Introductory Python Review Guide

Updated June 28, 2025

Contents

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 simple games on command line in Python.

Use a Python interpreter like `programiz.com` to test the presented code blocks.

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

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.

```
x = 7
y = "Hello World!"

print(x)
print(y)
```

```
7
Hello World!
```

In above code, the first two lines assigned values on the right-hand-side of `=` to the respective variable on the left-hand-side. Now, `x` and `y` are variables, each representing a specific value.

By common practice, variables are named in lower-case alphabets and numbers spaced with underscores (`"_"`). Names that use non-alphanumeric characters (other than underscores) or start with a number are not allowed.

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
```

Integers, floats, and Boolean types are classified as **primitive data types**, which implies that a variable holds a single piece of data. In contrast, strings are not primitive because they are sequences of characters, implying that they comprise multiple data units.

- **Strings** are ordered 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** (floating numbers) are numbers, both positive and negative, with a decimal point.
- **Boolean** values can be either `True` or `False`.

If you're familiar with other common languages like Java or C++, you'll notice that in Python, the data type isn't explicitly specified when defining a variable. If you're familiar with languages like C# or JavaScript, you also won't find variable markers like `local` or `let`. However, as a compensation, it's crucial to always *initialize* a variable when defining it.

2.2 Operators

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

```
1  x = 9              # 9 is assigned to x.
2  y = 2              # 2 is assigned to y.
3
4  sm = x + y            # Sum
5  diff = x - y          # Difference
6  prod = x * y          # Product
7  quo = x / y           # Quotient
8  intquo = x // y       # Floored Quotient
9  mod = x % y           # Modulus
10                       #   (Remainder)
11 pwr = x ** y          # Power (x^y)
12
13 print(sm, diff, prod, quo, intquo, mod, pwr)
```

```
11 7 18 4.5 4 1 81
```

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

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
```

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 the optional `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 the optional `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 compiler will ignore the conditions and bodies in the subsequent `elif`- and `else`-statements.

3.1 Inequality

These are the syntax to represent inequalities in Python:

`x == y`	$\longleftrightarrow$	$x = y$
`x > y`	$\longleftrightarrow$	$x > y$
`x < y`	$\longleftrightarrow$	$x < y$
`x >= y`	$\longleftrightarrow$	$x \geq y$
`x <= y`	$\longleftrightarrow$	$x \leq y$
`x != y`	$\longleftrightarrow$	$x \neq y$

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.

```
1  age = 20
2  has_drivers_license = True
3  if age >= 18 and has_drivers_license:
4      print("You can drive.")
5
6  is_student = True
7  is_employed = False
8  if is_student or is_employed:
9      print("You can get discount.")
10
11 is_hungry = False
12 if not is_hungry:
13     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 5 (exclusive). It is similar to the "repeat until" block in Scratch, except while-loops behave the opposite with the condition.

Using the while-loop, how can we simulate the "forever" block in Scratch, where the body repeats infinitely? (Hint: think of an if-statement that *always* passes.)

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 5.2 of page 12).

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

for letter in "Python":
    print(letter)
```

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

The first for-loop behaves the same as the previous code. `range(5)` gives you a range of integers from 0 to 4, inclusive. While there are more use cases of `range()` (see next part), in here, `i` is assigned with each number of the range in order per iteration. Overall, this for-loop repeats 5 times.

4.2.1 `range()`

`range()` is a function (see section 6 of page 20 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(5)` gives you the range from 0 to 4, 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 5.2.3 of page 14).

```
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 four commonly-used collections: strings, lists, tuples, and dictionaries.

5.1 Strings

Recall from chapter 2 that strings are sequences of characters enclosed in quotation marks. While we initially introduced strings as a basic data type alongside integers and floats, strings are actually **non-primitive** data types that belong to the family of collections.

Unlike primitive data types (integers, floats, Booleans) that store a single value, strings are **collections of characters**. Each character in a string occupies a specific position (index), making strings **ordered sequences**. This collection nature allows strings to share many behaviors with other collections like lists—they can be indexed, sliced, traversed, and measured for length using the same methods.

For example, the string `"Hello"` is actually a collection containing five characters: `'H'`, `'e'`, `'l'`, `'l'`, and `'o'`, each accessible by their index position (0, 1, 2, 3, 4 respectively).

Understanding strings as collections will help you see the similarities between string operations and the list operations we'll explore later in this chapter.

5.1.1 String Concatenation

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. Like numerical addition, `+=` can be used if the same variable is used to store concatenation done on the *right* side of a string.

