Python Introduction

September 14, 2023

Contents

1 Introduction

Python is a versatile and high-level programming language that has gained immense popularity in the world of software development and data science. Created by Guido van Rossum and first released in 1991, Python is known for its simplicity, readability, and a vast ecosystem of libraries and frameworks that make it suitable for a wide range of applications. Python is an excellent choice for engineering for the following reasons:

- Readability: Python's syntax is clear and easy to read, making it an ideal language for both beginners and experienced programmers. Its use of indentation (whitespace) to define code blocks enforces clean and consistent coding practices.
- Versatility: Python is a general-purpose programming language, which means it can be used for a wide variety of tasks, from web development and data analysis to artificial intelligence and scientific computing.
- Extensive Libraries: Python boasts a rich ecosystem of libraries and frameworks that simplify complex tasks. For example, libraries like NumPy, Pandas, and Matplotlib are essential for data manipulation and visualization, while Django and Flask are popular choices for web development.
- Cross-Platform: Python is available on multiple operating systems, including Windows, macOS, and various Linux distributions, making it highly portable.
- Community and Support: Python has a large and active community of developers who contribute to its growth and provide support through forums, online tutorials, and extensive documentation.
- Interpreted Language: Python is an interpreted language, which means you can write and run code directly without the need for compilation. This feature speeds up development and debugging.
- Object-Oriented: Python supports object-oriented programming (OOP) principles, allowing you to create reusable and organized code structures.
- Dynamic Typing: Python uses dynamic typing, meaning you don't need to specify variable types explicitly. This makes code more concise and flexible but requires careful attention to variable naming and usage.
- Scalability: While Python may not be the fastest language for all tasks, it offers good scalability through libraries that interface with lower-level languages like C/C++ for performance-critical code sections.
- Open Source (Free): Python is open-source, which means it's freely available, and you can modify and distribute it according to open-source licenses.

Python's popularity continues to grow in fields like web development, data analysis, machine learning, and automation due to its simplicity, versatility, and the ever-expanding ecosystem of libraries and frameworks. Whether you're a beginner or an experienced developer, Python is an excellent language to learn and add to your toolkit for various programming tasks.

1.1 Variable Types

With any programming language, it is essential to know how to define and manipulate the different built-in variable types. For python, the most widely used variable types are:

- Numeric Types:
  - int: Represents integers, such as 1, -5, 1000.
  - float: Represents floating-point numbers with decimal places, such as 3.14, -0.01, 2.0.

  - complex: Represents complex numbers in the form a + bj, where a and b are floating-point numbers and j represents the imaginary unit (e.g., 3 + 2j).
- Text Type:
  - str: Represents strings of characters, enclosed in single (`' '`), double (`" "`), or triple (`''' '''` or `""" """`) quotes, such as `"Hello, World!"` or `'Python'`. Triple quotes are used for mult-line strings and are also used for documentation.
- Sequence Types:
  - list: Mutable collections of items. Lists are defined using square brackets, e.g., `[1, 2, 3]`.
  - tuple: Immutable collections of items. Tuples are defined using parentheses, e.g., `(1, 2, 3)`.
  - set: Mutable collection of items. Sets are defined using curly braces, e.g., `{1, 2, 3}`.
- Mapping Type:
  - dict: Collections of key-value pairs. Dictionaries are defined using curly braces and colons, e.g., `{'name': 'John', 'age': 30}`.
- Boolean Type:
  - bool: Represents Boolean values, either `True` or `False` and is used for logical operations.
- None Type:
  - NoneType: Represents the absence of a value. It is often used to signify the absence of a return value from functions or methods.
- Custom Types:
  - You can define your own custom data types using classes.

Python's dynamic typing allows you to change the type of variable by reassigning it with a value of a different type. For example, you can initialize a variable with an integer and later assign it a string value.

Example: Define variables for each commonly used variable type.

