ADVANCED PYTHON PROGRAMMING

Python Overview and Programming Basic

Peter Lo

Who am I?

- Senior Business Analyst
- Over 15 years in consulting and I.T. experiences
- Over 20 years in University teaching experiences

- **Email:** Peter@Peter-Lo.com
- **Facebook:** http://www.facebook.com/PeterLo111

掃描上面的QR Code，加我WeChat。

Course Outline

Chapter	Topic
1	Computer Programming and Python Fundamentals
2	Control Flow
3	Data Collections
4	Functions
5	PCEP Preparation
6	Modules and Packages
7	Regular Expression
8	File Manipulation
9	String and Exceptions
10	Data Manipulation with Pandas
11	Array Processing by NumPy
12	Visualization with Matplotlib
13	Web Services
14	Advanced Data Analysis

Where can you find the material?

- Workshop Notes
  - http://www.Peter-Lo.com/Teaching/AMA-Python/
- Python Official Page
  - http://www.python.org
- The Python Standard Library
  - https://docs.python.org/3/library/

Overview

Introduction to Programming Language and Python

What is Programming?

- A programming language is a computer language engineered to create a standard form of commands.
- These commands can be interpreted into a code understood by a machine.
- Programs are created through programming languages to control the behavior and output of a machine through accurate algorithms, similar to the human communication process.

How Computer Understanding Program?

- Computers do not understand human languages, so programs must be written in a language a computer can use.
- There are hundreds of programming languages, and they were developed to make the programming process easier for people.
- All programs must be converted into a language the computer can understand.

Level of Programming Languages

High-level program

```
class Triangle {
  ...
  float surface()
    return b*h/2;
  }
```

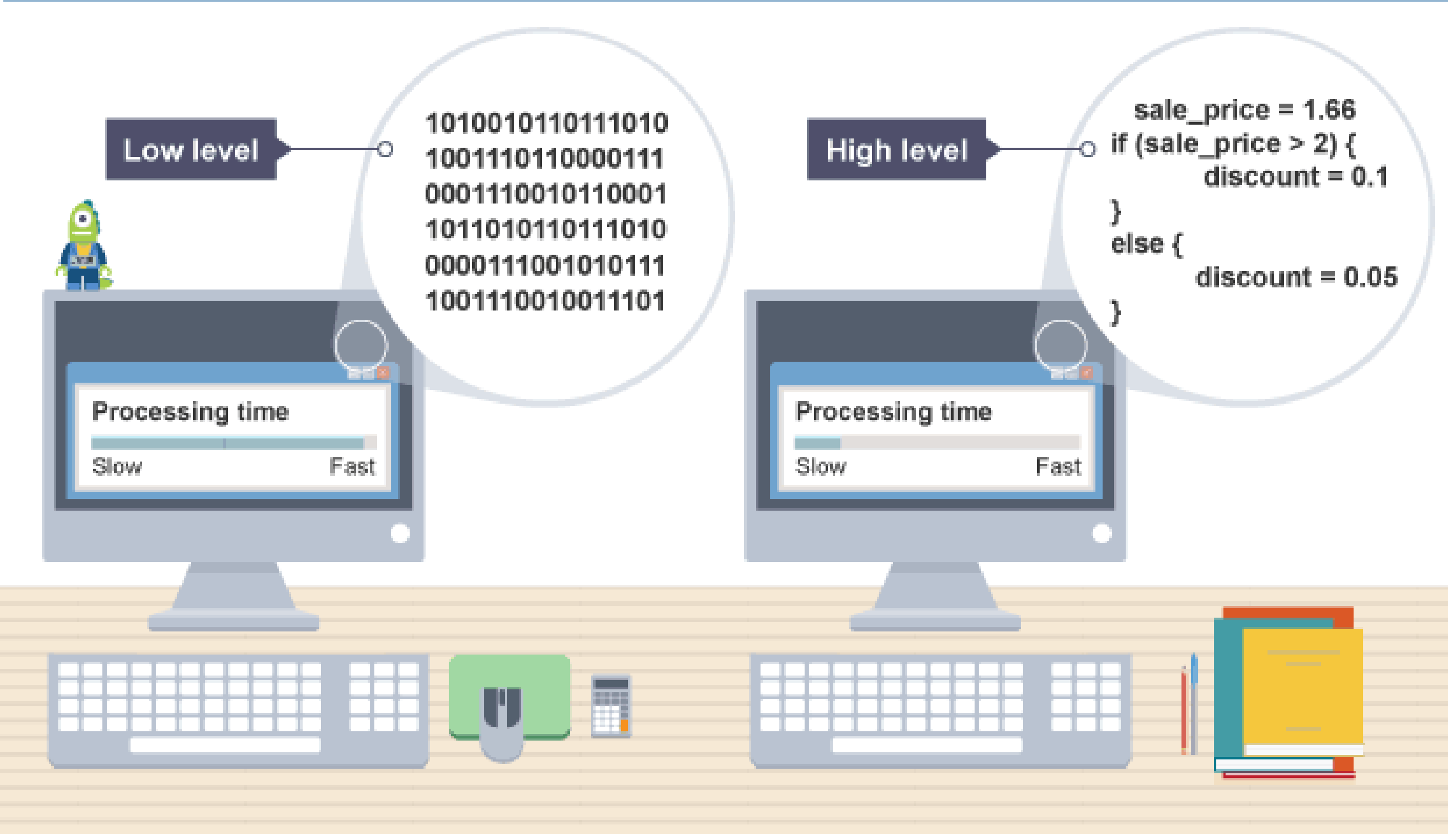
Low-level program

```
LOAD r1,b
LOAD r2,h
MUL r1,r2
DIV r1,#2
RET
```

Executable Machine code

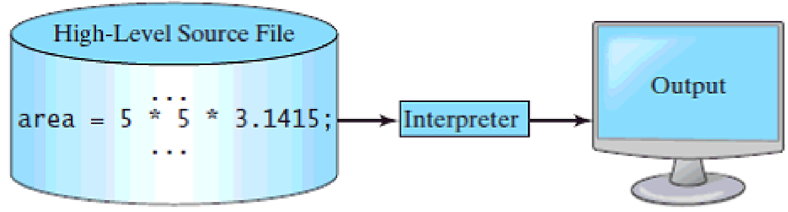
```
0001001001000101
0010010011101100
10101101001...
```

Low-Level vs. High-Level Languages

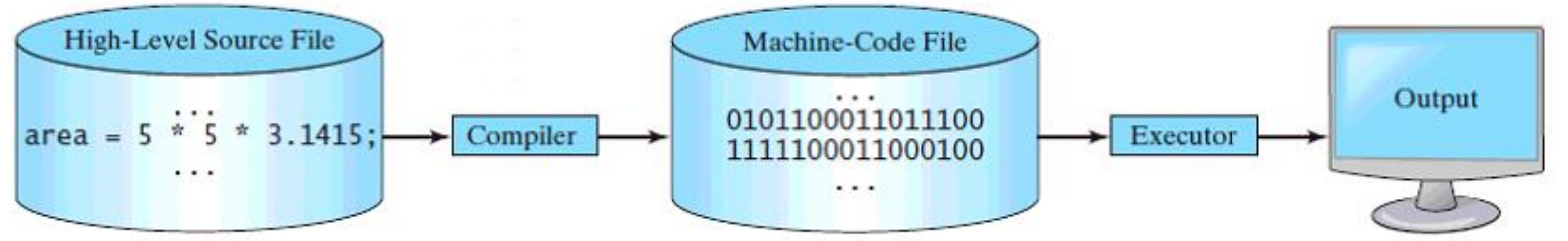


Interpreter and Compiler

- An **Interpreter** reads one statement from the source code, translates it to the machine code or virtual machine code, and then executes it right away



- A **Compiler** translates the entire source code into a machine-code file, and the machine code file is then executed



Interpreter vs. Compiler

Compilation

- The execution of the translated code is usually faster;
- Only the user has to have the compiler - the end-user may use the code without it;
- The translated code is stored using machine language - as it is very hard to understand it, your own inventions and programming tricks are likely to remain your secret.

Interpretation

- You can run the code as soon as you complete it - there are no additional phases of translation;
- The code is stored using programming language, not machine language - this means that it can be run on computers using different machine languages; you don't compile your code separately for each different architecture.

Top 10 Programming Languages

- According to IEEE Spectrum's interactive ranking, Python is the top programming language of 2021, followed by Java and C.

Rank	Language	Type	Score
1	Python		100.0
2	Java		95.4
3	C		94.7
4	C++		92.4
5	JavaScript		88.1
6	C#		82.4
7	R		81.7
8	Go		77.7
9	HTML		75.4
10	Swift		70.4

What is Python?

