Introductory Python Review Guide

Updated February 11, 2026

Contents

Prologue

This guide provides a concise overview of fundamental Python programming topics, including variables, data types, control flow, strings, lists, functions, and more. It will help beginners solve problems on CodingBat and start creating simple programs 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

Welcome to Python! Python is one of the most popular programming languages in the world, and it's a great first language to learn. In this guide, you'll learn everything you need to start writing your own programs.

1.1 Getting Started

To write and run Python code, you can use a free online editor like `replit.com` or `python.org/shell`. Just type your code and hit the Run button. If Python is installed on your computer, you can also save your code in a `.py` file and run it from the terminal with `python my_file.py`.

1.2 Print Statements

The first step in programming is to display something on the screen. We do this with a **print statement**:

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

Output:

```
Hello World!
```

Think of `print()` as a way to make the computer talk to you. Whatever you put inside the parentheses and quotation marks will appear on the screen. Print statements are useful for showing results and figuring out what your code is doing.

You can print multiple things by separating them with commas:

```
print("My name is", "Alice")
print("I am", 10, "years old")
```

Output:

```
My name is Alice
I am 10 years old
```

1.3 Comments

Throughout this guide, you'll see text that follows a # symbol. These are **comments**. Comments don't do anything when the code runs — they're notes that describe what the code does.

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

Output:

```
Hello World!
```

Comments are helpful for:

- Explaining what your code does
- Adding notes for future reference
- Temporarily disabling a line of code without deleting it
- Making your code easier to read for others

1.4 Input

What if you want the user to type something? The `input()` function lets you ask a question and save the answer. It's like the "ask and wait" block in Scratch.

```
name = input("What is your name? ")
print("Hello, " + name + "!")
```

Output:

```
What is your name? Gary
Hello, Gary!
```

In the example above, `"Gary"` is typed by the user while the program is running. The `input()` function always gives you text (a string), even if the user types a number. We'll learn how to convert between types in the next chapter.

Try it yourself! Write a program that asks for the user's name and favorite color, then prints a sentence using both.

2 Variables and Data Types

A **variable** is like a labeled box that stores a value. You give the box a name, put something inside it, and can look at or change what's inside later. It's similar to the "set my variable to" block in Scratch.

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

print(x)
print(y)
```

Output:

```
7
Hello World!
```

In the code above, the `=` sign **assigns** the value on the right side to the variable name on the left side. After these lines, `x` holds the value `7` and `y` holds the value `"Hello World!"`.

2.1 Naming Variables

By convention, variable names use lowercase letters and numbers, separated by underscores (_). For example: `my_score`, `player_name`, `total_count`. **Names that use characters other than letters, numbers, and underscores — or that start with a number — are not allowed.**

2.2 Data Types

Python has several **data types** — different kinds of values that variables can hold:

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

Here's what each type means:

- **Strings** are text — a group of characters like letters, numbers, and punctuation. You create a string by putting text inside quotation marks (single `'...'` or double `"..."`).
- **Integers** are whole numbers, both positive and negative, **without** a decimal point.
- **Floats** (floating-point numbers) are numbers, both positive and negative, **with** a decimal point.
- **Booleans** are either `True` or `False` — like a yes/no answer. The name "Boolean" comes from the mathematician George Boole.

Integers, floats, and Booleans are simple types — each variable holds a single value. Strings, on the other hand, are sequences of characters, and we'll explore them in depth in 5.

2.3 Type Conversion

Sometimes you need to change a value from one type to another. Python gives you built-in functions for this:

```
# String to integer
age_text = "15"
age_number = int(age_text)
print(age_number + 1)

# String to float
pi_text = "3.14"
pi_number = float(pi_text)
print(pi_number)

# Number to string
score = 100
score_text = str(score)
print("Your score is " + score_text)

# Float to integer (cuts off the decimal --- does NOT round)
x = int(3.9)
print(x)
```

Output:

```
16
3.14
Your score is 100
3
```

Type conversion is especially useful with `input()`, since `input()` always returns a string:

```
age = int(input("How old are you? "))
print("Next year you'll be", age + 1)
```

Output:

```
How old are you? 10
Next year you'll be 11
```

You can check what type a value is using `type()`:

```
x = 5
print(type(x))
y = "hello"
print(type(y))
```

Output:

```
<class 'int'>
<class 'str'>
```

2.4 Operators

Python supports a series of math operators:

```
x = 9
y = 2

sm = x + y            # Sum: 11
diff = x - y          # Difference: 7
prod = x * y          # Product: 18
quo = x / y           # Quotient: 4.5
intquo = x // y       # Floored Quotient: 4
mod = x % y           # Modulus (Remainder): 1
pwr = x ** y          # Power (x to the y): 81

print(sm, diff, prod, quo, intquo, mod, pwr)
```

Output:

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

There are two kinds of division: `/` gives the full decimal answer (a float), while `//` gives the answer rounded **down** to a whole number (called **floor division**).

The **modulus** operator `%` gives you the **remainder** after division. For example, `9 % 2` is `1` because 9 divided by 2 is 4 with a remainder of 1. This is very handy for checking whether a number is even or odd: if `n % 2 == 0`, the number is even.

2.4.1 Assignment Operator Shortcuts

If you want to update a variable using its current value, you can use shortcut operators. For example, instead of writing `x = x + y`, you can write `x += y`:

Long Form	Short Form
`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/Elif/Else Statements

Most decision-making in Python uses **if-statements**. It's like the "if-then-else" block in Scratch. To create an if-statement, follow these steps:

1. Write the `if` keyword
2. Define the **condition** to check
3. Write the **body** — the code that runs if the condition is true

The body is defined by **indenting** a block of lines (usually 4 spaces). These lines only run if the condition is true. A condition is true when it simplifies to the Boolean value `True`.

By adding the optional `else` keyword, you can define a second body that runs only if the condition is false.

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

Output:

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

This prints `x is less than or equal to y.` because the condition `x > y` evaluates to `False`.

3.1 Elif Statements

The optional `elif` keyword (short for "else if") lets you check a second condition when the first one is not satisfied:

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

Output:

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

Once a condition is met, Python will **skip all the remaining `elif` and `else` branches** below it. You can chain as many `elif` branches as you need.

3.2 Comparison Operators

These are the symbols used to compare values in Python:

Python Syntax	Meaning
`x == y`	$x = y$ (equal to)
`x != y`	$x \neq y$ (not equal to)
`x > y`	$x > y$ (greater than)
`x < y`	$x < y$ (less than)
`x >= y`	$x \geq y$ (greater than or equal to)
`x <= y`	$x \leq y$ (less than or equal to)

Note that the double equal sign (`==`) **compares** two values to check if they're equal, while the single equal sign (`=`) **assigns** a value to a variable. Mixing these up is a very common beginner mistake!

3.3 Logical Operators: `and`, `or`, `not`

You can combine conditions using `and`, `or`, and `not` to create **compound conditions**:

- `a and b` is true if **both** `a` and `b` are true.
- `a or b` is true if **at least one** of `a` or `b` is true.

- `not a` is true if `a` is false (it flips the result).

```
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 a discount.")

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

Output:

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

When mixing operators, `not` is evaluated first, then `and`, then `or`. Parentheses can be used to change the order, just like in math. To practice this, try building **truth tables** for compound conditions.

3.4 Nested If-Statements

Sometimes you need to check a second condition only after the first one passes. You can put an if-statement *inside* another if-statement — this is called **nesting**.

```
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.")
```

Output:

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

You can nest as many layers of if-statements as you need, but try to keep it readable. Often, compound conditions with `and`/`or` can replace deeply nested code.

4 Loops

Sometimes you want certain code to run multiple times. Maybe each cycle is slightly different, but you notice a repeating pattern. That's when you use **loops**.

4.1 While-Loops

Recall if-statements: the body runs only if the condition is true. If you replace the `if` keyword with `while`, the body will **repeat** as long as the condition stays true. It stops once the condition becomes false.

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

Output:

```
0
1
2
3
4
```

This code prints the value of `i`, starting at 0, adds 1 to it, and repeats until `i` reaches 5 (at which point `i < 5` is false, so the loop stops). It's similar to the "repeat until" block in Scratch, except Scratch's version stops when the condition becomes true, while Python's while-loop repeats *while* the condition is true.

Think about it: How can you make a loop that repeats forever, like the "forever" block in Scratch? Hint: what condition is *always* true?

4.2 For-Loops

Unlike while-loops that check a condition each time, for-loops **go through** a sequence of items and process one item at a time. Data that you can loop through is called **iterable** — meaning "something you can go through one item at a time." Examples include a range of numbers, the characters of a string, and lists.

```
1 for i in range(5):
2     print(i)
3
4 for letter in "Python":
5     print(letter)
```

Output:

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

The first for-loop behaves the same as the while-loop example above. `range(5)` gives you the numbers 0 through 4, and `i` takes on each number in order. The second for-loop goes through each character in the string `"Python"`, one at a time.

4.2.1 The `range()` Function

`range()` gives you a sequence of numbers. It can be called in three ways:

```
range(stop)                 # Numbers from 0 up to (but not including) stop
range(start, stop)          # Numbers from start up to (but not including) stop
range(start, stop, step)    # Same, but counting by step instead of 1
```

All values must be integers. The `stop` value is always **excluded** from the range.

```
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
```

To count *backwards*, use a negative `step` and make `start` larger than `stop`.

4.2.2 The `enumerate()` Function

Sometimes you need both the **index** (position number) and the **value** of each item while looping. The `enumerate()` function gives you both:

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

Output:

```
0 apple
1 banana
2 cherry
```

Here, `i` gets the index (0, 1, 2) and `fruit` gets the value at that index. This works with any iterable — strings, lists, and more.

4.3 Nested Loops

Just like nested if-statements, you can put a loop *inside* another loop. This is useful when you need to repeat a pattern within a pattern, such as going through the rows and columns of a grid.