[1]:
```
# Numeric types
integer_variable = 42
float_variable = 3.14
complex_variable = 2 + 3j

# Text type
string_variable = "Hello, Python!"

# Sequence types
list_variable = [1, 2, 3]
tuple_variable = (4, 5, 6)
set_variable = {7, 8, 9}

# Mapping type
dict_variable = {'name': 'Alice', 'age': 25}

# Boolean type
bool_variable = True
```

```
# None type
none_variable = None
```

1.1.1 Numeric Types

Learning concepts for numeric data types in Python involve understanding how to work with integers, floating-point numbers, and complex numbers, as well as common operations and conversions.

Example: Show some of the common operations between integers, float, and complex datatypes.

[2]:
```
# Integers
x = 10
y = 5
# x, y = 10, 5  # Better way to define multiple variables

# Floats
x_float, y_float = float(x), float(y)  # Conversion to int can be done through␣
 ↪int() function. Same is true for all the datatypes.

# Complex
x_complex, y_complex = 2 + 3j, 1 - 2j

addition_result = x + y  # 15
subtraction_result = x - y  # 5
multiplication_result = x * y  # 50
division_result = x / y  # 2.0 (Note: Division always returns a float in Python␣
 ↪3)
power_result = x ** y  # 100_000
modulus_result = x % y  # 0 (Remainder of division)

# You can display them in jupyter using built-in display() function.
# This is essentially the print() statement, but tailored for jupyter notebooks.
display(addition_result, subtraction_result, multiplication_result,␣
 ↪division_result, power_result, modulus_result)
```

```
15
```

```
5
```

```
50
```

```
2.0
```

```
100000
```

```
0
```

[3]:
```
# For the floats
addition_result = x_float + y_float  # 15.0
subtraction_result = x_float - y_float  # 5.0
multiplication_result = x_float * y_float  # 50.0
division_result = x_float / y_float  # 2.0
power_result = x_float ** y_float  # 100000.0
display(addition_result, subtraction_result, multiplication_result,␣
 ↪division_result, power_result)
```

```
15.0
```

```
5.0
```

```
50.0
```

```
2.0
```

```
100000.0
```

[4]:
```
# For the complex numbers
addition_result = x_complex + y_complex
subtraction_result = x_complex - y_complex
multiplication_result = x_complex * y_complex
division_result = x_complex / y_complex
display(addition_result, subtraction_result, multiplication_result,␣
 ↪division_result)
```

```
(3+1j)
```

```
(1+5j)
```

```
(8-1j)
```

```
(-0.8+1.4j)
```

If there is a need for trig functions, square root functions, etc., the built in `math` module can be used. **Note:** There is very seldom a need for the `math` module when you have `numpy` imported.

[5]:
```
# More operations
import math

sqrt_result = math.sqrt(25)
sine_result = math.sin(math.radians(30))
display(sqrt_result, sine_result)
```

```
5.0
```

```
0.49999999999999994
```

[6]:
```
# Numeric comparisons
x, y, z = 3, 7, 3

x > y
```

[6]: False

[7]:
```
x >= z
```

[7]: True

[8]:
```
x == z
```

[8]: True

1.1.2 Strings

Learning concepts for string data types in Python involve understanding how to create, manipulate, and work with text-based data.

Example: Show string operations, indexing/slicing, some built in string methods, formatting, and escaping.

[9]:
```
# String operations
first, last = 'John', 'Davis'

string_add = first + " " + last  # Concatenation
string_replication = first*3  # Replication
display(string_add, string_replication)
```

'John Davis'

'JohnJohnJohn'

[10]:
```
# Indexing/Slicing
text = "System Dynamics"

first_character = text[0]  # indexing starts at zero
last_character = text[-1]
substring = text[2:6]
display(first_character, last_character, substring)
```

'S'

's'

'stem'

[11]:
```
# Some built-in string methods
text = "   Python Programming   \n"

cleaned_text = text.strip()  # Removed unwanted whitespace at the end and␣
 ↪beginning of the string
```

```
uppercase_text = cleaned_text.upper()  # Makes all characters uppercase
lowercase_text = cleaned_text.lower()  # Makes all characters lowercase
replace_text = cleaned_text.replace('Python', 'C++')  # Replaces text
split_text = cleaned_text.split(' ')  # Returns a list of the text split at␣
 ↪given string value
display(text, cleaned_text, uppercase_text, lowercase_text, replace_text,␣
 ↪split_text)
```

```
'   Python Programming   \n'
```

```
'Python Programming'
```

```
'PYTHON PROGRAMMING'
```

```
'python programming'
```

```
'C++ Programming'
```

```
['Python', 'Programming']
```