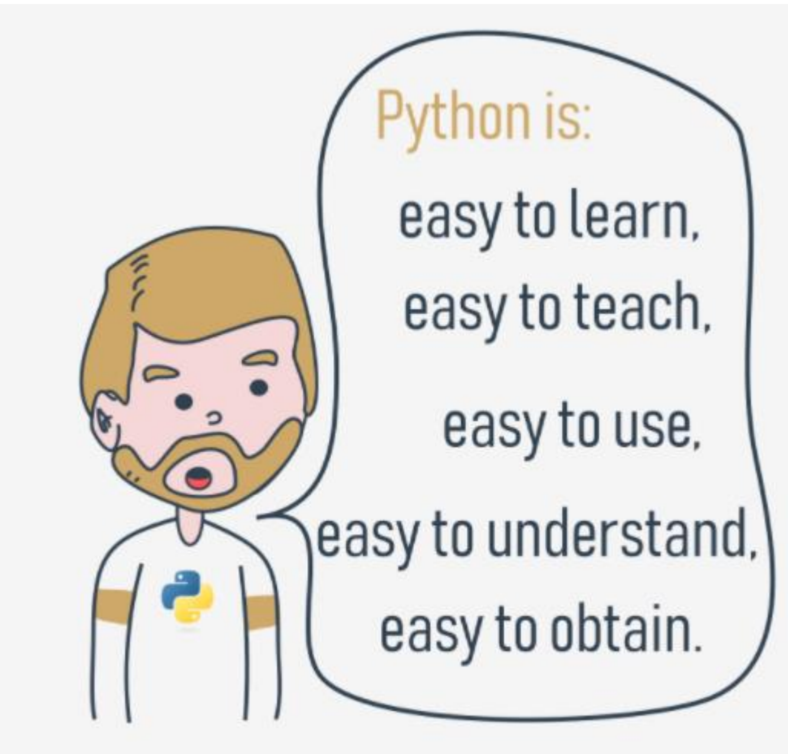
- Python is a widely-used, interpreted, object-oriented, and high-level programming language with dynamic semantics, used for general-purpose programming.
- Python was created by Guido van Rossum, and first released on 20 February 1991.
- To download Python, please visit https://www.python.org

Python Goals

- In 1999, Guido van Rossum defined his goals for Python:
  - An easy and intuitive language just as powerful as those of the major competitors;
  - Open source, so anyone can contribute to its development;
  - Code that is as understandable as plain English;
  - Suitable for everyday tasks, allowing for short development times.

What makes Python Special?

- Works on different platforms
- Has a simple syntax similar to the English language
- Has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Can be treated in a procedural way, an object-orientated way or a functional way.



What can Python do?

- Used on a server to create web applications
- Used alongside software to create workflows
- Connect to database systems. It can also read and modify files
- Handle big data and perform complex mathematics
- Create rapid prototyping, or for production-ready software development

Why not Python?

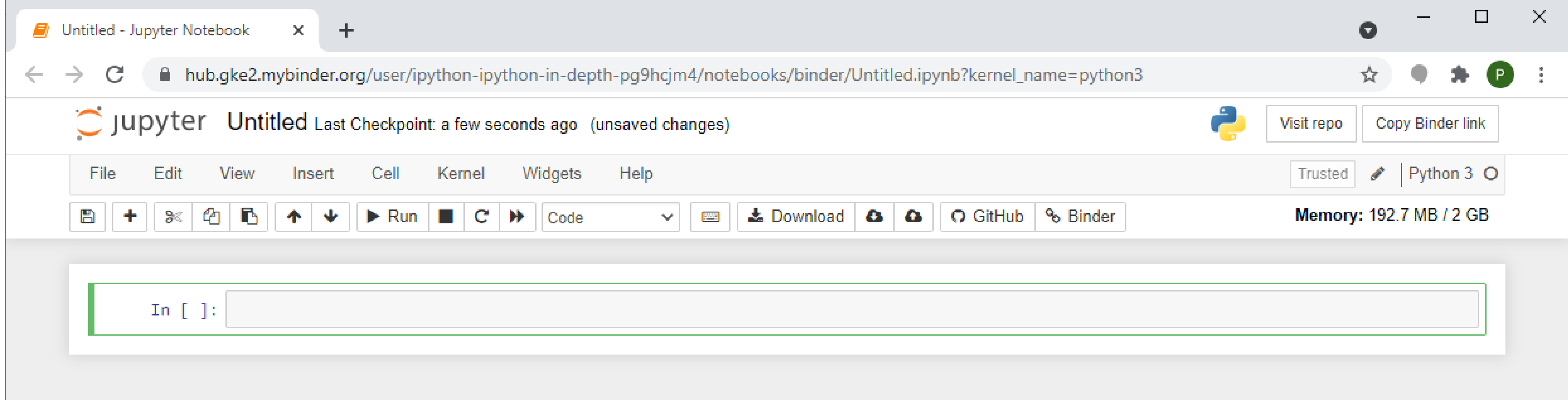
- Despite Python's growing popularity, there are still some niches where Python is absent, or is rarely seen:
  - **Low-Level Programming:** if you want to implement an extremely effective driver or graphical engine, you wouldn't use Python;
  - **Applications for Mobile Devices:** although this territory is still waiting to be conquered by Python, it will most likely happen someday.

Development Tools

Google Colab Notebook

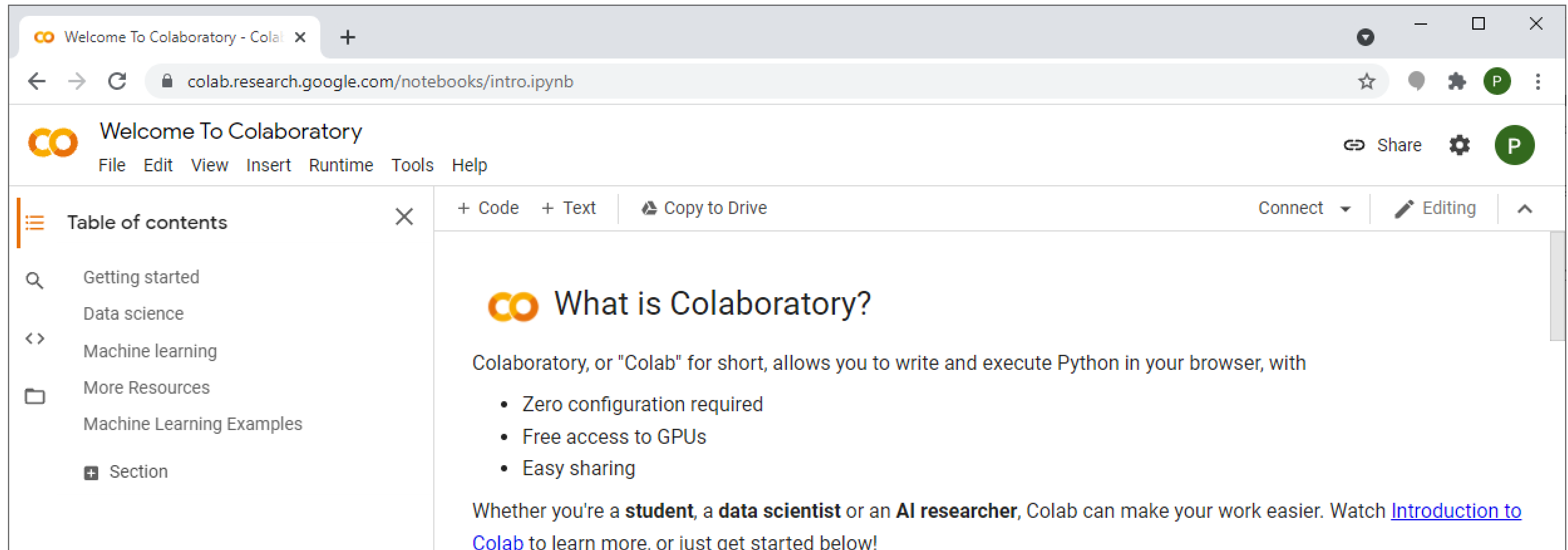
Jupyter Notebook

- Jupyter Notebook (https://jupyter.org) is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text.
- It provides an environment, where you can document your code, run it, look at the outcome, visualize data and see the results without leaving the environment.



