Crash Course in Python

Introduction

This guide is a *swift* introduction to Python and programming for non-programmers. For the sake of brevity, many details have been omitted. However, resources will be provided for the ambitious.

You should follow along as you read this guide.

1. What is Programming?

Programming is the process of specifying a series of instructions for a computer to carry out. Such instructions depend on the programming language, just like how the words used to form ideas depend on a language. Programming languages also differ in levels in abstraction, hence low or high-level languages. We use programming to solve problems that can be explicitly written out with such instructions.

Abstraction is an important, reoccurring topic in computer science. Besides abstracting complexity in programming languages, abstraction simplifies instructions and data in code. Respective analogies would be how the words "make a sandwich" abstract away the steps of making a sandwich and how the words "Eiffel Tower" abstracts away the details of what that object is.

Python is a simple, yet powerful high-level programming language that is widely used in research.

x. To Consider

1. Why Python?
    1. 1.8 Why Python? - Introduction to Python Programming | OpenStax
2. What is being abstracted in a high-level language?
    1. Why do some programmers categorize C, Python, C++ differently? - regarding level - Software Engineering Stack Exchange
3. How does a computer understand code written in a (relatively) low-level language?
    1. 0.2 — Introduction to programming languages – Learn C++
4. How does a computer understand Python code?

1. How does a Python interpreter work? - tutorialspoint

2. Python Installation

Visit https://www.python.org/downloads/ and download the latest version for your platform. This download should additionally include IDLE (Integrated Development and Learning Environment), which is not much more than a text editor that you can run code from.

Visit https://python.swaroopch.com/installation.html for more details regarding this process.

a. Using Brew on MacOS

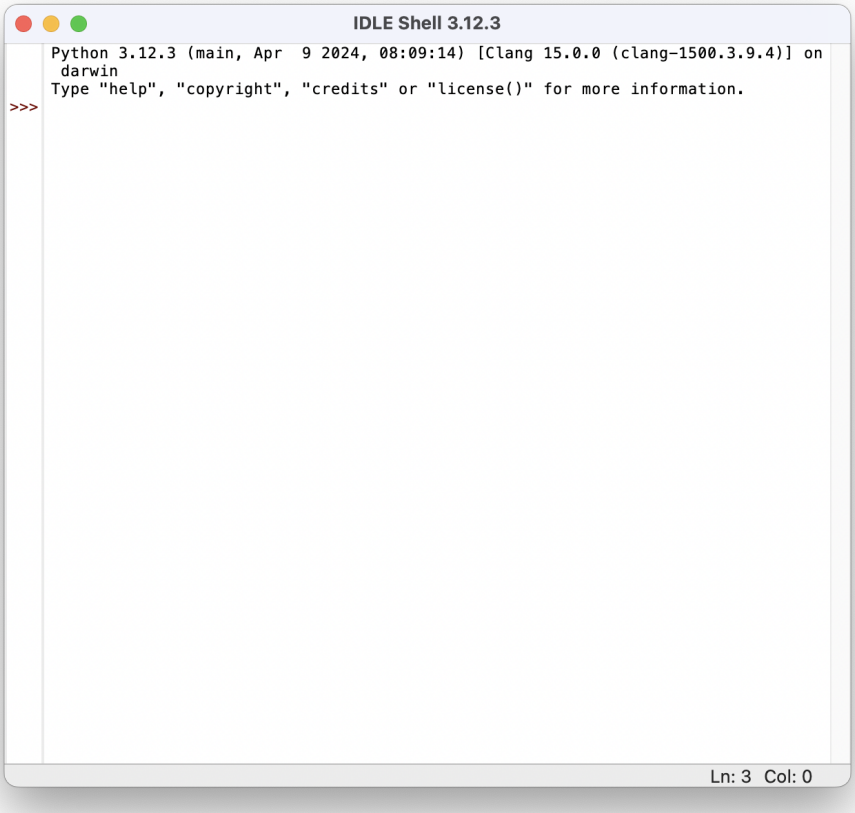
If you opted to use Homebrew to install python, an additional step will be required to open IDLE.

First, you will also need to install `Tkinter` (don't worry about what it is) with Homebrew with `brew install python-tk`. Then you must open `Terminal`, type `idle3`, and press `[Enter]` to run IDLE.

To open the terminal, open Spotlight search `[Command + Space]`, type `Terminal`, and press `[Enter]`.

3. First Steps

Opening IDLE will open the IDLE Shell.