[12]:
```
# Formatting
name, age = 'John', 22

formatted_string = f'My name is {name}, and I am {age} years old.'
# formatted_string = 'My name is {}, and I am {} years old.'.format(name, age) ␣
 ↪# Same thing

formatted_number = f'This is pi formatted to 5 digits: {math.pi:.5f}.'
display(formatted_string, formatted_number)
```

```
'My name is John, and I am 22 years old.'
```

```
'This is pi formatted to 5 digits: 3.14159.'
```

[13]:
```
# String escaping (add extra \)
special_characters = 'Escape a newline using \\n'
special_characters
```

[13]:
```
'Escape a newline using \\n'
```

[14]:
```
print(special_characters)
```

```
Escape a newline using \n
```

[15]:
```
# Alternatively, define a raw string
special_characters = r'Escape a newline using \n'
special_characters
```

[15]:
```
'Escape a newline using \\n'
```

There are many more features available with strings, but one module to note is the built-in `re` module (regular expressions), which is capable of finding patterns in text (i.e. phone numbers, email addresses, etc.).

1.1.3 Python Lists

Lists are an essential datatype, and they provide a means for storing and operating on large data sets. **Note:** Python lists behave quite differently compared to numpy arrays.

Example: Show how to index/slice, operations with lists, some important methods associated with lists, and some functions associated with lists.

[16]:
```
# Indexing/slicing
states = ['Alabama', 'Mississippi', 'Georgia', 'Florida', 'Texas']

second = states[1]  # Mississippi (indexing starts at zero)
last = states[-1]
second_to_last = states[-2]
first_three = states[:3]  # does not include index 3
every_other = states[::2]  # [start:stop:step]
display(second, last, second_to_last, first_three, every_other)
```

```
'Mississippi'
```

```
'Texas'
```

```
'Florida'
```

```
['Alabama', 'Mississippi', 'Georgia']
```

```
['Alabama', 'Georgia', 'Texas']
```

[17]:
```
# Operations (similar to strings)
fruits = ['apple', 'banana']
veggies = ['carrot', 'lettuce']

combined = fruits + veggies  # adding concatenates
repeated = fruits*3  # multiplying will duplicate the list
is_apple_in_there = 'apple' in fruits
display(combined, repeated, is_apple_in_there)
```

```
['apple', 'banana', 'carrot', 'lettuce']
```

```
['apple', 'banana', 'apple', 'banana', 'apple', 'banana']
```

```
True
```

[18]:
```
# Some important methods
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
numbers.append(12)
numbers
```

[18]:
```
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 12]
```

[19]:
```
numbers.insert(1, 17)
numbers
```

[19]:
```
[3, 17, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 12]
```

[20]:
```
numbers.remove(3)  # If there are duplicates, it will only remove the first one␣
↪it sees
numbers
```

[20]:
```
[17, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 12]
```

[21]:
```
numbers.count(5)  # returns a value
```

[21]:
```
3
```

[22]:
```
numbers.index(4)  # returns the index of the given value
```

[22]:
```
2
```

[23]:
```
numbers.sort()  # Sorts the list in place (alphabetical for list of strings)
numbers
```

[23]:
```
[1, 1, 2, 3, 4, 5, 5, 5, 6, 9, 12, 17]
```

[24]:
```
numbers.clear()
numbers
```

[24]:
```
[]
```

[25]:
```
# Some important functions associated with lists
nums = [12, 1, 2, 3, 3, 8, 5, 7, 6, 5]
len(nums)  # This works with strings as well
```

[25]:
```
10
```

[26]:
```
sorted_nums = sorted(nums)  # creates another instance rather than changing it␣
↪in place
sorted_nums
```

[26]:
```
[1, 2, 3, 3, 5, 5, 6, 7, 8, 12]
```

[27]:
```
nums
```

[27]:
```
[12, 1, 2, 3, 3, 8, 5, 7, 6, 5]
```

Lists in python are mutable, which means that you can make changes to them. This is not true for tuples, which are immutable. That is, once it is defined, you cannot change it. Instead, you would have to create a new instance of the tuple that has the changes.