```
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_string2 += " I'm doing fine!"
8  print(my_string2)
9
10 my_age = 15
11 my_string3 = my_string2 + " I am " + str(my_age) + " years old!"
12 print(my_string3)
```

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

Python 3.6 and above support **f-strings** to simplify string concatenations of both constants and variables. An `f` is attached before the opening quotation mark to initiate an f-string. Unlike the previous method, non-string variables do not require conversion to a string. F-strings are perceived as a string type.

```
my_age = 15
my_string = f"I am {my_age} years old!"
print(my_string)
print(type(my_string))
```

```
I am 15 years old!
<class 'str'>
```

5.1.2 String Traversal

Strings can be traversed by characters using a for loop, similar to how we'll see lists can be traversed by elements.

```
my_string = "Hello"
for letter in my_string:
    print(letter)
```

```
H
e
l
l
o
```

Like we'll see with lists, `enumerate` can be used to obtain both index and character of a string per cycle.

5.1.3 Substrings

Like other collections, one can obtain a character from a string using its corresponding index. This indexing behavior mirrors what we'll see with list elements.

A **substring** is a string consisting of some characters from another string, comparable to how sublists contain some items of other lists (which we'll explore later).

Substrings can be obtained using indexing at a character level, using the same slicing syntax we'll see for lists.

```
my_string = "Hello world!"
print(my_string[7])
print(my_string[:5])
print(my_string[6:11])
```

```
o
Hello
world
```

5.1.4 Useful Built-in Functions

There are several built-in functions (see section 6 of page 20 to learn about properties of functions) for strings. Below are commonly used string functions:

- `len(<string>)` returns the **length** of a string (number of characters).

```
1 my_string = "Hello"
2 print(len(my_string))
```

```
5
```

- `<string>.upper()` converts all characters to uppercase.

```
1 my_string = "Hello World"
2 print(my_string.upper())
```

```
HELLO WORLD
```

- `<string>.lower()` converts all characters to lowercase.

```
1 my_string = "Hello World"
2 print(my_string.lower())
```

```
hello world
```

- `<substring> in <string>` checks whether the string contains a specific substring.

```
1 my_string = "Hello World"
2 if "World" in my_string:
3     print("Found 'World' in the string!")
```

```
Found 'World' in the string!
```

5.2 Lists

A **list** is an array consisting of data in a specific order. Like strings, 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. As with strings, indices are 0-based; the first element 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 may contain items of different data types, including lists.

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

5.2.1 List Traversal

Like strings, lists are iterables, so one can use for-loops 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.

If you need access to each element's index per cycle, you can use `enumerate` just like with strings:

```
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`.

5.2.2 Sublists

A sublist can be retrieved by using a range of indices, using the same slicing syntax we saw for substrings.

```
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']
```

An empty start or end index will automatically include all elements from beginning or to end, respectively.

```
print(my_list[:3])
print(my_list[2:])
print(my_list[::-1])
```

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

5.2.3 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.

5.2.4 Useful Built-in Functions

There are several built-in functions (see section 6 of page 20 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.

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

```
4
```

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

```
1 my_list = ["a", "b", "c", "d"]
2 my_list.append("e")
3 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.

```
1 my_list = ["a", "c", "d"]
2 my_list.insert(1, "b")
3 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.

```
1 my_list = ["a", "b", "c", "d"]
2 my_list.pop(0)
3 print(my_list)
4 my_list.pop()
5 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".
```

Note that there are many more built-in functions, so it's strongly recommended to search for the function not listed in this guide based on your specific needs.

5.3 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.

