Introductory Python Review Guide

Hyosang Ahn

Prologue

This guide provides a concise overview of fundamental topics and features of Python programming, which will help beginners solve problems on CodingBat and create light games in Python.

NOTE: This compact guide is not a textbook, not comprehensive, and may not cover all the details in the topics presented. If you have any questions about any topic in this guide, feel free to search the internet or ask a coach.

Contents

1 Hello World

1.1 Print Statements

The first step in programming is to print something on the screen. By convention, we call this step a "print statement":

```
1 print("Hello World!")
```

```
Hello world!
```

While being a simple command, print statements are useful in monitoring the status of code execution and debugging.

1.2 Comments

Throughout this guide, you'll see words that are followed after `#`. These words are **comments**. They do not contribute to running the code but are used to describe code.

```
print("Hello World!")
# I don't do anything.
```

```
Hello world!
```

2 Variables and Data Types

A **variable** in Python is a data storage and tracking asset. You might have encountered variables in expressions like the following:

$$x = 7$$
$$y = 3.14$$
$$z = x + 3$$

2.1 Data Types

Python supports various data types such as **integers**, **strings**, **floats**, and **Booleans**:

```
location = "Pasadena, CA"    # String
x = 7                        # Integer
y = 3.14                     # Float
is_daytime = True            # Boolean
```

These four types are called **primitive data types**, which means that a variable contains a single data.

- **Strings** are data sets made up of a group of characters, such as letters, numbers, and punctuation marks. These data are represented by enclosing the words or phrases within a pair of single or double quotation marks.
- **Integers** are numbers, both positive and negative, without a decimal point.
- **Floats** are numbers, both positive and negative, with a decimal point.
- **Boolean** values can be either `True` or `False`.

In above code, the four lines assigned values on the right-hand-side of `=` to the respective variable on the left-hand-side.

2.2 Operators

Python supports a series of operators, including addition, subtraction, multiplication, division, modulus (remainder), and power.

There are two modes of division: `/` gives a quotient in decimal form, whereas `//` gives a quotient in integer, floored, or rounded down, from the decimal.

```
x = 7              # 7 is assigned to x.
y = 3              # 3 is assigned to y.

sm = x + y             # Sum
diff = x - y           # Difference
prod = x * y           # Product
quo = x / y            # Quotient
intquo = x // y        # Floored Quotient
mod = x % y            # Modulus
                       #   (Remainder)
pwr = x ** y           # Power (x^y)
```

If any of the *binary* operations assigns the output back to the same variable, such as `x = x + y`, they can be shortened into following:

```
x = x + y  →  x += y
x = x - y  →  x -= y
x = x * y  →  x *= y
x = x / y  →  x /= y
x = x // y →  x //= y
x = x % y  →  x %= y
x = x ** y →  x **= y
```

2.3 Strings

Strings can be **concatenated** using the + operator. A combination of strings and string variables can also be concatenated. However, data of other types, such as integers or floats, must be converted to strings before concatenation using the `str(<any>)` function.

```
1 my_string = "Hello " + "world!"
2 print(my_string)
3
4 my_string2 = my_string + " How are you
      doing?"
5 print(my_string2)
6
7 my_age = 15
8 my_string3 = my_string2 + " I am " +
      str(my_age) + " years old!"
9 print(my_string3)
```

```
Hello world!
Hello world! How are you doing?
Hello world! How are you doing? I
    am 15 years old!
```

3 If-Else Statements

Most logics in Python consist of **if-statements**. To create an if-statement, we do the following steps:

1. insert an `if` keyword
2. define the condition to satisfy
3. insert the body

The bodies are defined by indenting a block of lines. These lines are executed only if the condition defined in the if-statement is *true*. A condition being true means that the condition can be *simplified into the value* ***True*** (recall the Boolean values).

