Python Textbook

Nicole Oni*

February 2026

Contents

*Using the w3schools tutorials.

1 Basics

1.1 Introduction

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server side),
- software development,
- mathematics,
- system scripting

Python can:

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

Python Syntax compared to other programming languages:

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loos, functions and classes. Other programming languages often use curly-brackets for this purpose.

Other:

- The extension for Python files is `.py`.
- The command line syntax for checking if Python is installed on your computer (and also to check the python version) is `python --version`.
- Use `exit()` to exit the Python command line interface.

1.2 Python Syntax

1.2.1 Python Indentation

- Indentation refers to the spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- Python uses indentation to indicate a black of code.

For example, Python will give you an error if you skip the indentation:

```
1 if 5 > 2:
2 print("Five is greater than two!")
```

Correct version:

```
1 if 5 > 2:
2     print("Five is greater than two!")
```

1.2.2 Python Statements

- A **computer program** is a list of "instructions" to be "executed" by a computer.
- In a programming language, these programming instructions are called **statements**.
- In Python, a statement usually ends when the line ends. You do *not* need to use a semicolon (;) like in many other programming languages (for example, Java or C).

Example:

```
1 print("Python is fun!")
```

Many Statements:

- Most Python programs contain many statements.
- The statements are executed one by one, in the same order as they are written.

Example:

```
1 print("Hello World!")
2 print("Have a good day.")
3 print("Learning Python is fun!")
```

Semicolons:

- Semicolons are optional in Python.

- You can write multiple statements on one line by separating them with ; but this is rarely used because it makes it hard to read.
- If you put two statements on the same line without a separator (newline or ; , Python will give an error.

Example:

```
print("Hello"); print("How are you?"); print("Bye bye!")
```

Best practice: Put each statement on its own line so your code is easy to understand.

1.2.3 Challenge

Inside the editor, complete the following steps:

1. Write a statement that prints "Hello World!"
2. Write a statement that prints "Have a good day."
3. Write a statement that prints "Learning Python is fun!"

Solution:

```
print("Hello World!")
print("Have a good day.")
print("Learning Python is fun!")
```

1.3 Python Output / Print

- The `print()` function can be used to display text or output values.
- You can use the `print()` function as many times as you want. Each cell prints text on a new line by default.
- Text in Python must be inside quotes. You can use either `"` double quotes or `'` single quotes.
- If you forget to put the text inside quotes. Python will give an error.

```
print("Hello World!")
print("This will work!")
print('This will also work!')
print(This will cause an error)
```

1.3.1 Print Without a New Line

- By default, the `print()` function ends with a new line.
- If you want to print multiple words on the same line, you can use the `end` parameter.

Example:

```
print("Hello World!", end=" ")
print("I will print on the same line.")
```

Note that we add a space after `end=" "` for better readability.

1.3.2 Print Numbers

- You can also use the `print()` function to display numbers.
- However, unlike text, we don't put numbers inside double quotes.
- You can also do maths inside the `print()` function.

Example:

```
print(3)
print(358)
print(2*5)
```

1.3.3 Mixing Text and Numbers

You can combine text and numbers in one output by separating them with a comma. For example:

```
print("I am", 35, "years old.")
```

1.3.4 Challenge

Inside the editor, complete the following steps:

1. Print the text **"I am"** and the number **25** in one print statement.

Solution:

```
print("I am", 25)
```

1.4 Comments

- Python has commenting capability for the purpose of in-code documentation.
- Uses of comments:
  - Comments can be used to explain Python code.
  - Comments can be used to make the code more readable.
  - Comments can be used to prevent execution when testing code.
- Comments start with a #, and Python will render the rest of the line as a comment.

- Comments can be placed at the end of a line, and Python will ignore the rest of the line.

Example:

```
# This is a comment
print("Hello, World!")

print("Hello, World!") # This is a comment
```

1.4.1 Multiline Comments

Option 1:

- Python does not really have a syntax for multiline comments.
- To add a multiline comment you could insert a # for each line.

Option 2:

- Or, not quite as intended, you can use a **multiline string**.
- Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it.
- As long as the string is not assigned to a variable, Python will read the code, but then ignore it, and you have made a multiline comment.

Example:

```
# This is a comment
# written in
# more than just one line

print("Hello, World!")

"""
This is a comment
written in
more than just one line
"""
```

1.4.2 Challenge

Inside the editor, complete the following steps:

1. Add a single-line comment that says **This is a comment**
2. Comment out the line `print("This should not run")` so it does not execute.
3. Add a multiline comment (using triple quotes) that says **This is a multiline comment**

Solution:

```
# This is a comment
# print("This should not run")
"""
This is
a multiline
comment
"""
```

1.5 Python Variables

- Variables are containers for storing data values.
- Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.

Example:

```
x = 5
y = "John"
print(x)
print(y)
```

Variables do not need to be declared with any particular *type* and can even change type after they have been set. For example:

```
x = 4 # x is of type int
x = "Sally" # x is now of type str
print(x)
```

1.5.1 Get the Type

You can get the data type of a variable with the `type()` function.
Example:

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

1.5.2 Other

- String variables can be declared either by using single or double quotes.
- Variable names are case-sensitive.