[28]:
```
a = [1, 2, 3, 4]
a[2] = 12
a
```

[28]:
```
[1, 2, 12, 4]
```

[29]:
```
a = (1, 2, 3, 4)
# a[2] = 12
# a
```

[30]:
```
# There is not much you can do with tuples
a = list(a)  # converts it to a list
a[2] = 12
a
```

[30]:
```
[1, 2, 12, 4]
```

A common issue with beginners for python lists is the incorrect method of creating a copy of a list.

[31]:
```
a = [1, 2, 3]
b = a
b.append(4)
b
```

[31]:
```
[1, 2, 3, 4]
```

[32]:
```
a
```

[32]:
```
[1, 2, 3, 4]
```

[33]:
```
# The proper way to make a copy
a = [1, 2, 3]
b = a.copy()  # b = a[:] also works
b.append(4)
b
```

[33]:
```
[1, 2, 3, 4]
```

[34]:
```
a
```

[34]:
```
[1, 2, 3]
```

1.1.4 Dictionaries

Python dictionaries are a unique way of storing and accessing packets of information. It is particularly advantageous for its ability to find specific information without knowledge of the index

number as you would have to deal with for lists.

Example: Show how you can access dictionaries and some of their common methods.

[35]:
```
# How you can access values
# {key: value}
person = {
    "name": "John",
    "age": 22,
    "school": "MSU"
}

# my_dict = dict(name='John', age=22, school='MSU')  # Same thing

person['name']
```

[35]:
```
'John'
```

[36]:
```
# You can add values
person['major'] = "Mechanical Engineering"

# my_dict.update({'major': 'Mechanical Engineering'})  # Same thing

person
```

[36]:
```
{'name': 'John', 'age': 22, 'school': 'MSU', 'major': 'Mechanical Engineering'}
```

[37]:
```
# Remove values
del person['major']

# major = my_dict.pop('major')  # This will remove and return the removed value␣
 ↪(key has to exist though)

person
```

[37]:
```
{'name': 'John', 'age': 22, 'school': 'MSU'}
```

[38]:
```
# Get the list of keys or the list of values
# You can ensure that they are converted to python lists by using the list()␣
 ↪function
keys = list(person.keys())
values = list(person.values())

keys, values
```

[38]:
```
(['name', 'age', 'school'], ['John', 22, 'MSU'])
```

[39]:
```
# You can get a list of the key, value pairs like this
items = list(person.items())
items
```

[39]:
```
[('name', 'John'), ('age', 22), ('school', 'MSU')]
```

[40]:
```
# If you try to get access to a key that is not present, then you can do one of␣
 ↪two things

# my_dict['girlfriend']  # This way will throw an error
print(person.get('girlfriend'))  # This way will return None if it doesn't exist
```

```
None
```

1.2 Functions, If Statements, and Loops

Functions, if statements, and loops are fundamental building blocks of programming and are essential for solving a wide range of computational problems. Here's why they are important:

- Functions:
  - Modularity and Reusability: Functions allow you to encapsulate a block of code into a reusable and modular unit. This promotes code reusability, reduces redundancy, and simplifies code maintenance.
  - Abstraction: Functions hide the implementation details and provide a high-level interface to perform specific tasks, making the code more understandable and manageable.
  - Organization: By breaking code into smaller functions, you can organize your program's logic into manageable chunks, improving code readability and maintainability.
  - Testing and Debugging: Functions make it easier to isolate and test specific parts of your code, aiding in the debugging process.
  - Collaboration: In collaborative development, functions provide a clear interface for team members to work on different parts of a project independently.
- If Statements (Conditional Statements):
  - Decision-Making: Conditional statements (if, elif, else) allow your program to make decisions based on specific conditions. This is crucial for implementing logic that reacts differently to varying inputs or circumstances.
  - Control Flow: Conditional statements control the flow of your program, enabling it to follow different paths depending on the conditions met. This is essential for building responsive and adaptable software.
  - Error Handling: Conditional statements can be used for error handling by catching and handling exceptions when unexpected conditions occur.
  - Validation: You can use conditional statements to validate input data and ensure that it meets certain criteria before proceeding with the execution of code.
- Loops:
  - Repetition: Loops (for and while) enable you to execute a block of code repeatedly, which is essential for automating repetitive tasks.
  - Iterating Over Data: Loops are used to iterate over collections (e.g., lists, dictionaries) or sequences (e.g., range of numbers) to process each element individually.