By using `else` keyword, we can define another body to be executed only if the condition is *false* (the condition *simplifies into the value* ***False***).

```
1 x = 3
2 y = 7
3 if x > y: # This simplifies to True
4     print("x is greater than y.")
5 else: # In other words, x <= y.
6     print("x is less than or equal to
          y.")
```

```
x is less than or equal to y.
```

If above code is run, it will only print the following: `x is greater than y.` This is because the condition defined in line 3 is true.

By using `elif` keyword (`elif` stands for "else if"), we can define a second condition to check if the preceding condition is not satisfied.

```
1 x = 7
2 y = 3
3 if x > y:
4     print("x is greater than y.")
5 elif x < y:
6     print("x is less than y.")
7 else: # In other words, x == y.
8     print("x is equal to y.")
```

```
x is greater than y.
```

Note that if the previous condition is met, the computer will **ignore the conditions and followup actions in the subsequent `elif`-statements**.

3.1 Inequality

These are the syntax to represent inequalities in Python:

$$\begin{aligned}
\texttt{x == y} &\longleftrightarrow x = y \\
\texttt{x > y} &\longleftrightarrow x > y \\
\texttt{x < y} &\longleftrightarrow x < y \\
\texttt{x >= y} &\longleftrightarrow x \geq y \\
\texttt{x <= y} &\longleftrightarrow x \leq y \\
\texttt{x != y} &\longleftrightarrow x \neq y
\end{aligned}$$

Note that the double equal signs (==) is used to express the *condition of two variables with*

equal values, while the single equal sign (`=`) *assigns a value into a variable.*

3.2 `and`, `or`, `not`

We can create *compound conditions* with a mix of `and`, `or`, and `not` syntax.

- `a and b` outputs true if **both** conditions `a` and `b` are true.
- `a or b` outputs true if **at least one** condition of `a` or `b` is true.
- `not a` outputs true if `a` is **not** true.

```
age = 20
has_drivers_license = True
if age >= 18 and has_drivers_license:
    print("You can drive.")

is_student = True
is_employed = False
if is_student or is_employed:
    print("You can get discount.")

is_hungry = False
if not is_hungry:
    print("You are full.")
```

```
You can drive.
You can get discount.
You are full.
```

The **order of operations matter** in compound condition. That is, `not` is operated first, then `and` and `or` in order. Parentheses can be used to change the order. To familiarize with this, consider practicing to build **truth tables** with given compound conditions.

3.3 Nested if-statements

Suppose condition `a` is true, and one wants to build a consequential action that depends on whether condition `b` is true or false. That's when the nested if-statement comes in. A series of if-statements can be wrapped as a followup action of an outer if-statement.

```
age = 20
has_drivers_license = False
has_training_permit = True
if age >= 18:
    if has_drivers_license:
        print("You can drive.")
    elif has_training_permit:
        print("You can drive under
            supervision.")
    else:
        print("You cannot drive.")
else:
    print("You cannot drive.")
```

```
You can drive under supervision.
```

Multiple layers of if-statements are allowed.

4 Loops

Sometimes we want certain codes to run multiple times. Perhaps each cycle differs a bit, but you notice a pattern inside of it. In this case, we use **loops**. There are two ways to use loops.

4.1 While-Loops

Recall if-statements. The body is run only if the condition is true. By replacing the `if` syntax with `while`, the body will *repeat* while the condition is true. In other words, it will *stop repeating* once the condition becomes false.

```
i = 0
while i < 5:
    print(i)
    i += 1
```

```
0
1
2
3
4
```

Above code will print the value of `a`, starting with 0, increment the value by 1, and repeat, until `a` becomes 10 (exclusive). It is similar

to the "repeat until" block in Scratch, except while-loops behave the opposite with the condition.

Using what we've learned, how can we simulate the "forever" block in Scratch, where the body repeats infinitely?