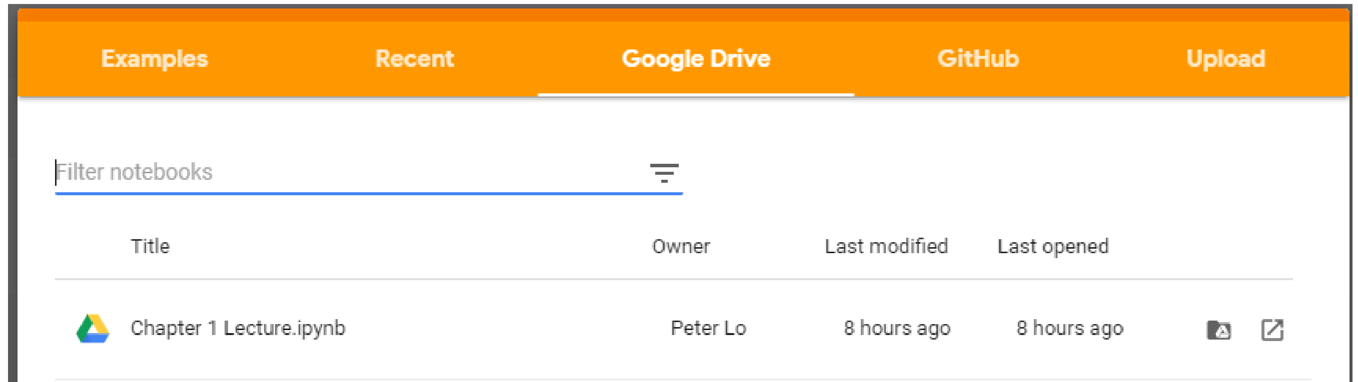
Google Colab

- Google Colab (https://colab.research.google.com) is a product from Google Research. Colab allows anybody to write and execute arbitrary python code through the browser, and is especially well suited to machine learning, data analysis and education.



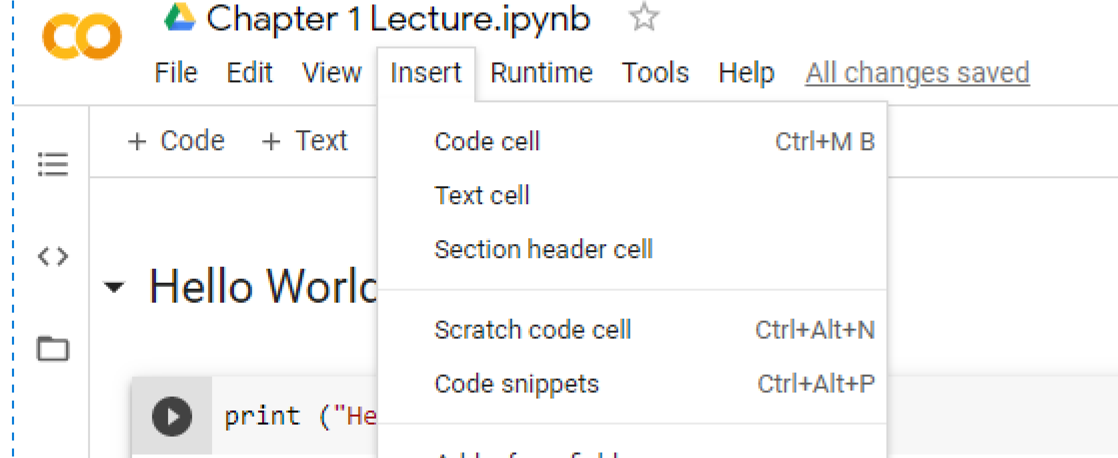
Where will Colab Notebook Store?

- Colab notebooks allow you to combine executable code and rich text in a single document, along with images, HTML, LaTeX and more.
- When you create your own Colab notebooks, they are stored in your Google Drive account.
- You can easily share your Colab notebooks with co-workers or friends, allowing them to comment on your notebooks or even edit them.



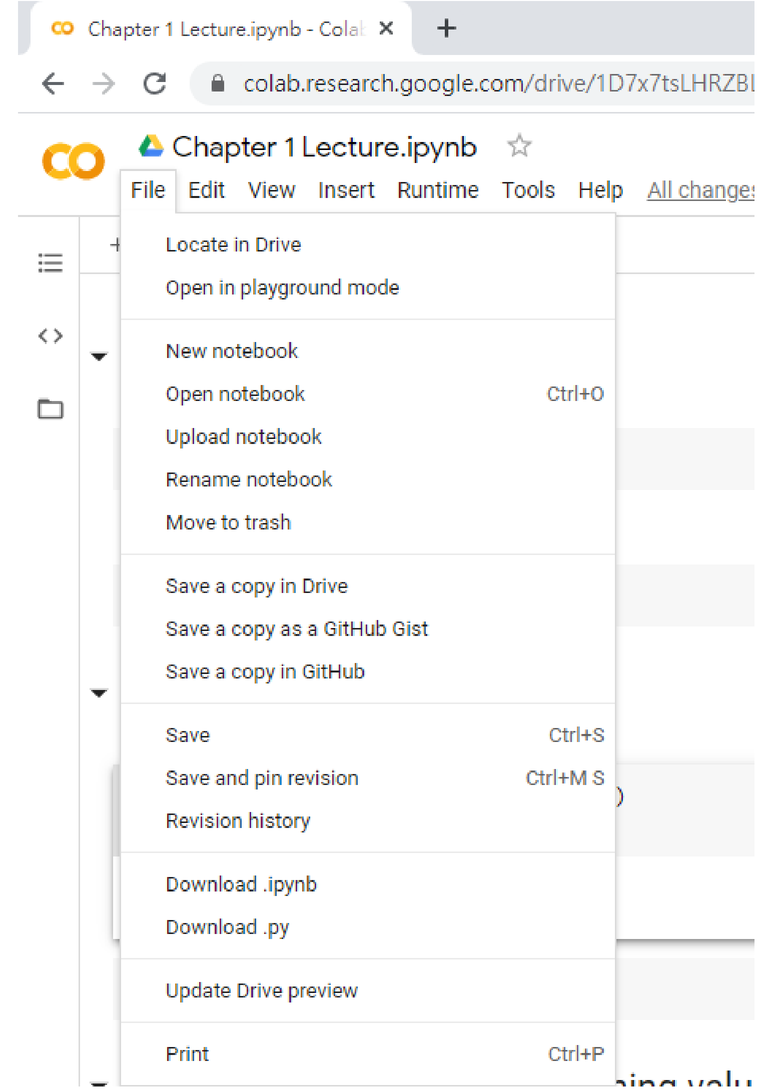
Cell Type

Cell Type	Description
Code	It is where you type your code
Text	This is where you type your text. You can add your conclusions after running a code, add comments, etc.
Header	This is where you add Headings to separate sections and make your notebook look tidy and neat. This has now been converted into the Markdown option itself. Add a '##' to ensure that whatever you type after that will be taken as a heading



Saving and Sharing Notebook

- You can download your Notebook in any of the options provided.
- The most commonly used is download as .ipynb file so other person can replicate your code on their machine



Hello World

- There is a long-standing custom in the field of computer programming that the first code written in a newly installed language is a short program that simply displays the string “Hello World” to the console.
- In this program, we have used the built-in **print()** function to print the string “*Hello World*” on screen.

Hello World

```
[1] print ("Hello World")
```

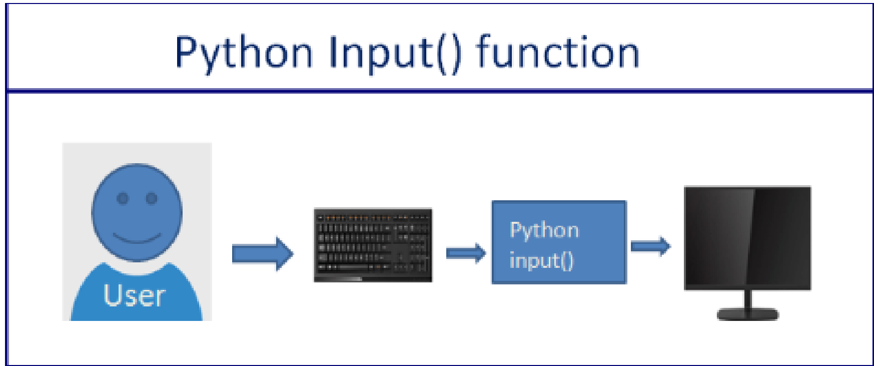
```
Hello World
```

Obtain User Input

- Python has a built-in function **input()** to accept user input.
- This function reads a line from an input and converts into a string and returns it, then you can use this input string in your python program

```
name = input ("What is your name?")
print ("Your name is ", name)
```

```
What is your name?
```



Commenting Code

- In general, commenting is describing your code for developers.
- In conjunction with well-written code, comments help to guide the reader to better understand your code and its purpose and design
- Comments are created in Python using the pound sign (#) and should be brief statements no longer than a few sentences.

Python

```python
def hello_world():
    # A simple comment preceding a simple print statement
    print("Hello World")
```

Programming Basic

Variables and Constants

Variables

- Variable is a named location used to store data in the memory in most of the programming languages.
- Each variable must have a unique name called identifier.
- It is helpful to think of variables as container that hold data which can be changed later throughout programming.
  - Non technically, you can suppose variable as a bag to store books in it and those books can be replaced at anytime.