Example:

```
x = "John"
# is the same as
x = 'John'

a = 4
A = "Sally"
# A will not overwrite a
```

1.5.3 Variable Names

A variable can have a short name (like `x` and `y`) or a more descriptive name (`age, carname, total_volume`.
Rules for Python variables:

- A variable name must start with a letter or the underscore character.
- A variable name cannot start with a number.
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
- Variable names are case-sensitive (`age, Age` and `AGE` are three different variables).
- A variable name cannot be any of the Python keywords.

Legal variable names:

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
```

Illegal variable names:

```
2myvar= "John"
my-var = "John"
my var = "John"
```

Multi Words Variable Names:
Variable names with more than one word can be difficult to read. There are several techniques you can use to make them more readable:

- **Camel Case:** Each word, except the first, starts with a capital letter: `myVariableName = "John"`
- **Pascal Case:** Each word starts with a capital letter: `MyVariableName = "John"`
- **Snake Case:** Each word is separated by an underscore character: `my_variable_name = "John"`

1.5.4 Assign Multiple Values

Many Values to Multiple Variables:

- Python allows you to assign values to multiple variables in one line.
- Make sure the number of variables matches the number of values, or else you will get an error.

Example:

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

One Value to Multiple Variables:
And you can assign the *same* value to multiple variables in one line:

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

1.5.5 Unpack a Collection

If you have a collection of values in a `list`, `tuple` etc. Python allows you to extract the values into variables. This is called *unpacking*.

Example (unpack a list):

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)
print(y)
print(z)
```

1.5.6 Output Variables

- The `print()` function is often used to output variables.
- In the `print()` function, you output multiple variables, separated by a comma.
- You can also use the `+` operator to output multiple variables.
  - For strings, add a space at the end otherwise there will be no spaces between words.
  - For numbers, the `+` character works as a mathematical operator.
  - In the `print` function, when you try to combine a string and a number with the `+` operator, Python will give you and error.
- The best way to output multiple variables in the `print()` function is to separate them with commas, which even support different data types.

```
# Output multiple variables using a comma
x = "Python"
y = "is"
z = "awesome"
print(x, y, z)

# Output multiple variables using +
```

```
x = "Python"
y = "is"
z = "awesome"
print(x + y + z)

# Using the + character as a mathematical operator
x = 5
y = 10
print(x + y)

# Combining strings and numbers with the + operator
x = 5
y = "John"
print(x + y)
```

1.5.7 Global Variables

- Variables that are created outside of a function (as in all the previous examples) are known as **global variables**.
- Global variables can be used by everyone, both inside of functions and outside.

Example:
Create a variable outside of a function, and use it inside the function.

```
x = "awesome"

def myfunc():
    print("Python is" + x)

myfunc()
```

If you create a variable with the same name inside a function, this variable will be **local**, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

Example:
Create a variable inside a function, with the same name as the global variable.

```
x = "awesome"

def myfunc():
    x = "fantatic"
    print("Python is " + x)

myfunc()

print("Python is " + x)
```

The global keyword

- Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.
- To create a global variable inside a function, you can use the `global` keyword.

- If you use the `global` keyword, the variable belongs to the global scope.

Example:

```
def myfunc():
    global x
    x = "fantastic"

myfunc()

print("Python is " + x)
```

To change the value of a global variable inside a function, refer to the variable by using the `global` keyword. For example:

```
x = "awesome"

def myfunc():
    global x
    x = "fantastic"

myfunc()

print("Python is " + x)
```

1.5.8 Challenge

Inside the editor, complete the following steps:

1. Create a variable `x` and assign it the value **5**.
2. Create a variable `y` and assign it the value **"John"**.
3. Use the `type()` function to print the type of `x`.

Solution:

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

1.6 Python Data Types

1.6.1 Built-in Data Types

- In programming, data type is an important concept.
- Variables can store data of different types, and different types can do different things.
- Python has the following data types built-in by default, in these categories:

Text Type	`str`
Numeric Types	`int, float, complex`
Sequence Types	`list, tuple, range`
Mapping Type	`dict`
Set Types	`set, frozenset`
Boolean Type	`bool`
Binary Types	`bytes, bytearray, memoryview`
None Type	`NoneType`

1.6.2 Getting the Data Type

You can get the data type of any object by using the `type()` function.

```
x = 5
print(type(x))
```

Example	Data Type
`x = "Hello World"`	`str`
`x = 20`	`int`
`x = 20.5`	`float`
`x = 1j`	`complex`
`x = ["apple", "banana", "cherry"]`	`list`
`x = ("apple", "banana", "cherry")`	`tuple`
`x = range(6)`	`range`
`x = {"name": "John", "age": 36}`	`dict`
`x = {"apple", "banana", "cherry"}`	`set`
`x = frozenset({"apple", "banana", "cherry"})`	`frozenset`
`x = True`	`bool`
`x = b"Hello"`	`bytes`
`x = bytearray(5)`	`bytearray`
`x = memoeryview(bytes(5))`	`memoryview`
`x = None`	`NoneType`

1.6.3 Setting the Specific Data Type

If you want to specify the data type, you can use the following constructor functions:

Example	Data Type
`x = str("Hello World")`	`str`
`x = int(20)`	`int`
`x = float(20.5)`	`float`
`x = complex(1j)`	`complex`
`x = list(["apple", "banana", "cherry"])`	`list`
`x = tuple(("apple", "banana", "cherry"))`	`tuple`
`x = range(6)`	`range`
`x = dict(name="John" age=36)`	`dict`
`x = set({"apple", "banana", "cherry"})`	`set`
`x = frozenset({"apple", "banana", "cherry"})`	`frozenset`
`x = bool(5)`	`bool`
`x = bytes(5)`	`bytes`
`x = bytearray(5)`	`bytearray`
`x = memoeryview(bytes(5))`	`memoryview`

1.6.4 Challenge

Inside the editor, complete the following steps:

1. Create a variable `x` with the value **5**.
2. Create a variable `y` with the value **3.14**.
3. Create a variable `z` with the value **"Hello"**.
4. Print the data type of each variable using `type()`.

Solution:

```
x = 5
y = 3.14
z = "Hello"

print(type(x))
print(type(y))
print(type(z))
```

1.7 Python Numbers

- Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.
- Float, or "floating point number" is a number, positive or negative, containing one or more decimals.
- Float can also be scientific numbers with an "e" to indicate the power of 10.
- Complex numbers are written with "j" as the imaginary part.
- You can convert from one type to another with the `int()`, `float()` and `complex()` methods.

```python
# The three numeric types in Python
x = 1 # int
y = 2.8 # float
y = 35e3
z = 1j # complex

# To verify the type of object
print(type(x))
print(type(y))
print(type(z))

# Convert from one type to another
a = float(x)
b = int(y)
c = complex(x)
```

You cannot convert complex numbers into another number type.

1.7.1 Random Number

Python does not have a `random()` function to make a random number, but Python has a built-in module called `random` that can be used to make random numbers.

```python
import random
print(random.randrange(1,10)) # displays a random number from 1 to 9
```

1.7.2 Challenge

Inside the editor, complete the following steps:

1. Create a variable `x` with the value **5**.
2. Create a variable `y` with the value **3.14**.
3. Create a variable `z` with the value **2 + 3j**.
4. Print the type of each variable using `type()`.

Solution:

```python
x = 5
y = 3.14
z = 2 + 3j

print(type(x))
print(type(y))
print(type(z))
```

1.8 Python Casting

If you want to specify the data type of a variable, this can be done with casting. Python is an object-oriented language, and as such it uses classes to define data types, including its primitive types.

Casting in python is therefore done using constructor functions:

- `int()` - constructs an integer number from:
  - an integer literal
  - a float literal (by removing all decimals)
  - a string literal (providing the string represents a whole number)
- `float()` - constructs a float number from:
  - an integer literal
  - a float literal
  - a string literal (providing the string represents a float or an integer)
- `str()` - constructs a string from a wide variety of data types, including:
  - strings
  - integer literals
  - float literals

Example:

```
x = str(3) # x will be '3'
y = int (3) # y will be 3
z = float(3) # z will be 3.0

print(int(35.88)) # Output is 35
print(float(35)) # Output is 35.0
print(str(35.82)) # Output is 35.82
```

1.8.1 Challenge

Inside the editor, complete the following steps:

1. Create a variable `x` with integer value **1**.
2. Convert `x` to a float and store it in `a`.
3. Convert `x` to a string and store it in `b`.
4. Print `a` and `b`.

Solution:

```
# Create an integer
x = 1

# Convert to float
a = float(x)

# Convert to string
b = str(x)

# Print values
print(a)
print(b)
```

1.9 Python Strings

- Strings in python are surrounded by either single quotation marks, or double quotation marks.
- `'hello'` is the same as `"hello"`.
- You can display a string literal with the `print()` function.

Example:

```
print("Hello")
print('Hello')
```

1.9.1 Quotes Inside Quotes

You can use quotes inside a string, as long as they don't match the quotes surrounding the string.

Example:

```
print("It's alright")
print("He is called 'Johnny'")
print('He is called "Johnny"')
```

1.9.2 Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string.

Example:

```
a = "Hello"
print(a)
```

1.9.3 Multiline Strings

You can assign a multiline string to a variable by using three single/double quotes.

Example:

```
a = """Hello my name is Nicole,
I am 18 years old,
My birthday is June 2nd,
And I am a Gemini."""
print(a)
```

Note: in the result, the line breaks are inserted at the same position as in the code.

1.9.4 Strings are Arrays

- Like many other popular programming languages, strings in Python are arrays of unicode characters.
- However, Python does not have a character data type, a single character is simply a string with a length of 1.
- Square brackets can be used to access elements of the string.

Example:

```
# Get the character at position 1 (remember that the first character has the position 0)
a = "Hello, World!"
print(a[1]) # Output would be e
```

1.9.5 Looping Through a String

Since strings are arrays, we can loop through the characters in a string, with a `for` loop.

Example:

```
# Loop through the letters in the word "banana"
for x in "banana":
    print(x)
```

1.9.6 String Length

To get the length of a string, use the `len()` function.

Example:

```
a = "Hello, world!"
print(len(a)) # The len() function returns the length of a string - in this case, 13
```

1.9.7 Check String

- To check if a certain phrase or character is present in a string, we can use the keyword `in`.
- To check if a certain phrase or character is NOT present in a string, we can use the keyword `not in`.