4.2 For-Loops

Unlike while-loops that repeatedly check a condition to continue executing, for-loops traverse through iterable data and retrieve an item from the data in each iteration. Data is considered **iterable** if a loop can be used to iterate through its contents. Examples include a range of integers via `range()` (see section 4.2.1), the letters of a string, or lists (see section 5.1 of page 6).

```
for i in range(5):
    print(i)

for letter in "Python is fun!":
    print(letter)
```

```
0
1
2
3
4
P
y
t
h
o
n

i
s

f
u
n
!
```

The first for-loop behaves the same as the previous code. `range(10)` gives you a range of integers from 0 to 9, inclusive. While there are more use cases of `range()` (see next part), in here, `a` is assigned with each number of the range in order per iteration. Overall, this for-loop repeats 10 times.

4.2.1 `range()`

`range()` is a function (see section 6 of page 9 to learn about properties of functions) that can give you not just the range of numbers starting at 0.

```
range(start=0, end, step=1)
```

where all parameters are integers, and `start` and `step` are defaulted at `0` and `1`, respectively. It returns a range of numbers starting at `start` (inclusive) and stops at `end` (exclusive), where the distance between two consecutive numbers in the range is `step`. This is why `range(10)` gives you the range from 0 to 9, spaced by 1.

To *reverse* the order of a range of numbers, simply make the `step` negative and swap the values of `start` and `end`. However, be sure to consider whether the range is inclusive or exclusive.

```
range(4)           # 0, 1, 2, 3
range(2, 10, 2)    # 2, 4, 6, 8
range(0, 7, 2)     # 0, 2, 4, 6
range(6, -1, -2)   # 6, 4, 2, 0
```

4.3 Nested Loops

Like nested if-statements, loops can be nested by each other as well. This is useful for cycles where each sequence requires running a separate cycle, such as 2D list (see section 5.1.2 of page 7).

```
for i in range(3):
    j = 0
    while j < 5:
        print(i, j)
        j += 1
```

```
0 0
0 1
0 2
0 3
```

```
0 4
1 0
1 1
...
```

Remember that a loop moves to the next cycle after reaching the end of its body in the current cycle.

4.4 `break` and `continue`

The two syntax **`break`** and **`continue`** are used to exit a loop or the current cycle of the loop, respectively. For example:

```
for i in range(10):
    if i == 5:
        break
    print(i)

for i in range(10):
    if i % 2 == 1:
        continue
    print(i)
```

```
0
1
2
3
4
0
2
4
6
8
```

The first for-loop will print `i` up to 4 because the loop exits when `i` reaches 5. The second for-loop will only print even numbers because the cycle is skipped if `i` is an odd number.

5 Collections

In Python, a series of data can be organized in a **collection**. There are three commonly-used collections: lists, tuples, and dictionaries.

5.1 Lists

A **list** is an array consisting of data in a specific order. An **element** of a list can be retrieved by using **index**, a number that represents the *position* of the element in the list's order. In Python, indices are 0-based; in other words, the first element of the list has index 0, the second has 1, and so on.

Using indices, elements of the lists can be retrieved or replaced using get and set methods. See below example to see how they're done:

```
my_list = [1, 2, 3]

# Get method
print(my_list[0])  # Access the first element
print(my_list[1])  # Access the second element
print(my_list[-1]) # Access the LAST element
print(my_list[-2]) # Access the SECOND LAST element

# Set method
my_list[1] = 4
print(my_list)
```

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

A list is an iterable, so one can use for-loop to traverse through a list.

```
my_list = [1, 2, 3]
for elem in my_list:
    print(elem)
```

```
1
2
3
```

Which prints each number in `my_list` per line.

A list may contain items of different data types, including lists.

```
my_list = [1, 2.4, "Coding", [2, 5]]
```

If you need the access to each element's index per cycle, you can do the following:

```
my_list = ["a", "b", "c"]
for i, elem in enumerate(my_list):
    print(i, elem)
```

```
0 a
1 b
2 c
```

In the above code, `i` stores the index of the element in each iteration. For example, when `elem = "a"`, then `i = 0`.