Variable in Python

- Declaring Variables
  - Variables do not need declaration to reserve memory space. The "variable declaration" or "variable initialization" happens automatically when we assign a value to a variable.

Every variable has a name - to identify it from other variables

- Assigning value to a Variable
  - You can use the assignment operator **=** to assign the value to a variable.

Python

```
>>> n = 300
```

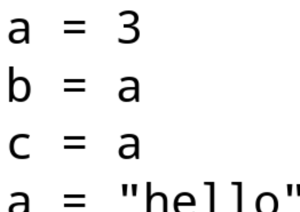
variable assignment

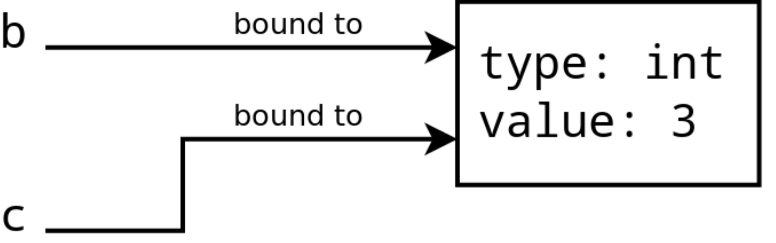
a = 5

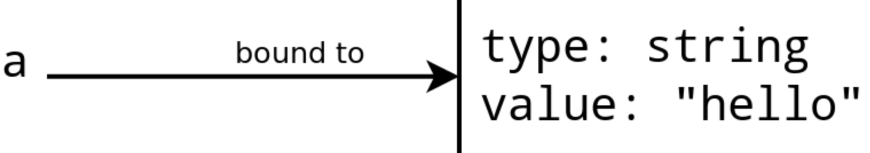
Re-declare Variable

- In Python, you can re-declare the variable even after you have declared it once.

Executed Code:
Variable Assignment **Variables** **Values in Memory**







Variable Names

- A variable can have a short name (*like x and y*) or a more descriptive name (*age, CarName, Total_Volume*).
- Rules for Python variables name:
  - Must start with a letter or underscore
  - Cannot start with a number
  - Can only contain alpha-numeric characters and underscores (*A-z, 0-9, and _*)
  - Case-sensitive (*age, Age and AGE are three different variables*)

Reserved Words

- Reserved words in Python and used to perform an internal operation. You can not use reserved words as variable names or identifiers

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

Variable Naming Convention

- The most commonly used methods of constructing a multi-word variable name are:
  - **Camel Case:** Second and subsequent words are capitalized, to make word boundaries easier to see.
    - Example: numberOfStudent
  - **Pascal Case:** Identical to Camel Case, except the first word is also capitalized.
    - Example: NumberOfStudent
  - **Snake Case:** Words are separated by underscores.
    - Example: number_of_student

Object Identity

- In Python, every object that is created is given a number that uniquely identifies it.
- It is guaranteed that no two objects will have the same identifier during any period in which their lifetimes overlap.
- Once an object’s reference count drops to zero and it is garbage collected, then its identifying number becomes available and may be used again.

Object Identity Example

- The built-in Python function **id()** returns an object's integer identifier. Using the **id()** function, you can verify that two variables indeed point to the same object:

Python

```python
>>> n = 300
>>> m = n
>>> id(n)
60127840
>>> id(m)
60127840

>>> m = 400
>>> id(m)
60127872
```

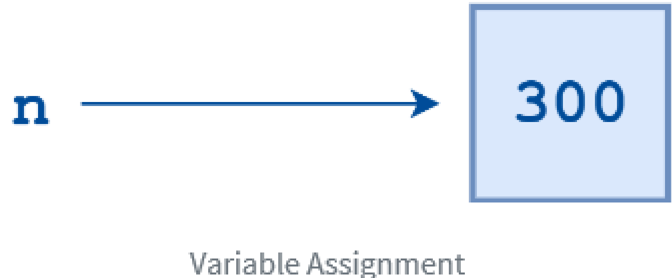
Object Identity Example (Cont.)

- Statement

Python

```
>>> n = 300
```

- This assignment creates an integer object with the value 300 and assigns the variable n to point to that object



Variable Assignment

Object Identity Example (Cont.)

- Statement

```python
>>> m = n
```

- Python does not create another object. It simply creates a new symbolic name or reference, m, which points to the same object that n points to.

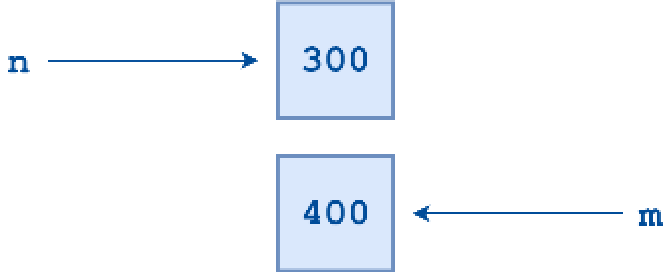
Multiple References to a Single Object

Object Identity Example (Cont.)

- Statement

Python
```
>>> m = 400
```

- Python creates a new integer object with the value 400, and m becomes a reference to it.

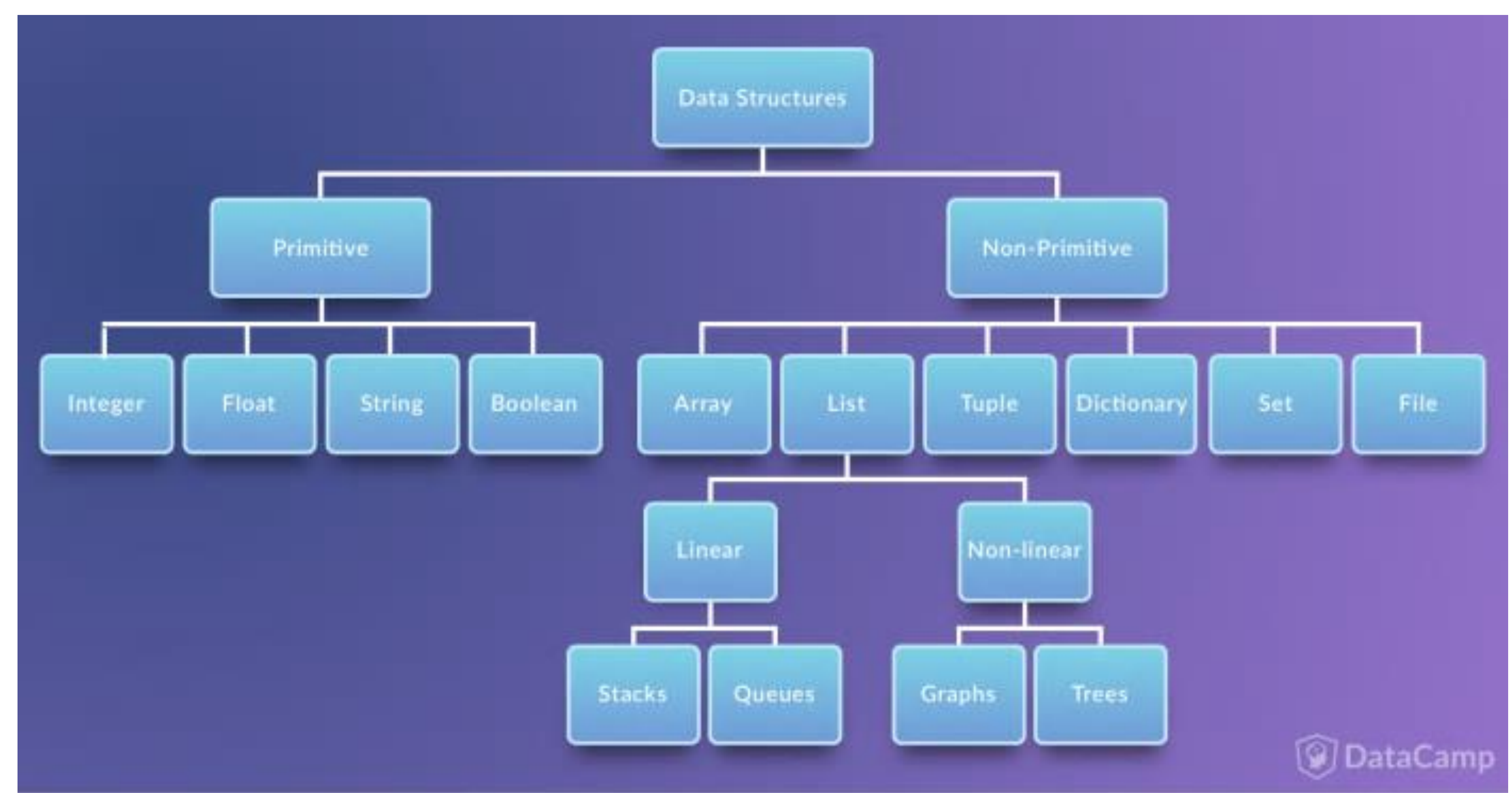


References to Separate Objects

Basic Data Type

Common Build-in Data Types and Operators in Python

Basic Data Types in Python



Integer

- In Python 3, there is effectively no limit to how long an integer value can be.
- Of course, it is constrained by the amount of memory your system has, as are all things, but beyond that an integer can be as long as you need it to be:

Python

```
>>> print(10)
10
```

Python

```
>>> 10
10
```

Python

```
>>> type(10)
<class 'int'>
```

Hexadecimal

- The following strings can be prepended to an integer value to indicate a base other than 10:

Prefix	Interpretation	Base
`0b` (zero + lowercase letter `'b'`) `0B` (zero + uppercase letter `'B'`)	Binary	2
`0o` (zero + lowercase letter `'o'`) `0O` (zero + uppercase letter `'O'`)	Octal	8
`0x` (zero + lowercase letter `'x'`) `0X` (zero + uppercase letter `'X'`)	Hexadecimal	16

Python

```
>>> print(0o10)
8

>>> print(0x10)
16

>>> print(0b10)
2
```

ASCII Table

<table>
<tr><th>Char</th><th>Dec</th><th>Oct</th><th>Hex</th><th>Char</th><th>Dec</th><th>Oct</th><th>Hex</th><th>Char</th><th>Dec</th><th>Oct</th><th>Hex</th></tr>
<tr><td>(sp)</td><td>32</td><td>0040</td><td>0x20</td><td>@</td><td>64</td><td>0100</td><td>0x40</td><td>`</td><td>96</td><td>0140</td><td>0x60</td></tr>
<tr><td>!</td><td>33</td><td>0041</td><td>0x21</td><td>A</td><td>65</td><td>0101</td><td>0x41</td><td>a</td><td>97</td><td>0141</td><td>0x61</td></tr>
<tr><td>"</td><td>34</td><td>0042</td><td>0x22</td><td>B</td><td>66</td><td>0102</td><td>0x42</td><td>b</td><td>98</td><td>0142</td><td>0x62</td></tr>
<tr><td>#</td><td>35</td><td>0043</td><td>0x23</td><td>C</td><td>67</td><td>0103</td><td>0x43</td><td>c</td><td>99</td><td>0143</td><td>0x63</td></tr>
<tr><td>$</td><td>36</td><td>0044</td><td>0x24</td><td>D</td><td>68</td><td>0104</td><td>0x44</td><td>d</td><td>100</td><td>0144</td><td>0x64</td></tr>
<tr><td>%</td><td>37</td><td>0045</td><td>0x25</td><td>E</td><td>69</td><td>0105</td><td>0x45</td><td>e</td><td>101</td><td>0145</td><td>0x65</td></tr>
<tr><td>&</td><td>38</td><td>0046</td><td>0x26</td><td>F</td><td>70</td><td>0106</td><td>0x46</td><td>f</td><td>102</td><td>0146</td><td>0x66</td></tr>
<tr><td>'</td><td>39</td><td>0047</td><td>0x27</td><td>G</td><td>71</td><td>0107</td><td>0x47</td><td>g</td><td>103</td><td>0147</td><td>0x67</td></tr>
<tr><td>(</td><td>40</td><td>0050</td><td>0x28</td><td>H</td><td>72</td><td>0110</td><td>0x48</td><td>h</td><td>104</td><td>0150</td><td>0x68</td></tr>
<tr><td>)</td><td>41</td><td>0051</td><td>0x29</td><td>I</td><td>73</td><td>0111</td><td>0x49</td><td>i</td><td>105</td><td>0151</td><td>0x69</td></tr>
<tr><td>*</td><td>42</td><td>0052</td><td>0x2a</td><td>J</td><td>74</td><td>0112</td><td>0x4a</td><td>j</td><td>106</td><td>0152</td><td>0x6a</td></tr>
<tr><td>+</td><td>43</td><td>0053</td><td>0x2b</td><td>K</td><td>75</td><td>0113</td><td>0x4b</td><td>k</td><td>107</td><td>0153</td><td>0x6b</td></tr>
<tr><td>,</td><td>44</td><td>0054</td><td>0x2c</td><td>L</td><td>76</td><td>0114</td><td>0x4c</td><td>l</td><td>108</td><td>0154</td><td>0x6c</td></tr>
<tr><td>-</td><td>45</td><td>0055</td><td>0x2d</td><td>M</td><td>77</td><td>0115</td><td>0x4d</td><td>m</td><td>109</td><td>0155</td><td>0x6d</td></tr>
<tr><td>.</td><td>46</td><td>0056</td><td>0x2e</td><td>N</td><td>78</td><td>0116</td><td>0x4e</td><td>n</td><td>110</td><td>0156</td><td>0x6e</td></tr>
<tr><td>/</td><td>47</td><td>0057</td><td>0x2f</td><td>O</td><td>79</td><td>0117</td><td>0x4f</td><td>o</td><td>111</td><td>0157</td><td>0x6f</td></tr>
<tr><td>0</td><td>48</td><td>0060</td><td>0x30</td><td>P</td><td>80</td><td>0120</td><td>0x50</td><td>p</td><td>112</td><td>0160</td><td>0x70</td></tr>
<tr><td>1</td><td>49</td><td>0061</td><td>0x31</td><td>Q</td><td>81</td><td>0121</td><td>0x51</td><td>q</td><td>113</td><td>0161</td><td>0x71</td></tr>
<tr><td>2</td><td>50</td><td>0062</td><td>0x32</td><td>R</td><td>82</td><td>0122</td><td>0x52</td><td>r</td><td>114</td><td>0162</td><td>0x72</td></tr>
<tr><td>3</td><td>51</td><td>0063</td><td>0x33</td><td>S</td><td>83</td><td>0123</td><td>0x53</td><td>s</td><td>115</td><td>0163</td><td>0x73</td></tr>
<tr><td>4</td><td>52</td><td>0064</td><td>0x34</td><td>T</td><td>84</td><td>0124</td><td>0x54</td><td>t</td><td>116</td><td>0164</td><td>0x74</td></tr>
<tr><td>5</td><td>53</td><td>0065</td><td>0x35</td><td>U</td><td>85</td><td>0125</td><td>0x55</td><td>u</td><td>117</td><td>0165</td><td>0x75</td></tr>
<tr><td>6</td><td>54</td><td>0066</td><td>0x36</td><td>V</td><td>86</td><td>0126</td><td>0x56</td><td>v</td><td>118</td><td>0166</td><td>0x76</td></tr>
<tr><td>7</td><td>55</td><td>0067</td><td>0x37</td><td>W</td><td>87</td><td>0127</td><td>0x57</td><td>w</td><td>119</td><td>0167</td><td>0x77</td></tr>
<tr><td>8</td><td>56</td><td>0070</td><td>0x38</td><td>X</td><td>88</td><td>0130</td><td>0x58</td><td>x</td><td>120</td><td>0170</td><td>0x78</td></tr>
<tr><td>9</td><td>57</td><td>0071</td><td>0x39</td><td>Y</td><td>89</td><td>0131</td><td>0x59</td><td>y</td><td>121</td><td>0171</td><td>0x79</td></tr>
<tr><td>:</td><td>58</td><td>0072</td><td>0x3a</td><td>Z</td><td>90</td><td>0132</td><td>0x5a</td><td>z</td><td>122</td><td>0172</td><td>0x7a</td></tr>
<tr><td>;</td><td>59</td><td>0073</td><td>0x3b</td><td>[</td><td>91</td><td>0133</td><td>0x5b</td><td>{</td><td>123</td><td>0173</td><td>0x7b</td></tr>
<tr><td><</td><td>60</td><td>0074</td><td>0x3c</td><td>\</td><td>92</td><td>0134</td><td>0x5c</td><td>|</td><td>124</td><td>0174</td><td>0x7c</td></tr>
<tr><td>=</td><td>61</td><td>0075</td><td>0x3d</td><td>]</td><td>93</td><td>0135</td><td>0x5d</td><td>}</td><td>125</td><td>0175</td><td>0x7d</td></tr>
<tr><td>></td><td>62</td><td>0076</td><td>0x3e</td><td>^</td><td>94</td><td>0136</td><td>0x5e</td><td>~</td><td>126</td><td>0176</td><td>0x7e</td></tr>
<tr><td>?</td><td>63</td><td>0077</td><td>0x3f</td><td>_</td><td>95</td><td>0137</td><td>0x5f</td><td></td><td></td><td></td><td></td></tr>
</table>