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

Output:

```
0 0
0 1
0 2
0 3
0 4
1 0
1 1
...
```

The outer loop runs 3 times, and for each of those, the inner loop runs 5 times — giving you a total of 15 `print` statements. A loop moves to the next cycle only after finishing all the code in its body, including any inner loops.

4.4 `break` and `continue`

The keywords `break` and `continue` give you extra control inside loops:

- `break` exits the loop entirely.
- `continue` skips the rest of the current cycle and jumps to the next one.

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

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

Output:

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

The first loop prints `i` up to 4 because it exits when `i` reaches 5. The second loop skips odd numbers (when `i % 2 == 1`) and only prints the even ones.

4.5 The Accumulator Pattern

One of the most important patterns in programming is **building up a result inside a loop**. You start with an empty result (like `0` for a sum, `""` for a string, or `[]` for a list), then add to it on each cycle. This is called the **accumulator pattern**.

4.5.1 Accumulating a Sum

```
total = 0
for num in [10, 20, 30, 40]:
    total += num
print(total)
```

Output:

```
100
```

4.5.2 Building a String

```
result = ""
for letter in "Hello":
    result += letter + letter
print(result)
```

Output:

```
HHeellllloo
```

4.5.3 Building a List

```
evens = []
for num in [1, 2, 3, 4, 5, 6]:
    if num % 2 == 0:
        evens.append(num)
print(evens)
```

Output:

```
[2, 4, 6]
```

This pattern shows up everywhere in programming — in CodingBat problems, real-world projects, and beyond. Whenever you need to process a sequence and collect results, think "accumulator pattern."

Try it yourself! Write a loop that adds up all the numbers from 1 to 100.

5 Strings

Recall from 2 that strings are text enclosed in quotation marks. While they look like simple values, strings are actually **collections of characters**. Each character sits at a specific position called an **index**, starting at 0. This makes strings one of Python's most feature-rich data types.

For example, the string `"Hello"` contains five characters: `'H'` at index 0, `'e'` at index 1, `'l'` at index 2, `'l'` at index 3, and `'o'` at index 4.

5.1 String Concatenation

Strings can be **concatenated** (joined together) using the `+` operator. If you need to include a number in a string, convert it to a string first using `str()`.

```
greeting = "Hello " + "world!"
print(greeting)

greeting2 = greeting + " How are you?"
print(greeting2)

greeting2 += " I'm great!"
print(greeting2)

age = 15
message = "I am " + str(age) + " years old!"
print(message)
```

Output:

```
Hello world!
Hello world! How are you?
Hello world! How are you? I'm great!
```

```
I am 15 years old!
```

5.1.1 F-Strings

Python 3.6 introduced **f-strings**, a much easier way to insert variables into strings. Put an `f` before the opening quotation mark, then use curly braces {} around any variable or expression:

```
name = "Alice"
age = 15
print(f"My name is {name} and I am {age} years old!")
print(f"Next year I'll be {age + 1}.")
```

Output:

```
My name is Alice and I am 15 years old!
Next year I'll be 16.
```

F-strings don't require `str()` conversion — you can put any expression inside the curly braces.

5.2 Indexing

You can access a single character from a string using its **index** in square brackets. Remember, indexing starts at 0!

```
my_string = "Python"
print(my_string[0])   # First character
print(my_string[3])   # Fourth character
print(my_string[-1])  # Last character
print(my_string[-2])  # Second to last character
```

Output:

```
P
h
n
o
```

Negative indices count from the end: `-1` is the last character, `-2` is the second to last, and so on. This is really handy when you need to grab characters from the end without knowing the string's length.

5.3 Slicing and Substrings

Slicing lets you grab a portion of a string. The result is called a **substring** — a string made up of some characters from another string. The general syntax is:

```
string[start:stop:step]
```

- `start`: the index to begin at (inclusive, defaults to 0)
- `stop`: the index to stop at (exclusive, defaults to end of string)
- `step`: how many characters to skip (defaults to 1)

```
my_string = "Hello world!"
print(my_string[0:5])      # Characters from index 0 to 4
print(my_string[6:11])     # Characters from index 6 to 10
print(my_string[:5])       # From the start to index 4
print(my_string[6:])       # From index 6 to the end
print(my_string[::2])      # Every other character
print(my_string[::-1])     # The entire string, reversed!
```

Output:

```
Hello
world
Hello
world!
Hlowrd
!dlrow olleH
```

The trick `[::-1]` reverses any string — it's a useful pattern to remember.