5.1.1 Sublists

A sublist can be retrieved by using a range of indices, similar to the `range()` method (see section 4.2.1 in page 5).

```
my_list = ["a", "b", "c", "d", "e"]
print(my_list[1: 4])
print(my_list[4: 1: -1])
print(my_list[1: 4: 2])
```

```
['b', 'c', 'd']
['e', 'd', 'c']
['b', 'd']
```

5.1.2 2D and nD Lists

A 1D list may not be suitable for representing grids or tables. Instead, a 2D list can be constructed by leveraging the property of lists that allows them to contain other lists as elements. See how indexing and traversing (using nested for-loops) are done below:

```
my_list = [["a", "b", "c"],
           ["d", "e", "f"],
           ["g", "h", "i"]]

print(my_list[0][0])
print(my_list[1][-1])
print()

for i, row in enumerate(my_list):
    for j, elem in enumerate(row):
        print(i, j, elem)
```

```
a
f

0 0 a
0 1 b
0 2 c
1 0 d
1 1 e
1 2 f
2 0 g
2 1 h
2 2 i
```

By incorporating additional lists within the inner lists, lists with dimensions of three or higher can be constructed.

5.1.3 Useful Built-in Functions

There are several built-in functions (see section 6 of page 9 to learn about properties of functions) for lists. Below are commonly used list functions:

- `len(<list>)` returns the **length** of a list. A length is the number of elements in the list.

```
my_list = ["a", "b", "c", "d"]
print(len(my_list))
```

```
4
```

- `<list>.append(<any>)` adds an element at the end of the list.

```
my_list = ["a", "b", "c", "d"]
my_list.append("e")
print(my_list)
```

```
['a', 'b', 'c', 'd', 'e']
```

- `<list>.insert(<int>, <any>)` adds an element at the position of given index in the list. The subsequent parts of the list are pushed backward.

```
my_list = ["a", "c", "d"]
my_list.append(1, "b")
print(my_list)
```

```
['a', 'b', 'c', 'd']
```

- `<list>.pop(<int>=-1)` removes the element at given index of the list and returns it. If no index is given, it removes the element at *last* index.

```
my_list = ["a", "b", "c", "d"]
my_list.pop(0)
print(my_list)
my_list.pop()
print(my_list)
```

```
['b', 'c', 'd']
['b', 'c']
```

- `<any> in <list>` checks whether the list has at least one element of a certain value.

```
my_list = ["a", "b", "c", "d"]
if "b" in my_list: # True
    print("my_list has a \"b\".")
if "e" in my_list: # False
    print("my_list has an \"e\".")
```

```
my_list has a "b".
```

5.2 Tuples

Lists are **mutable**; in other words, the size (length) of lists can be changed after initialized using functions such as `append()`.

A **tuple**, while similar to lists, is an **immutable** collection. Once initialized, their length and elements cannot be changed. However, reassigning a tuple to a variable allows for replacing the entire tuple.

```
my_tuple = (0, 0)
print(my_tuple)
print(my_tuple[0])
my_tuple = (0, 1)
print(my_tuple)
my_tuple[0] = 1      # Throws error!
print(my_tuple)
```

```
(0, 0)
0
(0, 1)
TypeError: 'tuple' object does not
    support item assignment
```

Tuples can be substituted with lists in most use cases, which allows for more flexible actions. However, using tuples can help reduce memory usage.

5.3 Dictionaries

Lists have their elements ordered via their index system. On the other hand, a **dictionary** is an unordered collection where each of its elements consist of key-value pairs. Imagine that a list's indices are the "keys" to access or modify its elements. Similarly, accessing or modifying a value of a dictionary can be done by using the corresponding key. Below shows the structure of dictionaries:

`{keyA: valueA, keyB: valueB}`

where each key and value is separated by `:`.

```
my_list = ["a", "c", "e"]
print(my_list[1]) # 1 is the "key" to
    access the element "c".

my_dict = {"Andrew": 8, "Brian": 5, "
    Charlie": 10}
print(my_dict["Brian"]) # "Brian" is
    the key of the dictionary to access
     the corresponding value 5.

my_dict["Brian"] = 9
print(my_dict["Brian"])
```

```
c
5
9
```