Floating-Point Numbers

- The float type in Python designates a floating-point number. float values are specified with a decimal point.
- Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation:

Python

```
>>> 4.2
4.2
>>> type(4.2)
<class 'float'>
>>> 4.
4.0
>>> .2
0.2

>>> .4e7
4000000.0
>>> type(.4e7)
<class 'float'>
>>> 4.2e-4
0.00042
```

Strings

- Strings are sequences of character data. The string type in Python is called str.
- String literals may be delimited using either single or double quotes. All the characters between the opening delimiter and matching closing delimiter are part of the string:

```
>>> print("I am a string.")
I am a string.
>>> type("I am a string.")
<class 'str'>

>>> print('I am too.')
I am too.
>>> type('I am too.')
<class 'str'>
```

Interpolating Variables into a String

- In Python version 3.6, a new string formatting mechanism was introduced. This feature is formally named the Formatted String Literal, but is more usually referred to by its nickname **f-string**

Python

```
>>> n = 20
>>> m = 25
>>> prod = n * m
>>> print('The product of', n, 'and', m, 'is', prod)
The product of 20 and 25 is 500
```

Python

```
>>> n = 20
>>> m = 25
>>> prod = n * m
>>> print(f'The product of {n} and {m} is {prod}')
The product of 20 and 25 is 500
```

String Formatting Operator

- One of Python's coolest features is the string format operator %.

```
x = 123
y = "Hello World"
print ("x = %i and y = %s" %(x, y))
```

```
x = 123 and y = Hello World
```

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Number Formatting

- There are various ways to format numbers using *str.format()*:
- Example:
  - *print("{:.2f}".format(3.1415))* return 3.14

```
print("{:.2f}".format(3.1415926));
```

Number	Format	Output	Description
3.1415926	{:.2f}	3.14	Format float 2 decimal places
3.1415926	{:+.2f}	+3.14	Format float 2 decimal places with sign
-1	{:+.2f}	-1.00	Format float 2 decimal places with sign
2.71828	{:.0f}	3	Format float with no decimal places
5	{:0>2d}	05	Pad number with zeros (left padding, width 2)
5	{:x<4d}	5xxx	Pad number with x's (right padding, width 4)
10	{:x<4d}	10xx	Pad number with x's (right padding, width 4)
1000000	{:,}	1,000,000	Number format with comma separator
0.25	{:.2%}	25.00%	Format percentage
1000000000	{:.2e}	1.00e+09	Exponent notation
13	{:10d}	13	Right aligned (default, width 10)
13	{:<10d}	13	Left aligned (width 10)
13	{:^10d}	13	Center aligned (width 10)

Escape Sequences in Strings

- Sometimes, you want Python to interpret a character or sequence of characters within a string differently. This may occur in one of two ways:
  - You may want to suppress the special interpretation that certain characters are usually given within a string.
  - You may want to apply special interpretation to characters in a string which would normally be taken literally.
- You can accomplish this using a backslash (\) character. A backslash character in a string indicates that one or more characters that follow it should be treated specially.

Escape Sequences

- The following table list the escape sequences which cause Python to suppress the usual special interpretation of a character in a string:

Escape Sequence	Usual Interpretation of Character(s) After Backslash	"Escaped" Interpretation
`\'`	Terminates string with single quote opening delimiter	Literal single quote (`'`) character
`\"`	Terminates string with double quote opening delimiter	Literal double quote (`"`) character
`\newline`	Terminates input line	Newline is ignored
`\\`	Introduces escape sequence	Literal backslash (`\`) character

Escape Sequences (Cont.)

Escape Sequence	"Escaped" Interpretation
\a	ASCII Bell (BEL) character
\b	ASCII Backspace (BS) character
\f	ASCII Formfeed (FF) character
\n	ASCII Linefeed (LF) character
\N{<name>}	Character from Unicode database with given <name>
\r	ASCII Carriage Return (CR) character
\t	ASCII Horizontal Tab (TAB) character
\uxxxx	Unicode character with 16-bit hex value xxxx
\Uxxxxxxxx	Unicode character with 32-bit hex value xxxxxxxx
\v	ASCII Vertical Tab (VT) character
\ooo	Character with octal value ooo
\xhh	Character with hex value hh

Escape Sequences

- Escape sequence is typically used to insert characters that are not readily generated from the keyboard or are not easily readable or printable.

Python

```
>>> print("a\tb")
a	b
>>> print("a\141\x61")
aaa
>>> print("a\nb")
a
b
```

Raw Strings

- A raw string literal is preceded by r or R, which specifies that escape sequences in the associated string are not translated.
- The backslash character is left in the string:

```
>>> print('foo\nbar')
foo
bar
>>> print(r'foo\nbar')
foo\nbar

>>> print('foo\\bar')
foo\bar
>>> print(R'foo\\bar')
foo\\bar
```

Boolean Type

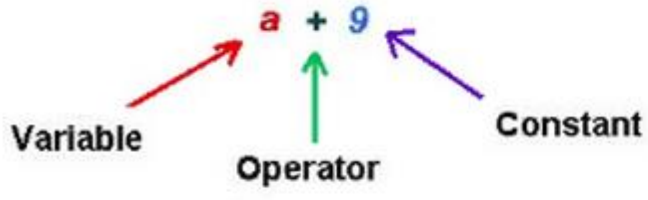
- Python 3 provides a Boolean data type.
- Objects of Boolean type may have one of two values, True or False:

Python

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Operators

- Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.



Arithmetic Operators

- Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.

Operator	Meaning	Example
+	Add two operands or unary plus	x + y +2
-	Subtract right operand from the left or unary minus	x - y -2
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

Example

Python

```
>>> 2 + 3   # Addition
5
>>> num1 = 10
>>> num2 = 9.99
>>> num3 = num1 + num2
>>> num3
19.990000000000002
>>> 8 - 5   # Subtraction
3
>>> 2 * 6   # Multiplication
12
>>> 12 / 3  # Division
4.0
>>> 7 % 3   # Modulus (returns the remainder from division)
1
>>> 3 ** 2  # Raise to the power
9
```

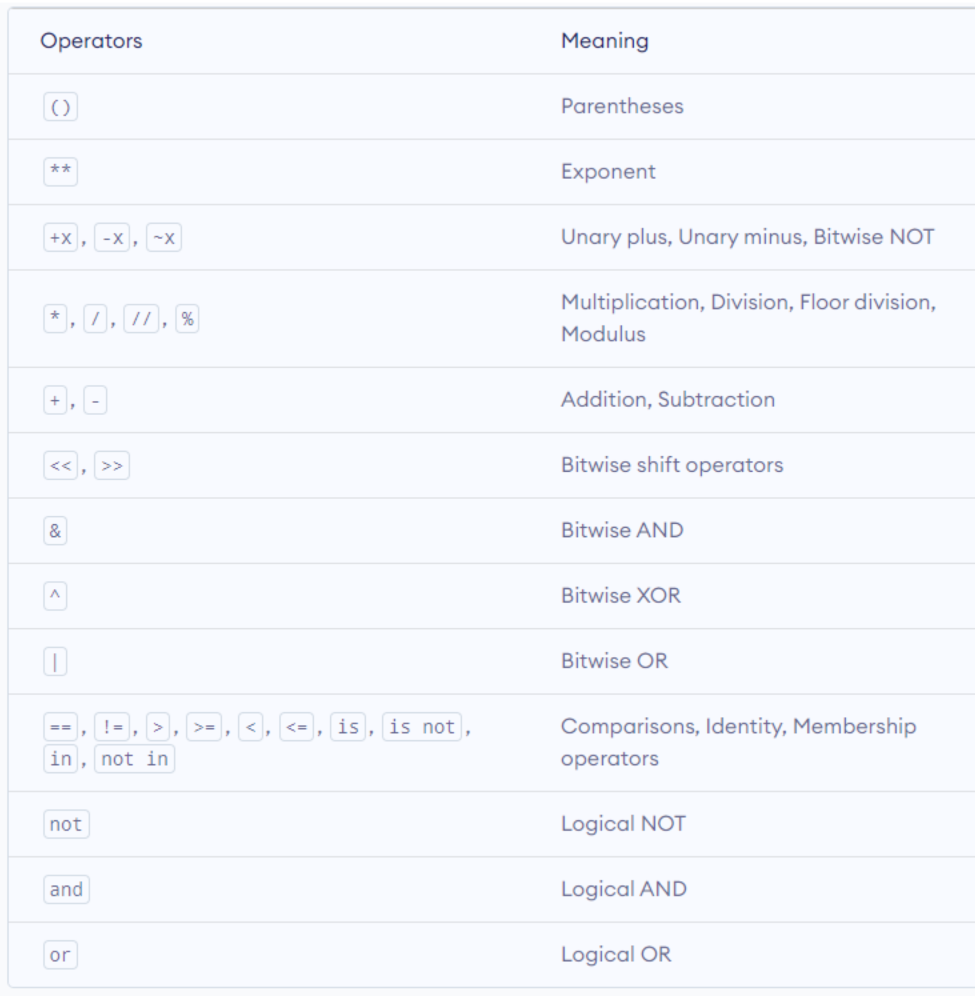
Assignment Operators

- Assignment operators are used in Python to assign values to variables.