Example:

```
txt = "The best things in life are free!"
if "free" in txt:
    print("Yes 'free' is present.")
if "expensive" not in txt:
    print("No, 'expensive' is NOT present.")
```

1.9.8 Slicing Strings

- You can return a rage of characters by using the slice syntax.
- Specify the start index and the end index, separated by a colon, to return a part of the string.

Example:

```
# Get the characters from position 2 to position 5 (not included)
b = "Hello, World!"
print(b[2:5]) # Output is llo
```

REMEMBER: The first character has index 0.

Slice From Start:
By leaving out the start index, the range will start at the first character.

Example:

```
# Get the characters from the start to position 5 (not included)
b = "Hello, World!
print(b[:5]) # Output is Hello
```

Slice To the End:
By leaving out the end index, the range will go to the end.

Example:

```
# Get the characters from position 2 all the way to the end
b = "Hello, World!"
print(b[2:]) # Output is llo, World!
```

Negative Indexing:
Use negative indexes to start the slice from the end of the string.

Example:

```
b = "Hello, World!"
print(b[-5:-2]) # Output is orl (remember does not include -2)
```

1.9.9 String Concatenation

- To concatenate, or combine, two strings you can use the + operator.
- To add a space between them, add a `" "`

Example:

```
a = "Hello"
b = "World"
c = a + b
d = a + " " + b
print(c) # Output is HelloWorld
print(d) # Output is Hello World
```

1.9.10 String Format

As we learnt in the Python Variables chapter, we **cannot** combine strings and numbers like this:

Example:

```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

But we can combine strings and numbers by using *f-strings* or the `format()` method.

F-Strings:

- F-strings were introduced in Python 3.6, and is now the preferred way of formatting strings.
- To specify a string as an f-string, simply put a `f` in front of the string literal, and add curly brackets `{}` as placeholders for variables and other operations.

Example:

```
age = 36
txt = f"My name is John, I am {age}"
print(txt)
```

Placeholders and Modifiers:

- A placeholder can contain variables, operations, functions, and modifiers to format the value.
- A placeholder can include a *modifier* to format the value.
- A modifier is included by adding a `:` followed by a legal formatting type, like `.2f` which means fixed point number with 2 decimals.
- A placeholder can contain Python code, like math operations.

Examples:

```
# Add a placeholder for the price variable
price = 59
txt = f"The price is {price} dollars"
print(txt) # Output is "The price is 59 dollars"

# Display the price with 2 decimals
txt = f"The price is {price:.2f} dollars"
print(txt) # Output is "The price is 59.00 dollars"

# Perform a math operation in the placeholder
txt = f"The price is {2 * price} dollars"
print(txt) # Output is "The price is 118 dollars"
```

1.9.11 Escape Characters

- To insert characters that are illegal in a string, use an escape character.
- An escape character is a \ followed by the character you want to insert.
- An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

```
txt = "We are the so-called "Vikings" from the North."
```

To fix this problem, use the escape character \":

```
txt = "We are the so-called \"Vikings\" from the North."
```

Other escape characters used in Python:

Code	Result
\'	Single Quote
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal Value
\xhh	Hex Value

Examples:

```
print('It\'s alright') # Output is It's alright
print("This will insert one \\ (backslash).") # Output is This will insert one \ backslash
print("Hello\nWorld!") # Output is Hello (new line) World!
print("Hello\rWorld!") # Output is World!
print("Hello\tWorld!") # Output is Hello    World!


# This example erases one character (backspace)
print("Hello \bWorld!") # Output is HelloWorld!

print("Hello\fWorld!") # Output would be Hello (top of next page) World!

# A backslash followed by three integers will result in an octal value
print("\110\145\154\154\157") # Output is Hello

# A backslash followed by an 'x' and a hex number represents a hex value
print("\x48\x65\x6c\x6c\x6f") # Output is Hello
```

1.9.12 String Methods

Python has a set of built-in methods that you can use on strings.

Note: All string methods return new values. They do not change the original string.

Method	Description
`capitalize()`	Converts the first character to upper case
`casefold()`	Converts string into lower case
`center()`	Returns a centered string
`count()`	Returns the number of times a specific value occurs in a string
`encode()`	Returns an encoded version of the string
`endswith()`	Returns true if the string ends with the specified value
`expandtabs()`	Sets the tab size of the string
`find()`	Searches the string for a specified value and returns the position of where it was found
`format()`	Formats specified values in a string
`format_map()`	Formats specified values in a string
`index()`	Searches the string for a specified value and returns the position of where it was found
`isalnum()`	Returns True if all characters in the string are alphanumeric
`isalpha()`	Returns True if all characters in the string are in the alphabet
`isascii()`	Returns True if all characters in the string are ascii characters
`isdecimal()`	Returns True if all characters in the string are decimals
`isdigit()`	Returns True if all characters in the string are digits
`isidentifier()`	Returns True if the string is an identifier
`islower()`	Returns True if all characters in the string are lower case
`isnnumeric()`	Returns True if all characters in the string are numeric
`isprintable()`	Returns True if all characters in the string are printable
`isspace()`	Returns True if all characters in the string are whitespaces
`istitle()`	Returns True if the string follows he rules of a title
`isupper()`	Returns True if all characters in the string are upper case
`join()`	Joins the elements of an iterable to the end of the string
`ljust()`	Returns a left justified version of the string
`lower()`	Converts a string into lower case
`lstrip()`	Returns a left trim version of the string
`maketrans()`	Returns a translation table to be used in translations
`partition()`	Returns a tuple where the string is parted into three parts
`replace()`	Returns a string where a specified value is replaced with a specified value
`rfind()`	Searches the string for a specified value and returns the last position of where it was found
`rindex()`	Searches the string for a specified value and returns the last position of where it was found
`rjust()`	Returns a right justified version of the string
`rpartition()`	Returns a tuple where the string is parted into three parts
`rsplit()`	Splits the string at the specified separator, and returns a list
`rstrip()`	Returns a right trim version of the string
`split()`	Splits the string at the specified separator, and returns a list
`splitlines()`	Splits the string at line breaks and returns a list
`startswith()`	Returns True if the string starts with the specified value
`strip()`	Returns a trimmed version of the string
`swapcase()`	Swaps cases,lower case becomes upper case and vice versa
`title()`	Converts the first character of each word to upper case
`translate()`	Returns a translated string
`upper()`	Converts a string into upper case

Examples:

```
a = "Hello, World!"
print(a.upper()) # Output is HELLO, WORLD!
print(a.lower()) # Output is hello, world!
print(a.replace("H", "J")) # Output is Jello, World!
print(a.split(",")) # Output is ['Hello', 'World!']

b = " Hello, World! "
print(a.strip()) #Output is "Hello, World!"
```

1.9.13 Challenge

Inside the editor, complete the following steps:

1. Create a variable `txt` with the value **"Hello, World!"**.
2. Print the characters from index **2** to **5** (slicing).
3. Print `txt` converted to **upper case**.
4. Create a variable `name` with the value **"Python"**.
5. Use a **f-string** to print **"I love Python"** using the `name` variable.

Solution:

```
# Create the variable
txt = "Hello, World!"

# Print characters from index 2 to 5
print(txt[2:5])

# Print in upper case
print(txt.upper())

# Create the name variable
name = "Python"

# Print using an f-string
print(f"I love {name}")
```

1.10 Python Booleans

- Booleans represents one of two values: `True` or `False`.
- In programming you often need to know if an expression is `True` or `False`.
- You can evaluate any expression in Python, and get one of two answers, `True` or `False`.
- When you compare two values, the expression is evaluated and Python returns the Boolean answer:

```
print(10 > 9) # Returns True
print (10 == 9) # Returns False
print (10 < 9) # Returns False
```

When you run a condition in an if statement, Python returns `True` or `False` :

```
# Print a message based on whether the condition is True or False
a = 200
b = 33

if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")
```

1.10.1 Evaluate Values and Variables

The `bool()` function allows you to evaluate any value, and give you `True` or `False` in return:

```
x = "Hello"
y = 15

print(bool(x)) # Returns True
print(bool(y)) # Returns True
```

1.10.2 Most Values are True

Almost any value is evaluated to `True` if it has some content:

- Any string is `True`, except empty strings.
- Any number is `True` , except `0` .
- Any list, tuple, set, and dictionary are `True`, except empty ones.

1.10.3 Some Values are False

There are not many values that evaluate to `False`, except empty values, such as:

- `()`
- `[]`
- `{}`
- The number `0`
- The value `False`
- The value `None`

Examples:

```
# Return True
bool("abc")
bool(123)
bool(["apple", "banana", "cherry"])

# Return False
bool(False)
bool(None)
bool(0)
bool("")
bool(())
```

An object that is made from a class with a `__len__` function that returns `0` or `False` evaluates to False. For example:

```
class myClass():
    def __len__(self):
        return 0

myobj = myClass()
print(bool(myobj)) # Returns False
```

1.10.4 Functions can Return a Boolean

You can create functions that return Boolean values and execute code based on the Boolean answer of the function.

Example:

```
def myFunction():
    return True
if myFunction():
    print("YES!")
else:
    print("NO!")
```

Python also has many built-in functions that return a Boolean value, like the `isinstance()` function, which can be used to determine if an object is of a certain data type.

Example:

```
x = 200
print(isinstance(x, int)) # Returns True
```

1.10.5 Challenge

Inside the editor, complete the following steps:

1. Print the result of `10 > 9`.
2. Print the result of `10 == 9`.
3. Print the result of `bool("Hello")`.

4. Print the result of `bool(0)`.

Solution:

```
print(10 > 9)
print(10 == 9)
print(bool("Hello"))
print(bool(0))
```

1.11 Python Operators

Operators are used to perform operations on variables and values.

- The + operator can be used to add two values.
- The + operator can also be used to add together a variable and two values, or two variables.

Examples:

```
sum1 = 100 + 50 # 150 (100 + 50)
sum2 = sum1 + 250 # 400  (150 + 250)
sum3 = sum1 + sum2 # 550 (150 + 400)
```

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

1.11.1 Arithmetic Operators

Arithmetic operators are used with **numeric values** to perform common mathematical operations:

Operator	Name	Example
`+`	Addition	`x + y`
`-`	Subtraction	`x - y`
`*`	Multiplication	`x * y`
`/`	Division	`x / y`
`%`	Modulus	`x % y`
`**`	Exponentiation	`x ** y`
`//`	Floor division	`x // y`

Example:

```
x = 15
y = 4

print(x + y) # 19
print(x - y) # 11
print(x * y) # 60
print(x / y) # 3.75 (returns a float)
print(x % y) # 3 (remainder)
print(x ** y) # 50625
print(x // y) # 3 (returns an integer, rounds DOWN to the nearest integer)
```

1.11.2 Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x|3
^=	x ^= 3	x = x^3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3
:=	print(x := 3)	x = 3 print(x)

The Walrus Operator:

- Python 3.8 introduced the := operator, known as the "walrus operator".
- It assigns values to variables as part of a larger expression.

Example:

```
numbers = [1, 2, 3, 4, 5]
if (count := len(numbers)) > 3:
    print(f"List has {count} elements") # Output is "List has 5 elements"
```

1.11.3 Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Comparison operators return `True` or `False` based on the comparison.

Example:

```
x = 5
y = 3
print(x == y) # Returns False
print(x != y) # Returns True
print(x > y) # Returns True
print(x < y) # Returns False
print(x >=y) # Returns True
print(x <= y) # Returns False
```

Chaining Comparison Operators:
Python allows you to chain comparison operators. For example:

```
x = 5
print(1 < x < 10) # Returns True
print(1 < x and x < 10) # Returns True
```

1.11.4 Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Examples:

```
x = 5

print(x > 0 and x < 10) # Returns True
print(x < 5 or x > 10) # Returns False
print(not(x > 3 and x < 10)) # Returns False
```

1.11.5 Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are NOT the same object	x is not y

Difference between `is` and `==`:

- `is` checks if both variables point to the same object in memory
- `==` checks if the values of both variables are equal

Examples:

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x

# The is operator returns True if both variables point to the same object
print(x is z) # Returns True
print(x is y) # Returns False
print(x == y) # Returns True

# The is not operator returns True if both variables do not point to the same
↪  object
print(x is not y) # Returns True
```

1.11.6 Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
`in`	Returns True if a sequence with the specified value is present in the object	`x in y`
`not in`	Returns True if a sequence with the specified value is NOT present in the object	`x not in y`

Examples:

```
fruits = ["apple", "banana", "cherry"]
print("banana" in fruits) # Returns True
print("pineapple" not in fruits) # Returns True
```

Membership in Strings:
The membership operators also work with strings. For example:

```
txt = "Hello World"
print("H" in txt) # Returns True
print("hello" in txt) # Returns False
print("z" not in txt) # Returns True
```

REMEMBER: Membership operators are case-sensitive when used with strings.

1.11.7 Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description	Example
`&`	AND	Sets each bit to 1 if both bits are 1	`x & y`
`|`	OR	Sets each bit to 1 if of of two bits are 1	`x | y`
`^`	XOR	Sets each bit to 1 if only one of two bits are 1	`x ^ y`
`~`	NOT	Inverts all the bits	`~x`
`<<`	Zero fill left shift	Shift right by pushing zeros in rom the right and let the leftmost bits fall off	`x << 2`
`>>`	Zero fill right shift	Shift right by pushing copies of the leftmost bit from the left and let the rightmost bits fall off	`x >> 2`

Examples

The `&` operator compares each bit and set it to 1 if both are 1, otherwise it is set to 0.

Example 1:

```
x = 3 # The binary representation of 3 is 0011
y = 6  # The binary representation of 6 is 0110

# The & operator compares the bits and returns 0010, which is 2 in decimal
print(x & y) # Output is 2
```

The `|` operator compares each bit and set it to 1 if one or both is 1, otherwise it is set to 0.

Example 2:

```
# The | operator compares the bits and returns 0111, which is 7 in decimal
print(3 | 6) # Output is 7
```

The `^` operator compares each bit and set it to 1 if only one is 1, otherwise (if both are 1 or both are 0) it is set to 0.

Example 3:

```
# The ^ operator compares the bits and returns 0101, which is 5 in decimal
print(3 ^ 6) # Output is 5
```

The `~` operator inverts all the bits (0 becomes 1, and 1 becomes 0).

Example 4:

```
# The ~ operator inverts the bits and returns 1001, which is -7 in decimal
↪  (two's complement)
print(~6) # Output is -7
```

The `<<` operator is a bitwise left shift.

Example 5:

```
# Three is written as 0011, shifting this two to left gives 1100, which is 12 in decimal
print(3 << 2) # Output is 12
```

The >> operator is a bitwise right shift.

Example 6:

```
# Three is written as 0110, shifting this two to right gives 0001, which is 1 in decimal
print(6 >> 2) # Output is 1
```

1.11.8 Operator Precedence

Operator precedence describes the order in which operators are performed.

The precedence order is described in the table below, starting with the highest precedence at the top:

Operator	Description
`()`	Parentheses
`**`	Exponentiation
`+x -x ~x`	Unary plus, unary minus, bitwise NOT
`* / // %`	Multiplication, division, floor division, modulus
`+ -`	Addition and subtraction
`<< >>`	Bitwise left and right shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`== != > >= < <= is is not in not in`	Comparisons, identity, membership operators
`not`	Logical NOT
`and`	AND
`or`	OR

If two operators have the same precedence, the expression is evaluated from left to right.