5.4 Dictionaries

Lists have their elements ordered via their index system. On the other hand, a **dictionary**–a.k.a. *hashmap* or *map*–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. Just like the indices of all elements in a list are unique, every key in a dictionary should also be unique. 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.

```
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:

```
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)
```

While `keys()`, `values()`, and `items()` return iterables, they are not lists. However, they can be converted to lists via `list()`.

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

keys = my_dict.keys()
print(type(keys), keys)

key_list = list(keys)
print(type(key_list), key_list)
```

```
<class 'dict_keys'> dict_keys(['Andrew', 'Brian', 'Charlie'])
<class 'list'> ['Andrew', 'Brian', 'Charlie']
```

5.4.1 Useful Built-in Functions

`len`, `pop`, and `in` functions used for strings and lists (see section 5.2.4) can be applied for dictionaries. Note that the `pop` function takes the *key* of a pair to remove an item from the dictionary (unlike lists, not including the key argument will result in error, but it is possible to set a "default" value to return if key is not found), while the `in` function asserts whether a *key* exists in the dictionary.

Sometimes, one cannot expect that a certain key exists and want to handle cases of getting a value without throwing errors. In such cases, use `get(<key>, <default_value=None>)`, which will work in the similar fashion as the traditional get method using `[<key>]`, except when the requested key does not exist, it will return `None` or the default value, if specified.

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

print(len(my_dict))

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

print(my_dict.pop("Daniel", -1)) # Since "Daniel" is not a key in my_dict, the "
    default" value -1 is returned.

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

print(my_dict.get("Daniel"))
print(my_dict.get("Daniel", 0))
```

```
3
5
{'Andrew': 8, 'Charlie': 10}
-1
True False
None
0
```

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.

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

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

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

5.5 Sets

A **set**, a.k.a. *hashset*, contain elements that are distinct from each other. Therefore, using a set is preferred if all elements have to be unique when adding elements to it. Like dictionary keys, tuples and other immutable data can be stored in sets.

```
my_set = set()
print(type(my_set))

my_set2 = {1, 2, 3, 4, 1}
print(type(my_set2), my_set2) # Prints the type and itself.
```

```
<class 'set'>
<class 'set'> {1, 2, 3, 4}
```

Since sets are unordered, and each item does not have unique keys, accessing a single item in a set is not possible. However, one can iterate through a set via a for-loop (consistent order of iterations is not guaranteed) and confirm whether a specific value is contained inside a set.

```
my_set = {1, 2, 3, 4, 1}
print(my_set)

for num in my_set:
    print(num)

print(2 in my_set)
print(5 in my_set)
```

```
{1, 2, 3, 4}
1
2
3
4
True
False
```

5.5.1 Useful Built-in Functions

`add()` and `remove()` will add a value to a list or remove from it, respectively. Remember that sets keep all its elements unique, so adding an existing value will do nothing to it (but won't throw an error). Note that attempting to remove a value that does not exist in a set will throw an error.

```
my_set = {1, 2, 3, 4}
print(my_set)

my_set.add(5)
print(my_set)

my_set.add(3)
print(my_set)

my_set.remove(3)
print(my_set)
```

```
{1, 2, 3, 4}
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
{1, 2, 4, 5}
```

Like mathematical sets, there are built-in set methods that do operations such as creating a union or intersection of two sets, or checking if one set is a superset or subset of another set. Search them up for more details about them!

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 concepts can be applied to Python, executing a consistent algorithm incorporating inputs and outputs. Functions are also known as **methods**. It is similar to the custom blocks (My Blocks) in Scratch.

6.1 Structure

To use a function, one should construct it by defining what inputs it takes, how it works, and what it outputs. It is similar to the "Define" blocks in Scratch.

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 22). 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 generally 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:
```

Like argument type setters, this is optional, but it is also generally advised to make functions have consistent return types.

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.

If a function returns multiple values at a time in form of a tuple, each item of the tuple can be assigned to independent variables like such:

```
<ret_var_1>, <ret_var_2>..., <ret_var_n>
= <func_name>(<param>...)
```

where `n` is the number of items in the tuple that is returned.