Operator	Example	Equivatent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

Precedence of Python Operators

- The operator precedence in Python is listed in the table.
- It is in descending order (upper group has higher precedence than the lower ones).



Operators	Meaning
`()`	Parentheses
`**`	Exponent
`+x`, `-x`, `~x`	Unary plus, Unary minus, Bitwise NOT
`*`, `/`, `//`, `%`	Multiplication, Division, Floor division, Modulus
`+`, `-`	Addition, Subtraction
`<<`, `>>`	Bitwise shift operators
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`==`, `!=`, `>`, `>=`, `<`, `<=`, `is`, `is not`, `in`, `not in`	Comparisons, Identity, Membership operators
`not`	Logical NOT
`and`	Logical AND
`or`	Logical OR

Operators and Their Bindings

- The binding of the operator determines the order of computations performed by some operators with equal priority, put side by side in one expression.
- Most of Python’s operators have left-sided binding, which means that the calculation of the expression is conducted from left to right.

Left-Sided Binding

- Consider the statement:
  - print(9 % 6 % 2)
- There are two possible ways of evaluating this expression:
  - From left to right: first 9 % 6 gives 3, and then 3 % 2 gives 1;
  - From right to left: first 6 % 2 gives 0, and then 9 % 0 causes a fatal error.
- The result should be 1. This operator has left-sided binding. But there's one interesting exception.

Right-sided Binding

- Consider the statement:
  - print(2 ** 2 ** 3)
- The two possible results are:
  - 2 ** 2 → 4; 4 ** 3 → 64
  - 2 ** 3 → 8; 2 ** 8 → 256
- Run the code. The result clearly shows that the exponentiation operator uses right-sided binding.

Strings Manipulation

Manipulating Strings using Operator and Build-in Function

String

- Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes.
- Creating strings is as simple as assigning a value to a variable.
- For example:
  - var1 = 'Hello'
  - var2 = "Python"

Concatenate String

- The **+** operator concatenates strings. It returns a string consisting of the operands joined together.

Python

```python
>>> "happy" + " " + "birthday"
'happy birthday'
>>> "my name is " + "john"
'my name is john'
```

Repeating String

- The * operator creates multiple copies of a string.

```
Python
>>> s = 'foo.'

>>> s * 4
'foo.foo.foo.foo.'
>>> 4 * s
'foo.foo.foo.foo.'
```

Exist in String

- Python also provides a membership operator that can be used with strings. The **in** operator returns True if the first operand is contained within the second, and False otherwise

Python

```
>>> s = 'foo'

>>> s in 'That\'s food for thought.'
True
>>> s in 'That\'s good for now.'
False
```

Python

```
>>> 'z' not in 'abc'
True
>>> 'z' not in 'xyz'
False
```

String Length

- By using the function **len( )**, the length of the string can be obtained.

Python

```
>>> s = 'I am a string.'
>>> len(s)
14
```

Capitalize First Character

- **capitalize()** returns a copy of string with the first character converted to uppercase and all other characters converted to lowercase.

Python

```
>>> s = 'foO BaR BAZ quX'
>>> s.capitalize()
'Foo bar baz qux'
```

Convert to Upper Case

- **upper()** returns a copy of string with all alphabetic characters converted to uppercase

Python

```
>>> 'FOO Bar 123 baz qUX'.upper()
'FOO BAR 123 BAZ QUX'
```

Convert to Lower Case

- **lower()** returns a copy of string with all alphabetic characters converted to lowercase.

Python

```
>>> 'FOO Bar 123 baz qUX'.lower()
'foo bar 123 baz qux'
```

Convert to Title Case

- **title()** returns a copy of string in which the first letter of each word is converted to uppercase and remaining letters are lowercase
- This method uses a fairly simple algorithm. It does not attempt to distinguish between important and unimportant words, and it does not handle apostrophes, possessives, or acronyms gracefully

Python

```
>>> "what's happened to ted's IBM stock?".title()
"What'S Happened To Ted'S Ibm Stock?"
```

Swaps Case of Characters

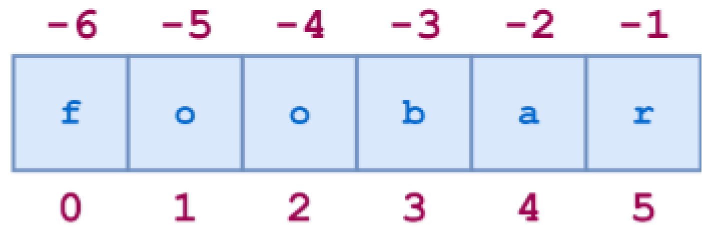
- **swapcase()** returns a copy of string with uppercase alphabetic characters converted to lowercase and vice versa

Python

```
>>> 'FOO Bar 123 baz qUX'.swapcase()
'foo bAR 123 BAZ Qux'
```

String Indexing

- String indexing in Python is zero-based: the first character in the string has index 0, the next has index 1, and so on. The index of the last character will be the length of the string minus one.
- For example, a schematic diagram of the indices of the string 'foobar' would look like this:



String Indexing (Cont.)

- The individual characters can be accessed by index as follows:

Python

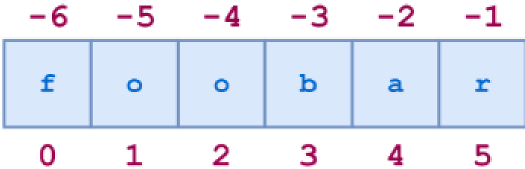
```python
>>> s = 'foobar'

>>> s[0]
'f'
>>> s[1]
'o'
>>> s[3]
'b'
>>> len(s)
6
>>> s[len(s)-1]
'r'
```

Python

```python
>>> s = 'foobar'

>>> s[-1]
'r'
>>> s[-2]
'a'
>>> len(s)
6
>>> s[-len(s)]
'f'
```

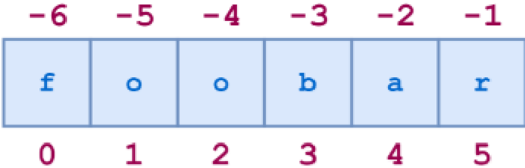


String Slicing

- Python also allows a form of indexing syntax that extracts substrings from a string, known as string slicing.
- If s is a string, an expression of the form s[m:n] returns the portion of s starting with position m, and up to but not including position n:

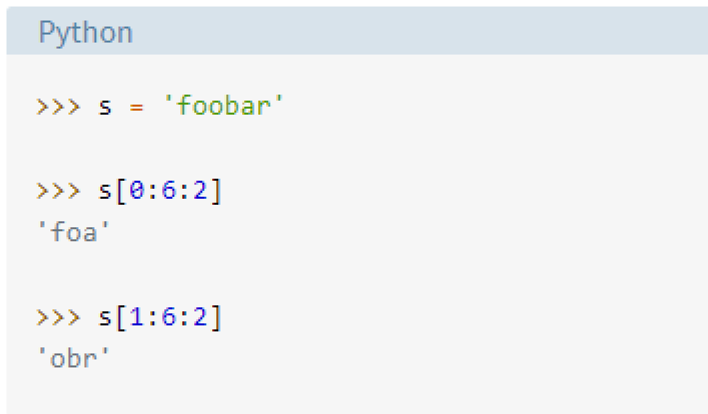
Python
```python
>>> s = 'foobar'

>>> s[:4]
'foob'
>>> s[0:4]
'foob'
```

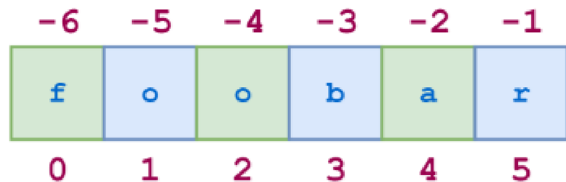


Specifying a Stride in a String Slice

- A third index designates a stride (step) can be added to slicing, which indicates how many characters to jump after retrieving each character in the slice.



For example, for the string 'foobar', the slice 0:6:2 starts with the first character and ends with the last character (the whole string), and every second character is skipped.
Similarly, 1:6:2 specifies a slice starting with the second character (index 1) and ending with the last character, and again the stride value 2 causes every other character to be skipped.