Examples:

```
print((6 + 3) - (6 + 2)) # Output is 1
print(100 + 5 * 3) # Output is 115
print(5 + 4 - 7 + 3) # Output is 5
```

1.11.9 Challenge

Inside the editor, complete the following steps:

1. Create two variables `a = 15` and `a = 4`.

2. Print the result of `a` **modulus** `b`.
3. Print the result of `a` **floor division** `b`.
4. Print the result of `a` **to the power of** `b`.
5. Use an **assignment operator** to add **10** to `a`.
6. Print the result of comparing `a > b`.

Solution:

```
# Create variables
a = 15
b = 4

print(a % b) # Output is 3
print(a // b) # Output is 3
print(a ** b) # Output is 50624

a += 10 # a = 25

print(a > b) # Output is True
```

1.12 Python Collections (Arrays)

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is unordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered, unchangeable (buy you can remove and/or add items whenever), and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered (as of Python 3.7) and changeable. No duplicate members.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

1.13 Python Lists

- Lists are used to store multiple items in a single variable.
- Lists are one of 4 built-in data types in Python used to store collections of data.
- The other three are tuple, set, dictionary, all with different qualities and usage.
- Lists are created using square brackets.

Example:

```
mylist = ["apple", "banana", "cherry"]
print(mylist) # Output is ['apple', 'banana', 'cherry']
```

1.13.1 List Items

- List items are **ordered, changeable,** and allow **duplicate** values.
- List items are indexed, the first item has index `[0]`, the second item has index `[1]` etc.

Ordered:

- When we say that lists are ordered, it means that the items have a defined order, and that order will not change.
- If you add new items to a list, the new items will be placed at the end of the list.
- Note: There are some list methods that will change the order, but in general: the order of the items will not change.

Changeable:

- The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Allow Duplicates:

- Since lists are indexed, lists can have items with the same value. For example:

```
mylist = ["apple", "banana", "cherry", "apple", "cherry"]
print(mylist) # Output is ['apple', 'banana', 'cherry', 'apple', 'cherry']
```

List Length:
To determine how many items a list has, use the `len()` function.

Example:

```
mylist = ["apple", "banana", "cherry"]
print(len(mylist)) # Output is 3
```

Data Types:

- List items can be of any data type.
- A list can contain different data types.
- From Python,s perspective, lists are defined s objects with the data type 'list': `< class 'list'>`

Examples:

```
# Examples of different lists
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
list4 = ["abc", 34, True, 40, "male"]

print(type(list1)) # Output is < class 'list'>
```

The `list()` Constructor:
It is also possible to use the `list()` constructor when creating a new list. For example:

```
mylist = list(("apple", "banana", "cherry")) # Note the double round-brackets
print(mylist)
```

1.13.2 Access List Items

Access Items:

- List items are indexed and you can access them bt referring to their index number
- **REMEMBER:** The first item has index 0.

Example:

```
mylist = ["apple", "banana", "cherry"]
print(mylist[1]) # Output is banana
```

Negative Indexing:

- Negative indexing means start from the end.
- -1 refers to the last item, -2 refers to the second last item etc.

Example:

```
mylist = ["apple", "banana", "cherry"]
print(mylist[-1]) # Output is cherry
```

Range of Indexes:

- You can specify a range of indexes by specifying where to start and where to end the range.
- When specifying a range, the return value will be a new list with the specified items.
- For the range, 2:5, the search will start at index 2 (included) and end at index 5 (not included).

Example:

```
# Return the third, fourth, and fifth item
mylist =["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(mylist[2:5]) # Output is ['cherry', 'orange', 'kiwi']
```

By leaving out the end value, the range will go on to the end of the list.

Example:

```
mylist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(mylist[2:]) # Returns the items from "cherry" to the end
```

Range of Negative Indexes:
Specify negative indexes if you want to start the search from the end of the list.

Example:

```
mylist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(mylist[-4:-1]) # Output is ["orange", "kiwi", "melon"]
```

Check if Item Exists:
To determine if a specified item is present in a list use the `in` keyword.

Example:

```
mylist = ["apple", "banana", "cherry"]
if "apple" in mylist:
    print("Yes, 'apple' is in the fruits list")
```

1.13.3 Change List Items

Change Item Value:
To change the value of a specific item, refer to the index number.

Example:

```
mylist = ["apple", "banana", "cherry"]
mylist[1] = "blackcurrant"
print(mylist) # Output is ['apple', 'blackcurrant', 'cherry']
```

Change a Range of Item Values:
To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values.

Example:

```
# Change the values "banana" and "cherry" with the values "blackcurrant" and
↪  "watermelon"
mylist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
mylist[1:3] = ["blackcurrant", "watermelon"]
print(mylist) # Output is ['apple', 'blackcurrant', "watermelon', 'orange',
↪  'kiwi', 'mango']
```

If you insert *more* items than you replace:

- The new items will be inserted where you specified, and the remaining items will move accordingly.
- The length of the list will change.