The keys of a dictionary can consist of any *immutable* data types (including tuples) and the mix of the types. The values of a dictionary can consist of any data types.

```python
my_dict = {"David": 9,
           8: ["Hello", "Hi"],
           (1, 2): 2.4
} # This is a valid dictionary.

print(my_dict[(1, 2)])
```

```
2.4
```

While dictionaries are unordered collections, they can be iterated using for loops, saving either key, value, or key-value pair (in form of tuple) as an iterable per cycle:

```python
my_dict = {"Andrew": 8, "Brian": 5, "
    Charlie": 10}

# Key as the iterable
for key in my_dict: # 'my_dict' may be
     replaced with 'my_dict.keys()'
    print(key)

# Value as the iterable
for value in my_dict.values():
    print(value)

# Key-Value pair (tuple) as the
    iterable
for item in my_dict.items():
    print(item)
```

```
Andrew
Brian
Charlie
8
5
10
('Andrew', 8)
('Brian', 5)
('Charlie', 10)
```

5.3.1 Useful Built-in Functions

`len`, `pop`, and `in` functions used for lists (see section 5.1.3 in page 9) can be applied for dictionaries. Note that the `pop` function takes the *key* of a pair to remove an item from the dictionary, while the `in` function asserts whether a *key* exists in the dictionary.

```python
my_dict = {"Andrew": 8, "Brian": 5, "
    Charlie": 10}

print(len(my_dict))

print(my_dict.pop("Brian"))
print(my_dict)

print("Charlie" in my_dict, "Brian" in
     my_dict)
```

```
3
5
{'Andrew': 8, 'Charlie': 10}
True False
```

Since dictionaries are unordered, `append` or `insert` functions cannot be used. Instead, the set method with an unintroduced key can be used to add new key-value pair.

```python
my_dict = {"Andrew": 8, "Brian": 5, "
    Charlie": 10}

my_dict["Daniel"] = 12
print(my_dict)
```

```
{'Andrew': 8, 'Brian': 5, 'Charlie
    ': 10, 'Daniel': 12}
```

6 Functions

If you've studied about algebra before, you may be familiar with **functions**. They allow assessing the results for every possible inputs (parameters), making them reusable tools for repeated operations.

$$f(x) = x + 6$$
$$g(x, y) = x * 2y$$

$$f(5) = 11$$
$$f(8) = 14$$

$$g(5, 3) = 11$$
$$g(13, 17) = 47$$

Such mathematical concept can be applied to Python.

6.1 Structure

6.1.1 Header

A header in Python is structured as below:

```
def <function_name>(<argument>...):
```

where `function_name` is the name of the function, and `argument` are the arguments of the function. Below is an example of the header:

```
def add_six(x):
```

An **argument** is an input variable–in any type–of the function, where its actual value is plugged in from the parameters of function callers (see section 6.2 in page 11). A function may have multiple arguments or no arguments.

In Python 3, arguments can be set with a certain data type like below:

```
def add_six(x: int):
```

Such ensures consistent data type of the parameters when calling the function. It is optional; however, it is strongly advised to treat the arguments as if it has consistent data type while running the function (e.g., if argument `a` is assumed as an int type, do not use `len` method).

6.1.2 Return Statement

Functions have inputs in form of arguments and outputs in form of **return** statements. Once a function has information to output, it can return in the form of below:

```
return <any>
```

Below shows an example function body with return statement:

```
to_return = x + 6
return to_return
```

Once the computer reaches a return statement of a function, it will immediately exit from the function and carry out the return value. All lines inside the function body after the return statement will be ignored.

Like arguments, functions can have multiple return statements, usually one per branch of the logic tree. In each return statement, multiple values can be returned, separated by a comma:

```
return <any>, <any>, ...
```

which will yield a tuple of return values.

The data type of the returned items, by convention, are called **"return types."** While it's possible for a function's return type to vary depending on the parameters, it's generally recommended to maintain consistency in return types per function.