5.4 String Traversal

You can loop through a string character by character:

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

Output:

```
P
y
t
h
o
n
```

If you also need the index, use `enumerate()`:

```
for i, letter in enumerate("Cat"):
    print(f"Index {i}: {letter}")
```

Output:

```
Index 0: C
Index 1: a
Index 2: t
```

5.5 String Built-in Functions

Here are the most commonly used string functions and methods:

5.5.1 Length

`len(string)` returns the number of characters in a string.

```
print(len("Hello"))     # 5
print(len(""))          # 0
print(len("Hi there")) # 8 (the space counts!)
```

5.5.2 Changing Case

```
my_string = "Hello World"
print(my_string.upper())  # "HELLO WORLD"
print(my_string.lower())  # "hello world"
```

5.5.3 Checking Content

- `substring in string` checks if a string contains another string.
- `.startswith(s)` checks if the string starts with `s`.
- `.endswith(s)` checks if the string ends with `s`.
- `.isdigit()` checks if every character is a digit.

```
my_string = "Hello World"
print("World" in my_string)          # True
print("world" in my_string)          # False (case-sensitive!)
print(my_string.startswith("Hello")) # True
print(my_string.endswith("World"))   # True
print("12345".isdigit())             # True
print("12.5".isdigit())              # False
```

5.5.4 Finding and Counting

- `.find(sub)` returns the index of the first occurrence of `sub`, or `-1` if not found.
- `.count(sub)` counts how many times `sub` appears.

```
sentence = "the cat sat on the mat"
print(sentence.find("cat"))    # 4
print(sentence.find("dog"))    # -1
print(sentence.count("the"))   # 2
print(sentence.count("at"))    # 3
```

5.5.5 Replacing

`.replace(old, new)` replaces all occurrences of `old` with `new` and returns a new string.

```
sentence = "I like cats and cats like me"
new_sentence = sentence.replace("cats", "dogs")
print(new_sentence)  # "I like dogs and dogs like me"
```

5.5.6 Splitting and Joining

- `.split(sep)` splits a string into a list of parts. If no separator is given, it splits on spaces.
- `sep.join(list)` joins a list of strings into one string, with `sep` between each item.

```
sentence = "apple,banana,cherry"
fruits = sentence.split(",")
print(fruits)  # ['apple', 'banana', 'cherry']

words = ["Hello", "World"]
result = " ".join(words)
print(result)  # "Hello World"
```

5.5.7 Stripping Whitespace

`.strip()` removes extra spaces (and newlines) from the beginning and end of a string.

```
messy = "   Hello World   "
print(messy.strip())  # "Hello World"
```

There are many more string methods — whenever you need to do something specific with strings, try searching "Python string methods" online.

Try it yourself! Write a program that takes a sentence and counts how many words it has. (Hint: use `.split()`.)

6 Lists

A **list** is a collection of items **in a specific order**. Like Scratch lists, you can add, remove, and access items by position. Each item has an **index** (position number) starting at 0, just like strings.

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

print(my_list[0])    # First item
print(my_list[1])    # Second item
print(my_list[-1])   # Last item
print(my_list[-2])   # Second to last
```

```
1
2
3
2
```

A list can contain items of different types, including other lists:

```
mixed = [1, 2.4, "Coding", True, [2, 5]]
```

6.1 Getting and Setting Items

You can read an item (get) or change it (set) using its index:

```
my_list = [10, 20, 30]

# Get
print(my_list[0])  # 10

# Set
my_list[1] = 99
print(my_list)     # [10, 99, 30]
```

This is a key difference from strings: lists are **mutable**, meaning you can change their contents. Strings are **immutable** — you can't change a single character within a string.

6.2 Slicing and Sublists

Slicing works the same way as with strings — you can grab a portion of a list (called a **sublist**) using `[start:stop:step]`:

```
my_list = ["a", "b", "c", "d", "e"]
print(my_list[1:4])        # ['b', 'c', 'd']
print(my_list[:3])         # ['a', 'b', 'c']
print(my_list[2:])         # ['c', 'd', 'e']
print(my_list[::2])        # ['a', 'c', 'e']
print(my_list[::-1])       # ['e', 'd', 'c', 'b', 'a']
print(my_list[4:1:-1])     # ['e', 'd', 'c']
```

6.3 List Traversal

Since lists are iterable (you can loop through them), you can use a for-loop to visit each item:

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

```
1
2
3
```

If you also need each item's index, use `enumerate()`:

```
colors = ["red", "green", "blue"]
for i, color in enumerate(colors):
    print(f"Index {i}: {color}")
```

```
Index 0: red
Index 1: green
Index 2: blue
```

6.4 2D Lists

Sometimes a flat list isn't enough — for example, when representing a grid or table. A **2D list** is a list that contains other lists as its items:

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

print(grid[0][0])     # "a" (row 0, column 0)
print(grid[1][2])     # "f" (row 1, column 2)
```

To loop through a 2D list, use **nested for-loops** — one for the rows, one for the items in each row:

```
for i, row in enumerate(grid):
    for j, item in enumerate(row):
        print(f"({i},{j}) = {item}")
```

```
(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
```

You can build lists with three or more dimensions by adding more nested lists, but 2D is the most common.

6.5 List Built-in Functions

Here are the most commonly used functions and methods for lists:

6.5.1 Length

`len(list)` returns the number of items in a list.

```
print(len([10, 20, 30]))  # 3
print(len([]))             # 0
```

6.5.2 Adding Items

- `.append(item)` adds an item at the **end** of the list.
- `.insert(index, item)` adds an item at a specific position, pushing the rest forward.