Example:

```
mylist = ["apple", "banana", "cherry"]
mylist[1:2] = ["blackcurrant", "watermelon"]
print(mylist) # Output is ['apple', 'blackcurrant', 'watermelon', 'cherry']
```

If you insert *less* items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly.

Example:

```
# Change the second and third value by replacing it with one value
mylist = ["apple", "banana", "cherry"]
mylist[1:3] = "watermelon"
print(mylist) # Output is ['apple', 'watermelon']
```

1.13.4 Add List Items

Append Items:
To add an item to the **end** of the list, use the `append()` method. For example:

```
mylist = ["apple", "banana", "cherry"]
mylist.append("orange")
print(mylist) # Output is ['apple', 'banana', 'cherry', 'orange']
```

Insert Items:

- To insert a new list item, without replacing any of the existing values, we can use the `insert()` method.
- The `insert()` method inserts an item at the specified index.

Example:

```
# Insert "watermelon" as the third item
mylist = ["apple", "banana", "cherry"]
mylist.insert(2, "watermelon")
print(mylist) # Output is ['apple', 'banana', 'watermelon', 'cherry']
```

Extend List:

- To append elements from *another list* to the current list, use the `extend()` method.
- The elements will be added to the **end** of the list.

Example:

```
mylist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
mylist.extend(tropical)
print(mylist) # Output is ['apple', 'banana', 'cherry', 'mango', 'pineapple',
↪  'papaya']
```

Add Any Iterable:
The `extend()` method does not have to append *lists*, you can add any iterable object (tuples, sets, dictionaries etc.). For example:

```
mylist = ["apple", "banana", "cherry"]
mytuple = ("kiwi", "orange")
mylist.extend(mytuple)
print(mylist) # Output is ['apple', 'banana', 'cherry', 'kiwi', 'orange']
```

1.13.5 Remove List Items

Remove Specified Item:

- The `remove()` method removes the specified item.
- If there are more than one item with the specified value, the `remove()` method removes the first occurence.

Example:

```
mylist = ["apple", "banana", "cherry", "banana", "kiwi"]
mylist.remove("banana")
print(mylist) # Output is ['apple', 'cherry', 'banana', 'kiwi']
```

Remove Specified Index:

- The `pop()` method removes the specified index.
- If you do not specify the index, the `pop()` method removes the last item.

Example:

```
# Specified Index
mylist = ["apple", "banana", "cherry", "dragon fruit"]
mylist.pop(1)
print(mylist) # Output is ['apple', 'cherry', 'dragon fruit']

# Unspecified Index
mylist.pop()
print(mylist)# Output is ['apple', 'cherry']
```

- The `del()` keyword also removes the specified index.
- The `del()` keyword can also delete the list completely.

Example:

```python
# Delete the specified index
mylist = ["apple", "banana", "cherry"]
del mylist[0]
print(mylist) # Output is ['banana', 'cherry']

# Delete the list completely
del mylist
print(mylist) # This will cause an error because you have successfully deleted
↪  "mylist",
```

Clear the List:

- The `clear()` method empties the list.
- The list still remains, but it has no content.

Example:

```python
mylist = ["apple", "banana", "cherry"]
mylist.clear()
print(mylist) # Output is []
```

1.13.6 Loop Lists

Loop Through a List: You can loop through the list items by using a `for` loop.

Example:

```python
mylist = ["apple", "banana", "cherry"]
for x in mylist:
    print(x) # Output is apple banana cherry (on separate lines)
```

Loop Through the Index Numbers:

- You can also loop through the list items by referring to their index number.
- Use the `range()` and `len()` functions to create a suitable iterable.

Example:

```python
# Print all items by referring to their index number
mylist = ["apple", "banana", "cherry"]
for i in range(len(mylist)):
    print(mylist[i]) # Output is apple banana cherry (on separate lines)
```

Using a While Loop:

- You can loop through the list items by using a `while` loop.
- Use the `len()` function to determine the length of the list, then start at 0 and loop your way through the list items by referring to their indexes.
- Remember to increase the index by 1 after each interation.

```
mylist = ["apple", "banana", "cherry"]
i = 0

while i < len(mylist):
    print(mylist[i])
    i = i + 1 # Output is apple banana cherry (on separate lines)
```

Looping Using List Comprehension:
List comprehension offers the shortest syntax for looping through lists.

Example:

```
# A shorthand for loop that will print all items in a list
mylist = ["apple", "banana", "cherry"]
[print(x) for x in mylist] # Output is apple banana cherry (on separate lines)
```

1.13.7 List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Example:

- Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.
- Without list comprehension you will have to write a `for` statement with a conditional test inside:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
    if "a" in x:
        newlist.append(x)

print(newlist) # Output is ['apple', 'banana', 'mango']
```

With list comprehension you can do all that with only one line of code:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]

print(newlist) # Output is ['apple', 'banana', 'mango']
```

The Syntax:

```
newlist = [expression for item in iterable if condition == True]
```

The return value is a new list, leaving the old list unchanged.

- The **condition** is like a filter that only accepts the items that evaluate to `True`. It is optional and can be omitted.

- The **iterable** can be any iterable object, like a list, tuple, set etc.
- The **expression** is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list. You can set the outcome to whatever you like.

Later:

- Using enumerate() with lists?