- Dynamic Data Handling: Loops allow you to handle data of varying lengths or sizes, making them versatile for tasks like data processing, reading files, and network communication.
- Efficiency: Loops can be used to perform operations on large datasets or perform calculations multiple times, improving the efficiency of your code.
- Flow Control: Loops control the flow of your program, enabling it to execute different parts of code based on conditions or until specific criteria are met.

In combination, functions, if statements, and loops provide the foundation for creating complex, responsive, and efficient programs in Python. They enable you to design logical and organized code, handle diverse data and input scenarios, and automate repetitive tasks. These concepts are essential for problem-solving and are used in almost every Python application, from simple scripts to large-scale software systems and data analysis.

1.2.1 Functions

Example: Write a function that greets a person, given their name.

[41]:
```
def greet(name_):
    print(f'Hello there, {name_}.')

greet('John')
```

```
Hello there, John.
```

[42]:
```
# You can add a default value for name_ in case one is not provided by using␣
↪keyword arguments.
def greet(name_="Jeffry"): print(f"Hello there, {name_}.")

greet()
```

```
Hello there, Jeffry.
```

[43]:
```
# One-liners can also be defined using lambda expressions
greet = lambda name_="Jeffry": print(f'Hello there, {name_}.')

greet(name_='Griffin')
```

```
Hello there, Griffin.
```

Functions in python are called *first class functions*, which means that they can be used as variables to be passed around.

Example: Write a function that uses the built-in `map()` function that will evaluate x^2 for each x in a list. The output of `[0, 1, 2, 3, 4]` would be `[0, 1, 4, 9, 16]`.

[44]:
```
def f(x_):
    return x_**2

# x = [0, 1, 2, 3, 4]
x = range(0, 5)  # Range object
out = list(map(f, x))
out
```

[44]:
```
[0, 1, 4, 9, 16]
```

1.2.2 If Statements

Example: Write an if statement that will determine if a score is an A, B, C, D, or F.

[45]:
```
score = 96

if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
elif score >= 60:
    grade = "D"
else:
    grade = "F"

print(f'Your grade is a(n) {grade}.')
```

```
Your grade is a(n) A.
```

Example: A person is eligible for a discount on pizza if they are older than 65 and attended MSU. Write a snippet that describes this problem.

[46]:
```
age = 76
school = 'MISSISSIPPI STATE'

if age > 65 and (school == 'MSU' or school.lower() == 'mississippi state'):
    print("You've got the discount")
else:
    print('No discount for you ... loser')
```

```
You've got the discount
```

1.2.3 Loops

Example: Write a snippet that will loop through a list and print each item.

[47]:
```
grades = [90, 87, 76, 100, 97, 56]

for grade in grades:
    print(grade)
```

```
90
87
76
100
97
56
```

Example: Write a snippet that will evaluate x^2 for each x in a list. The output of `[0, 1, 2, 3, 4]` would be `[0, 1, 4, 9, 16]`.

[48]:
```
output_list = []
for i in range(5):
    output_list.append(i**2)
output_list
```

[48]:
```
[0, 1, 4, 9, 16]
```

[49]:
```
# list comprehension
output_list = [i**2 for i in range(5)]
output_list
```

[49]:
```
[0, 1, 4, 9, 16]
```

[50]:
```
# Use numpy arrays for elementwise operations
import numpy as np
x = np.arange(5)
x**2
```

[50]:
```
array([ 0,  1,  4,  9, 16])
```