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

my_list.insert(2, "c")
print(my_list)             # ['a', 'b', 'c', 'd', 'e']
```

6.5.3 Removing Items

- `.pop(index)` removes and returns the item at the given index. If no index is given, it removes the **last** item.
- `.remove(value)` removes the **first** item that matches the given value.

```
my_list = ["a", "b", "c", "d"]

my_list.pop(0)
print(my_list)      # ['b', 'c', 'd']

my_list.pop()
print(my_list)      # ['b', 'c']

animals = ["cat", "dog", "cat", "bird"]
animals.remove("cat")
print(animals)      # ['dog', 'cat', 'bird'] --- only the first "cat" is removed
```

6.5.4 Searching

- `value in list` checks whether the list contains at least one item with that value.
- `.index(value)` returns the index of the **first** occurrence of the value.
- `.count(value)` counts how many times the value appears in the list.

```
my_list = ["a", "b", "c", "b", "d"]

print("b" in my_list)         # True
print("z" in my_list)         # False
print(my_list.index("b"))     # 1 (the first "b")
print(my_list.count("b"))     # 2
```

6.5.5 Sorting and Reversing

- `.sort()` sorts the list **in place** (changes the original list).
- `.reverse()` reverses the list **in place**.
- `sorted(list)` returns a **new** sorted list, leaving the original unchanged.

```
numbers = [3, 1, 4, 1, 5, 9]

print(sorted(numbers))  # [1, 1, 3, 4, 5, 9] --- new list
print(numbers)           # [3, 1, 4, 1, 5, 9] --- original unchanged

numbers.sort()
print(numbers)           # [1, 1, 3, 4, 5, 9] --- original is now sorted

numbers.reverse()
print(numbers)           # [9, 5, 4, 3, 1, 1]
```

There are many more list functions available — it's always worth searching for "Python list methods" when you need something specific.

Try it yourself! Write a program that takes a list of numbers and prints only the ones that are greater than the average.

7 Tuples, Dictionaries, and Sets

Beyond strings and lists, Python has three more collection types that are useful in different situations.

7.1 Tuples

A **tuple** is similar to a list, but it's **immutable** — once you create it, you can't change its items or length. Tuples use parentheses `()` instead of square brackets `[]`.

```
my_tuple = (0, 0)
print(my_tuple)
print(my_tuple[0])  # Indexing works the same as lists

my_tuple = (0, 1)   # You can reassign the whole tuple...
print(my_tuple)

my_tuple[0] = 1     # ...but you can't change individual items!
```

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

When should you use a tuple instead of a list? Tuples are great for data that shouldn't change, like a pair of coordinates `(x, y)` or a date `(year, month, day)`. They also use slightly less memory than lists.

You can loop through tuples and use indexing/slicing just like lists:

```
colors = ("red", "green", "blue")
for color in colors:
    print(color)

print(colors[1:])  # ('green', 'blue')
```

7.2 Dictionaries

A **dictionary** stores data as **key-value pairs**. Instead of accessing items by their position number (index), you access them by a unique **key**. Think of it like a real dictionary: you look up a word (the key) to find its definition (the value).

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

# Access a value using its key
print(scores["Brian"])  # 5

# Change a value
scores["Brian"] = 9
print(scores["Brian"])  # 9
```

```
5
9
```

Dictionaries use curly braces {} with key-value pairs separated by colons `:`.

Since **Python 3.7+**, dictionaries maintain the order in which items were added. However, unlike lists, you access items by key rather than by integer index.

7.2.1 Keys and Values

The keys of a dictionary must be **immutable** types (strings, numbers, tuples) and must be **unique**. Values can be any type.

```
my_dict = {"David": 9,
           8: ["Hello", "Hi"],
           (1, 2): 2.4}

print(my_dict[(1, 2)])  # 2.4
```

7.2.2 Looping Through Dictionaries

You can loop through a dictionary's keys, values, or both:

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

# Loop through keys
for name in scores:
    print(name)

# Loop through values
for score in scores.values():
    print(score)

# Loop through key-value pairs
for name, score in scores.items():
    print(f"{name}: {score}")
```

```
Andrew
Brian
Charlie
8
5
10
Andrew: 8
Brian: 5
Charlie: 10
```

Note: `keys()`, `values()`, and `items()` return special view objects, not lists. If you need an actual list, wrap them with `list()`:

```
key_list = list(scores.keys())
print(key_list)  # ['Andrew', 'Brian', 'Charlie']
```

7.2.3 Dictionary Built-in Functions

Adding and removing items:

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

# Add a new key-value pair
scores["Daniel"] = 12
print(scores)
# {'Andrew': 8, 'Brian': 5, 'Charlie': 10, 'Daniel': 12}

# Remove a key-value pair and get the value back
removed = scores.pop("Brian")
print(removed)  # 5
print(scores)   # {'Andrew': 8, 'Charlie': 10, 'Daniel': 12}
```

Checking if a key exists:

```
print("Charlie" in scores)  # True
print("Brian" in scores)    # False
```

Note that `in` checks for **keys**, not values.