In Python 3, return types can be set with a certain data type like below:

```
def add_six(x: int) -> int:
```

Below is the complete Python function with the same purpose as $f(x)$ and $g(x)$ from the beginning of this section. It's important to note that the return value can be a variable or an operation, as long as it simplifies to the desired return type.

```
1 def add_six(x):
2     to_return = x + 6
3     return to_return
4
5 def add_2y(x, y):
6     return x + 2 * y
```

See the next subsection to learn how to call a function.

6.2 Function Caller

At the beginning of the section, we made two algebraic functions, followed by their callers by plugging in numbers to the parameters. In this subsection, we will learn how to call functions in Python.

Function callers generally consist of below structure:

```
<ret_var>...  = <func_name>(<param>...)
```

A function caller assigns a value to each parameter (`<param>`), which is then plugged into the corresponding (same position in order) argument of the function. After executing the function with the provided parameters, the return value is assigned to the return variable `<ret_var>`. The subsequent lines can then utilize the value stored in the return variable for the subsequent operations. It's crucial to remember that, like variables, function names must be consistent to access them.

Below shows the callers of the previously defined example functions:

```python
def add_six(x):
    to_return = x + 6
    return to_return

def add_2y(x, y):
    return x + 2 * y

add_five_to_six = add_six(5)
print(add_five_to_six)

print(add_six(8))
print(add_2y(5, 3))
print(add_2y(13, 17))
```

```
11
14
11
47
```

6.3 Void Function

Functions without return statements are called **void functions**. Void functions are often used to run a series of other functions, modify variables defined outside the function, and other actions that do not need to return outputs, such as print statements.

To exit the void function in the middle of its body, return a null value (`return None`).

```python
def introduce(name, age):
    print(f"Hello, my name is {name}.")
    if age >= 18:
        print("I can drive a car.")
        return None # Exits from the function.
    print("I can't drive a car, but maybe I can get a permit.")

introduce("Andrew", 25)
introduce("Brian", 14)
```

```
Hello, my name is Andrew.
I can drive a car.
Hello, my name is Brian.
I can't drive a car, but maybe I can get a permit.
```

7 More Tips for Python

7.1 Libraries

Libraries offer functions and classes (object-oriented programming will be covered in the next guide) to simplify coding. However, to utilize these tools, the relevant library must be **imported**. For most cases, the import statements are placed at the beginning of the file to enable the subsequent lines of the file to access the library.

```
import <library_name>
...
<library_name>.<function_name>()
```

where `<library_name>` is replaced by the name of the library that contains the desired function.

This guide introduces two Python libraries, emphasizing the use of their imported functions and other properties. However, please note that the listed library properties are not exhaustive, so it's recommended to explore other methods available within the libraries for further learning.

7.1.1 `random`

`random` library offers tools for actions that relate to random selections. Below are a few examples of functions from `random`:

- `random.randint(start: int, end: int)` returns an integer randomly selected from the range between the `start` and `end`, inclusive.
- `random.choice(list: List)` returns an element randomly selected from `list`. This method works on tuples, but for dictionaries, `keys()`, `values()`, or `items()` should be used to convert into iterable sequences before using the method (see section 5.3 in page 8).

```
1  import random
2
3  print(random.randint(1, 10))
4
5  my_list = ["a", "c", "e"]
6  print(random.choice(my_list))
```

```
1
a
```

7.1.2 `math`

`math` library has tools that do mathematical operations that can't be done with Python's built-in methods. Below are a few examples of functions and constants from `math`:

- `math.sqrt(num: int/float)` returns the square root of `num` in float.
- `math.inf` is a float constant representing infinity.
- `math.pi` returns the float constant of π.

```
1  import math
2
3  print(math.sqrt(16))
4  print(math.sqrt(10.6))
5
6  print(math.inf)
7  if math.inf > 1e32:
8      print("Greater")
9
10 print(math.pi)
```

```
4.0
3.255764119219941
inf
Greater
3.141592653589793
```

Epilogue

Remember that this guide covers only the fundamental topics of Python necessary to solve all problems on CodingBat and create light games and projects. For more advanced projects like games with graphics, it is recommended to learn about object-oriented programming (OOP).