Counts Occurrences of Substring

- ***count(*<*sub*>*)*** returns the number of non-overlapping occurrences of substring <*sub*> in string:

Python

```
>>> 'foo goo moo'.count('oo')
3
```

Python

```
>>> 'foo goo moo'.count('oo', 0, 8)
2
```

The count is restricted to the number of occurrences within the substring indicated by <start> and <end>, if they are specified:

Determine Start String

- ***startswith(*** *<suffix>* ***)*** returns True if target string starts with the specified <*suffix*> and False otherwise

Python

```python
>>> 'foobar'.startswith('foo')
True
>>> 'foobar'.startswith('bar')
False
```

Python

```python
>>> 'foobar'.startswith('bar', 3)
True
>>> 'foobar'.startswith('bar', 3, 2)
False
```

The comparison is restricted to the substring indicated by <start> and <end>, if they are specified

Determines End String

- ***endswith(*<*suffix*>*)*** returns True if target string ends with the specified <*suffix*> and False otherwise

Python

```
>>> 'foobar'.endswith('bar')
True
>>> 'foobar'.endswith('baz')
False
```

Python

```
>>> 'foobar'.endswith('oob', 0, 4)
True
>>> 'foobar'.endswith('oob', 2, 4)
False
```

The comparison is restricted to the substring indicated by <start> and <end>, if they are specified

Searches for Substring

- ***find(***<*sub*>***)*** returns the lowest index in target string where substring <*sub*> is found

Python

```
>>> 'foo bar foo baz foo qux'.find('foo')
0
```

Python

```
>>> 'foo bar foo baz foo qux'.find('grault')
-1
```

Returns -1 if the specified substring not found

Python

```
>>> 'foo bar foo baz foo qux'.find('foo', 4)
8
>>> 'foo bar foo baz foo qux'.find('foo', 4, 7)
-1
```

The search is restricted to the substring indicated by <start> and <end>, if they are specified

Searches for Substring (Cont.)

- ***rfind(***<*sub*>***)*** returns the highest index in target string where substring <*sub*> is found

Python

```
>>> 'foo bar foo baz foo qux'.rfind('foo')
16
```

Python

```
>>> 'foo bar foo baz foo qux'.rfind('grault')
-1
```

Returns -1 if the specified substring not found

Python

```
>>> 'foo bar foo baz foo qux'.rfind('foo', 0, 14)
8
>>> 'foo bar foo baz foo qux'.rfind('foo', 10, 14)
-1
```

The search is restricted to the substring indicated by <start> and <end>, if they are specified

Trim Leading Space

- *lstrip( )* returns a copy of target string with any whitespace characters removed from the left end

Python

```
>>> '   foo bar baz   '.lstrip()
'foo bar baz   '
>>> '\t\nfoo\t\nbar\t\nbaz'.lstrip()
'foo\t\nbar\t\nbaz'
```

Python

```
>>> 'http://www.realpython.com'.lstrip('/:pth')
'www.realpython.com'
```

If the optional <chars> argument is specified, it is a string that specifies the set of characters to be removed

Trim Trailing Space

- ***rstrip( )*** returns a copy of target string with any whitespace characters removed from the right end

Python

```
>>> '   foo bar baz   '.rstrip()
'   foo bar baz'
>>> 'foo\t\nbar\t\nbaz\t\n'.rstrip()
'foo\t\nbar\t\nbaz'
```

Python

```
>>> 'foo.$$$;'.rstrip(';$.')
'foo'
```

If the optional <chars> argument is specified, it is a string that specifies the set of characters to be removed

Trim Leading and Trailing Space

- ***strip(<chars>)*** is essentially equivalent to invoking ***lstrip()*** and ***rstrip()*** in succession. Without the <*chars*> argument, it removes leading and trailing whitespace.

Python

```python
>>> are_you_happy = "     Yes      "
>>> are_you_happy.strip()
'Yes'
```

Python

```python
>>> 'www.realpython.com'.strip('w.moc')
'realpython'
```

As with .lstrip() and .rstrip(), the optional <chars> argument specifies the set of characters to be removed

Pad Leading Zero

- ***zfill(***<*width*>***)*** returns a copy of target string left-padded with '0' characters to the specified <*width*>

Python

```
>>> '42'.zfill(5)
'00042'
```

Python

```
>>> '+42'.zfill(8)
'+0000042'
>>> '-42'.zfill(8)
'-0000042'
```

If a string contains a leading sign, it remains at the left edge of the result string after zeros are inserted

Python

```
>>> '-42'.zfill(3)
'-42'
```

If string is already at least as long as <width>, it is returned unchanged

Alphanumeric Determination

- ***isalnum( )*** returns True if target string is nonempty and all its characters are alphanumeric (either a letter or a number), and False otherwise

Python

```
>>> 'abc123'.isalnum()
True
>>> 'abc$123'.isalnum()
False
>>> ''.isalnum()
False
```

Alphabetic Determination

- ***isalpha( )*** returns True if target string is nonempty and all its characters are alphabetic, and False otherwise

Python

```
>>> 'ABCabc'.isalpha()
True
>>> 'abc123'.isalpha()
False
```

Digit Determination

- ***isdigit( )*** returns True if target string is nonempty and all its characters are numeric digits, and False otherwise

```python
>>> '123'.isdigit()
True
>>> '123abc'.isdigit()
False
```

Upper Case Determination

- ***isupper( )*** returns True if target string is nonempty and all the alphabetic characters it contains are uppercase, and False otherwise. Non-alphabetic characters are ignored

Python

```
>>> 'ABC'.isupper()
True
>>> 'ABC1$D'.isupper()
True
>>> 'Abc1$D'.isupper()
False
```

Lower Case Determination

- ***islower( )*** returns True if target string is nonempty and all the alphabetic characters it contains are lowercase, and False otherwise.
- Non-alphabetic characters are ignored

Python

```
>>> 'abc'.islower()
True
>>> 'abc1$d'.islower()
True
>>> 'Abc1$D'.islower()
False
```

Title Case Determination

- ***istitle( )*** returns True if target string is nonempty, the first alphabetic character of each word is uppercase, and all other alphabetic characters in each word are lowercase. It returns False otherwise.

Python

```
>>> 'This Is A Title'.istitle()
True
>>> 'This is a title'.istitle()
False
>>> 'Give Me The #$#@ Ball!'.istitle()
True
```

Printable Character Determination

- ***isprintable( )*** returns True if target string is empty or all the alphabetic characters it contains are printable.
- It returns False if target string contains at least one non-printable character.
- Non-alphabetic characters are ignored

Python

```
>>> 'a\tb'.isprintable()
False
>>> 'a b'.isprintable()
True
>>> ''.isprintable()
True
>>> 'a\nb'.isprintable()
False
```

Space Determination

- ***isspace( )*** returns True if target sting is nonempty and all characters are whitespace characters, and False otherwise.
- The most encountered whitespace characters are space ' ', tab '\t', and newline '\n'

Python

```
>>> ' \t \n '.isspace()
True
>>> '   a   '.isspace()
False
```

Python

```
>>> '\f\u2005\r'.isspace()
True
```

'\f' and '\r' are the escape sequences for the ASCII Form Feed and Carriage Return characters; '\u2005' is the escape sequence for the Unicode Four-Per-Em Space.

Naming Convention

PEP 8 – Style Guide for Python Code

PEP 8 – Style Guide for Python Code

- PEP 8 is a Style Guide for Python Code, which gives coding conventions for the Python code comprising the standard library in the main Python distribution.
  - https://www.python.org/dev/peps/pep-0008/
- This style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in the language itself.
- Many projects have their own coding style guidelines. In the event of any conflicts, such project-specific guides take precedence for that project.

Code Layout

- Use 4 spaces per indentation level.
- Limit all lines to a maximum of 79 characters.
- Formulas always have line break before binary operations
- Surround top-level function and class definitions with two blank lines. Method definitions inside a class are surrounded by a single blank line.
- Code in the core Python distribution should always encoding in UTF-8
- Imports should usually be on separate lines

Whitespace in Expressions and Statements

- Avoid trailing whitespace anywhere.
- Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).
- If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority.

Comments

- Comments should be complete sentences. The first word should be capitalized, unless it is an identifier that begins with a lower case letter.
- Block comments generally consist of one or more paragraphs built out of complete sentences, with each sentence ending in a period.
- Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space.

Naming Conventions

- Never use the characters 'l', 'O', or 'I' as single character variable names.
- Identifiers used in the standard library must be ASCII compatible
- Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability
- Class names should normally use the CapWords convention.
- Names of type variables should normally use CapWords preferring short names: T, AnyStr, Num.
- Function names should be lowercase, with words separated by underscores as necessary to improve readability