Safe access with `.get()`:

Sometimes you're not sure if a key exists. Using `[key]` would cause an error, but `.get(key, default)` returns the value if the key exists, or a default value if it doesn't:

```
scores = {"Andrew": 8, "Charlie": 10}

print(scores.get("Andrew"))      # 8
print(scores.get("Daniel"))      # None
print(scores.get("Daniel", 0))   # 0 (custom default)
```

Length: `len(my_dict)` returns the number of key-value pairs.

7.3 Sets

A **set** is a collection where every item is **unique** — no duplicates allowed. If you add an item that's already there, nothing happens. Sets also use curly braces {}, but without the colon : that dictionaries use.

```
my_set = {1, 2, 3, 4, 1}  # The duplicate 1 is removed
print(my_set)             # {1, 2, 3, 4}

empty_set = set()  # Use set(), not {} (which creates an empty dictionary)
print(type(empty_set))    # <class 'set'>
```

Sets are **unordered**, so you can't access items by index. But you can loop through them and check if a value exists:

```python
my_set = {1, 2, 3, 4}

for num in my_set:
    print(num)

print(2 in my_set)  # True
print(5 in my_set)  # False
```

7.3.1 Set Built-in Functions

- `.add(value)` adds a value to the set.
- `.remove(value)` removes a value (throws an error if not found).
- `.discard(value)` removes a value (does nothing if not found — safer than `.remove()`).

```python
my_set = {1, 2, 3, 4}

my_set.add(5)
print(my_set)       # {1, 2, 3, 4, 5}

my_set.add(3)       # Already exists --- no change, no error
print(my_set)       # {1, 2, 3, 4, 5}

my_set.remove(3)
print(my_set)       # {1, 2, 4, 5}
```

Sets also support mathematical operations like union, intersection, and difference — search "Python set operations" to learn more!

8 Functions

If you've studied algebra, you may be familiar with **functions** — they take an input, do something with it, and produce an output:

$$
\begin{aligned}
f(x) &= x + 6 \\
g(x, y) &= x + 2y \\
f(5) &= 11 \\
f(8) &= 14 \\
g(5, 3) &= 11 \\
g(13, 17) &= 47
\end{aligned}
$$

Python functions work the same way: they take inputs, run a consistent set of steps, and (optionally) return an output. Functions let you **reuse** code instead of writing the same thing over and

over. If you've used the "My Blocks" feature in Scratch, you already know the idea — functions are custom blocks that you define once and use anywhere.

8.1 Function Structure

To create a function, you **define** it with the `def` keyword:

```
def add_six(x):
    result = x + 6
    return result
```

A function definition has three parts:

1. **Header**: `def function_name(parameters):` — the name and what inputs it takes
2. **Body**: the indented code that runs when the function is called
3. **Return statement**: `return value` — the output that the function sends back

8.2 Parameters and Arguments

The variables listed in the function definition are called **parameters**. The actual values you pass in when calling the function are called **arguments**.

```
def greet(name):     # "name" is the parameter
    print(f"Hello, {name}!")

greet("Alice")       # "Alice" is the argument
greet("Bob")         # "Bob" is the argument
```

```
Hello, Alice!
Hello, Bob!
```

A function can have multiple parameters, or no parameters at all:

```
def add(a, b):
    return a + b

def say_hi():
    print("Hi!")
```

8.3 Return Statement

The `return` keyword sends a value back to wherever the function was called. Once Python reaches a `return` statement, it **immediately exits** the function — any code after it is skipped.

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

result = add_six(5)
print(result)          # 11
print(add_six(8))      # 14
```

A function can return multiple values, separated by commas. These come back as a **tuple**:

```python
def min_and_max(a, b, c):
    return min(a, b, c), max(a, b, c)

smallest, largest = min_and_max(5, 2, 8)
print(smallest, largest)  # 2 8
```

8.4 Calling Functions

When you **call** a function, you write its name followed by the arguments in parentheses. The return value can be stored in a variable or used directly:

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

# Store the result in a variable
answer = add_2y(5, 3)
print(answer)         # 11

# Use the result directly
print(add_2y(13, 17))  # 47
```

Here's a more interesting example — a function that solves quadratic equations using the quadratic formula $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$:

```python
def solve_quadratic(a, b, c):
    discriminant = b**2 - 4*a*c
    if discriminant < 0:
        return None, None  # No real solutions
    root1 = (-b + discriminant ** 0.5) / (2 * a)
    root2 = (-b - discriminant ** 0.5) / (2 * a)
    return root1, root2

x1, x2 = solve_quadratic(1, -3, 2)
print("Solutions:", x1, x2)  # Solutions: 2.0 1.0
```

