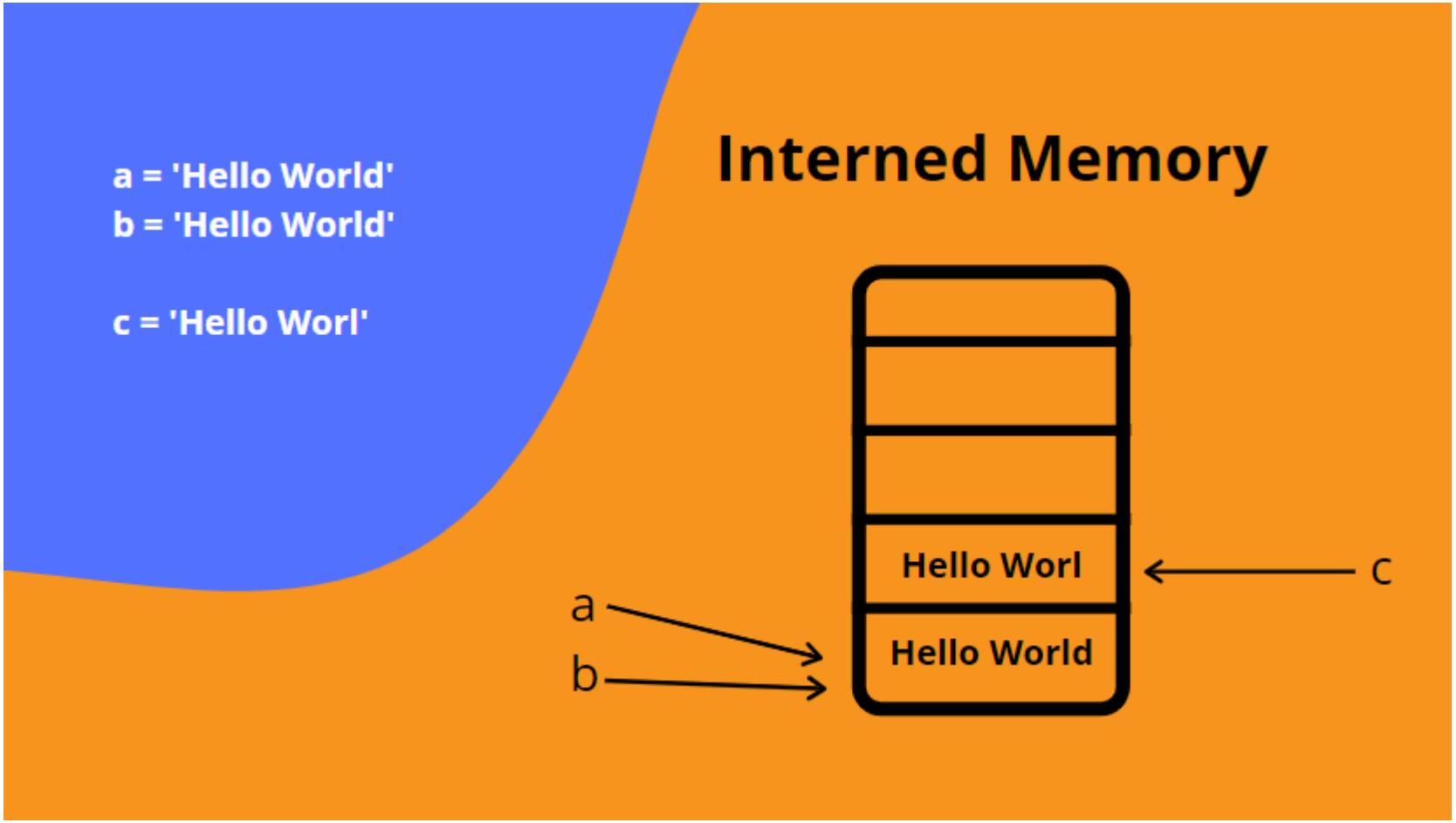
a is b and a == b in this case.

Finally, when we create the string c = 'Hello Worl', the compiler instantiates another object in interned memory because it could not find the same object for reference.

When we compare a and c+'d', the latter is evaluated to 'Hello World. However', since Python doesn't do interning during runtime, a new object is created instead. Thus, since no interning was done, these two aren't the same object and is returns False.

In contrast to the is operator, the == operator compares the values of the strings after computing runtime expressions - Hello World == Hello World.

At that time, a and c+'d' are the same, value-wise, so this returns True.



In [93]:
```
a = "Hello"
b = "Hello"
print(id(a))
print(id(b))

print(a is b)
print(a==b)

print(a.__eq__(b))
```

```
140340341071344
140340341071344
True
True
True
```

In [94]:
```
c = "Hell"
print(c+'o')
print(id(a))
print(id(c+'o'))

print(a is (c+'o'))
print(a==(c+'o'))
```

```
Hello
140340341071344
140340341956528
False
True
```

Additional links for string interning:

https://stackoverflow.com/questions/3588776/how-is-eq-handled-in-python-and-in-what-order (https://stackoverflow.com/questions/3588776/how-is-eq-handled-in-python-and-in-what-order)

https://www.tutorialsteacher.com/python/magic-methods-in-python (https://www.tutorialsteacher.com/python/magic-methods-in-python)

https://stackoverflow.com/questions/2988017/string-comparison-in-python-is-vs (https://stackoverflow.com/questions/2988017/string-comparison-in-python-is-vs)

https://stackabuse.com/guide-to-string-interning-in-python/ (https://stackabuse.com/guide-to-string-interning-in-python/)