Below shows the callers of the previously defined example functions:

```
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
```

Below shows an example function that solves quadratic equations and returns two values at a time:

```
def solve_quadratic(a, b, c):
    discriminant = b**2 - 4*a*c
    if discriminant < 0:
        return None, None  # No real roots
    root1 = (-b + discriminant ** (1 / 2)) / (2 * a)
    root2 = (-b - discriminant ** (1 / 2)) / (2 * a)
    return root1, root2

x1, x2 = solve_quadratic(1, -3, 2)
print("Roots:", x1, x2)
```

```
Roots: 2.0 1.0
```

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`), or simply nothing (`return`). Any attempts to retrieve the "return value" of a void function will result in `None`.

```
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.
    if age >= 16:
        print("I can drive a car under supervision.")
        return      # Exits from the function.
    print("I can't drive a car.")

introduce("Andrew", 25)
return_value = introduce("Brian", 14)
print(return_value)
```

```
Hello, my name is Andrew.
I can drive a car.
Hello, my name is Brian.
I can drive a car under supervision.
Hello, my name is Charlie.
I can't drive a car.
None
```

7 More Tips for Python

7.1 More Built-in Functions

We have learned about "freelance" built-in functions like `print()` and `range()`. Below are more essential built-in functions for Python programming.

7.1.1 `input()`

`input()` allows the user to insert a piece of text as input to the program, which is perceived as a string. It is helpful for building programs that need user interaction. It takes one string argument to be displayed as an instruction for the user, where the user input is formed in the same line as the instruction.

```
user_input = input("Type your name: ")
print("Hello, " + user_input + "!")
```

```
Type your name: Gary
Hello, Gary!
```

Note that "Gary" is typed by the user while running the code.

7.2 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 three 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.

7.2.1 `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 5.4 in page 16).

```
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
```

where each rerun of above will result in different outputs.

7.2.2 `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.

```
import math

print(math.sqrt(16))
print(math.sqrt(10.6))

print(math.inf)
if math.inf > 1e32:
    print("Greater")

print(math.pi)
```

```
4.0
3.255764119219941
inf
Greater
3.141592653589793
```

7.2.3 `time`

Python's `time` module provides tools to provide information related to time.

- `time.time()`
  Returns current time in seconds in float since the epoch. In Unix systems, the epoch is set at January 1, 1970, 00:00:00 UTC.
- `time.ctime(time: int/float)`
  Converts the seconds since epoch like the one above into the format `Day Mon DD HH:MM:SS Year`.
- `time.sleep(second: int/float)`
  Holds the code from continuing for the set seconds.

```
import time

curr_time = time.time()
print(curr_time, type(curr_time))

date_time = time.ctime(curr_time)
print(date_time, type(date_time))

time.sleep(5)
# Below code runs 5 seconds later.
print("Hi! It's been 5 seconds!")
```

```
1744389965.6022182 <class 'float'>
Fri Apr 11 16:46:05 2025 <class 'str'>
Hi! It's been 5 seconds!
```

7.3 Common Mistakes Among Beginners

Here are some common mistakes that beginners make which lead to errors or conflicts from code reviews.

7.3.1 Single vs Double =

While both symbols have similar implications, misusing them will lead to compilation errors. My advice for understanding the differences is to remember that:

- `=`
  assigns right-hand-side to left-hand-side. The value on the right-hand-side of `=` is saved to the variable specified on the left-hand-side.
- `==`
  compares left-hand-side to right-hand-side. The compiler checks whether the represented values on the left-hand-side and right-hand-side are equal.

7.3.2 : and Indentation

Python, an indentation-centric programming language, differs from other languages in that its function and if-statement bodies are not enclosed in brackets that serve as markers. Instead, Python uses indentation to indicate the boundaries of these blocks. Fortunately, if you've been following the indentation pattern in other languages, you'll find a similar pattern in Python.

Notice the `:` at the end of if-statements, else syntax, while and for-loop headers, and function headers. This marker indicates the beginning of the body, which should be encapsulated via indentation.

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). A separate guide about OOP is currently in progress and will be released soon in `https://github.com/hyosang2/Intro-to-Python`.