8.5 Void Functions

Functions without a `return` statement are called **void functions**. They perform an action (like printing) but don't send a value back. If you try to capture the return value anyway, you'll get `None`.

```python
def introduce(name, age):
    print(f"Hello, my name is {name}.")
    if age >= 18:
        print("I can drive a car.")
        return  # Exits the function early (returns None)
    if age >= 16:
        print("I can drive under supervision.")
        return
    print("I can't drive yet.")

introduce("Andrew", 25)
result = introduce("Brian", 17)
print(result)
```

```
Hello, my name is Andrew.
I can drive a car.
Hello, my name is Brian.
I can drive under supervision.
None
```

8.6 Default Parameter Values

You can give parameters **default values** so they become optional when calling the function:

```python
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

greet("Alice")              # Uses default: "Hello, Alice!"
greet("Alice", "Goodbye")   # Override: "Goodbye, Alice!"
greet("Bob", "Hey")         # Override: "Hey, Bob!"
```

```
Hello, Alice!
Goodbye, Alice!
Hey, Bob!
```

Parameters with default values must come **after** parameters without defaults.

8.7 Variable Scope

Variables created inside a function are **local** — they only exist within that function and disappear when the function ends. Variables created outside functions are accessible everywhere, but modifying them inside a function requires special care.

```python
def my_function():
    secret = "I only exist inside this function"
    print(secret)

my_function()

# This would cause an error:
```

```
# print(secret)  # NameError: name 'secret' is not defined
```

This means you can safely use the same variable name in different functions without them interfering with each other:

```
def function_a():
    x = 10
    print("A:", x)

def function_b():
    x = 20
    print("B:", x)

function_a()  # A: 10
function_b()  # B: 20
```

The `x` in `function_a` and the `x` in `function_b` are completely separate variables.

9 Useful Built-in Functions and Libraries

Python comes with many built-in functions beyond `print()`, `len()`, and `range()`. This chapter covers the ones you'll use most often, plus three helpful **libraries** that extend Python's capabilities.

9.1 Essential Built-in Functions

9.1.1 `abs()` — Absolute Value

`abs()` returns the absolute value of a number (its distance from zero, always positive):

```
print(abs(-5))     # 5
print(abs(5))      # 5
print(abs(-3.14))  # 3.14
```

This is especially useful when you want to find how far apart two numbers are, regardless of order:

```
a = 10
b = 3
distance = abs(a - b)
print(distance)  # 7
```

9.1.2 `max()` and `min()` — Largest and Smallest

`max()` returns the largest value and `min()` returns the smallest. They work with individual values or with lists:

```
print(max(3, 7, 2))        # 7
print(min(3, 7, 2))        # 2

scores = [85, 92, 78, 95, 88]
print(max(scores))         # 95
print(min(scores))         # 78
```

9.1.3 `sum()` — Add Up a Collection

`sum()` adds up all the numbers in a list (or other iterable):

```
numbers = [1, 2, 3, 4, 5]
print(sum(numbers))  # 15

# Combine with len() to find the average
average = sum(numbers) / len(numbers)
print(average)  # 3.0
```

9.1.4 `sorted()` — Sort Without Changing the Original

We saw `sorted()` in the lists chapter. As a reminder, it returns a **new** sorted list and leaves the original unchanged:

```
names = ["Charlie", "Alice", "Bob"]
print(sorted(names))    # ['Alice', 'Bob', 'Charlie']
print(names)             # ['Charlie', 'Alice', 'Bob'] --- unchanged
```

9.1.5 `round()` — Round a Number

`round()` rounds a float to a given number of decimal places (default is 0):

```
print(round(3.14159))        # 3
print(round(3.14159, 2))     # 3.14
print(round(2.5))            # 2 (Python uses "banker's rounding")
```

9.1.6 Quick Reference Table

Function	What it does	Example
`abs(x)`	Absolute value	`abs(-5)` → `5`
`max(...)`	Largest value	`max(3, 7, 2)` → `7`
`min(...)`	Smallest value	`min(3, 7, 2)` → `2`
`sum(list)`	Sum of all items	`sum([1,2,3])` → `6`
`sorted(list)`	New sorted list	`sorted([3,1,2])` → `[1,2,3]`
`round(x, n)`	Round to `n` decimals	`round(3.14, 1)` → `3.1`
`len(x)`	Length/count	`len("Hi")` → `2`
`type(x)`	Data type	`type(5)` → `<class 'int'>`
`int(x)`	Convert to integer	`int("5")` → `5`
`float(x)`	Convert to float	`float("3.14")` → `3.14`
`str(x)`	Convert to string	`str(42)` → `"42"`
`bool(x)`	Convert to Boolean	`bool(0)` → `False`
`input(prompt)`	Read user input	`input("Name? ")` → string

9.2 Libraries

Libraries provide extra functions and tools that aren't built into Python by default. To use a library, you need to **import** it — usually at the top of your file:

```
import library_name

library_name.function_name()
```

Here are three commonly used libraries.

9.2.1 `random` — Random Numbers and Choices

The `random` library lets you work with random values:

- `random.randint(a, b)` returns a random integer from `a` to `b`, inclusive.
- `random.choice(sequence)` picks a random item from a list, tuple, or string.

```
import random

print(random.randint(1, 10))    # Random number from 1 to 10

my_list = ["apple", "banana", "cherry"]
print(random.choice(my_list))   # Random item from the list
```

Each time you run the code, you may get different results.

Note: `random.choice()` needs a **sequence** (something with a length and indexes, like a list or string). To pick a random key from a dictionary, convert the keys to a list first:

```
scores = {"Alice": 95, "Bob": 87}
random_name = random.choice(list(scores.keys()))
```

9.2.2 `math` — Mathematical Operations

The `math` library provides math functions and constants beyond basic arithmetic:

- `math.sqrt(x)` returns the square root of `x`.
- `math.pi` is the constant π (approximately 3.14159).
- `math.inf` represents infinity.

```
import math

print(math.sqrt(16))       # 4.0
print(math.sqrt(10.6))     # 3.255...
print(math.pi)             # 3.1415926535897793

if math.inf > 1000000:
    print("Infinity is bigger!")  # This will print
```

9.2.3 `time` — Working with Time

The `time` library provides tools for tracking and controlling time:

- `time.time()` returns the current time in seconds (since January 1, 1970).
- `time.ctime(seconds)` converts seconds into a readable date-time string.
- `time.sleep(seconds)` pauses the program for the given number of seconds.

```
import time

current = time.time()
print(current)              # Something like 1744389965.6

print(time.ctime(current)) # Something like "Fri Apr 11 16:46:05 2025"

print("Wait for it...")
time.sleep(3)
print("3 seconds later!")
```

These libraries only scratch the surface — Python has hundreds of libraries for everything from web development to data science. You can always search online for "Python [topic] library" to find the right tool.

10 Common Mistakes and Next Steps

10.1 Common Mistakes Among Beginners

Here are some of the most frequent mistakes that beginners make. Knowing about them ahead of time can save you a lot of debugging!

10.1.1 Single = vs. Double ==

This is the most common mix-up for new programmers:

- `=` is the **assignment** operator — it puts a value into a variable.
- `==` is the **comparison** operator — it checks whether two values are equal.

```
x = 5        # Assigns the value 5 to x
x == 5       # Checks if x equals 5 (True)
```

If you accidentally use `=` inside an if-statement, Python will give you an error:

```
# Wrong:
if x = 5:      # SyntaxError!

# Correct:
if x == 5:     # This works
    print("x is five")
```

10.1.2 Forgetting the : and Indentation

Python uses `:` at the end of lines that start a block (if-statements, loops, functions), and **indentation** to mark what's inside that block. Forgetting either one causes an error:

```
# Wrong --- missing colon:
if x > 5
    print("big")

# Wrong --- missing indentation:
if x > 5:
print("big")

# Correct:
if x > 5:
    print("big")
```

This applies to `if`, `elif`, `else`, `for`, `while`, `def`, and more.

10.1.3 Modifying a List While Looping Through It

Removing items from a list while you're looping through it can cause unexpected behavior because the indices shift as items are removed:

```
# Wrong --- may skip items or cause errors:
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num % 2 == 0:
        numbers.remove(num)

# Better --- build a new list:
numbers = [1, 2, 3, 4, 5]
odds = []
for num in numbers:
    if num % 2 != 0:
        odds.append(num)
print(odds)  # [1, 3, 5]
```

10.1.4 Off-by-One Errors

Remember that indices start at **0**, and `range(n)` goes from 0 to `n-1` (not `n`). Slicing with `[start:end]` includes `start` but **excludes** `end`. This is the most common source of "off-by-one" bugs:

```
my_list = ["a", "b", "c", "d"]
print(my_list[4])      # IndexError! Valid indices are 0, 1, 2, 3
print(my_list[0:2])    # ['a', 'b'] --- NOT ['a', 'b', 'c']
```

10.1.5 Forgetting to Convert Types

`input()` always returns a string. If you want to do math with user input, you must convert it first:

```
# Wrong --- this concatenates strings, not adds numbers:
age = input("Age: ")
print(age + 1)  # TypeError!

# Correct:
age = int(input("Age: "))
print(age + 1)  # Works!
```

10.1.6 Using a Variable Before Defining It

Python reads code from top to bottom. If you try to use a variable before assigning a value to it, you'll get a `NameError`:

```
print(x)  # NameError: name 'x' is not defined
x = 5
```

Always make sure a variable is defined *before* the line that uses it.

10.2 Next Steps

Congratulations — you've learned the fundamentals of Python! Here's what you can do next:

Build something fun: Try making a simple game (number guessing, rock-paper-scissors, quiz game), a calculator, or a program that generates random stories.

Learn Object-Oriented Programming: The next guide in this series, *Object-Oriented Programming for Python with Turtle and Pygame*, covers classes, objects, and building graphical programs.

Happy coding!

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 available in `https://github.com/hyosang2/Intro-to-Python`.