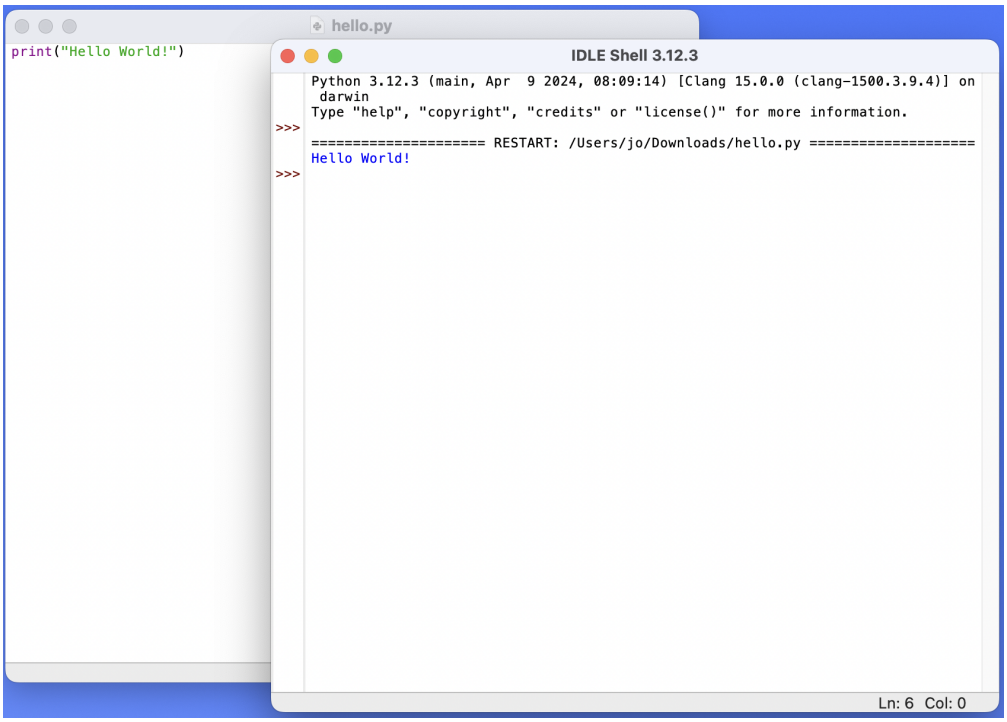


Click `File`, then `New File`. Then, save this file as `hello.py` (`hello` can be whatever you'd like, `.py` is the important part) in a folder for which you know the location of the folder (e.g. `Downloads`, `Documents`, even `Pictures`, it doesn't matter).

Type the following into the editor, then click `Run` and `Run Module`. Alternatively, you could hit `[F5]`.

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

The IDLE Shell will be focused and beneath `==...== RESTART: /path-to-folder/hello.py ==...==`, you will see that "Hello World!" has been "printed".



Congratulations, you have just run your first program in Python!

Now, and in the future, the code editor is where you will edit your code and the shell is where you will see your code's output.

x. To Consider

1. Why "Hello World!"?
    1. "Hello, World!" program - Wikipedia
2. What are other options for editing and running Python code?
    1. First Steps · A Byte of Python
3. What is the shell?
    1. 2.1 The Python shell - Introduction to Python Programming | OpenStax

4. Statements

In our `hello.py` program, we had a single **statement**, being `print("Hello World!)`, where `print` was a function that took `"Hello World!"` as an input. `print` literally prints any input given to it.

In this chapter, we will explore a variety of statements.

a. Comments

Comments are any text to the right of a `#` and are completely ignored by the interpreter. They are often used as notes for the reader of the program.

b. Variables

Literal constants are values that are explicitly defined in code. `"Hello World!"` is an example of a literal constant. It is called a literal because you use its value literally. It is also called a constant because its value cannot be changed.

In order to store or change information, we need to use variables, whose value can *vary*. Unlike literal constants, which are available only where we specify, variables must be accessed using a name.

In Python, anything used in a program is referred to as an **object**, just as we might say "that something". Furthermore, anything that we identify with a *name* is called an **identifier**. Besides variables, identifiers are used to identify a variety of other objects which may not necessarily be variable.

Here's a simple example of using a variable in a program.

Code:

```
# Note that my_variable is completely arbitrary. Any other valid name could
be used.
my_variable = "Hello World!"
print(my_variable)
```

Output:

```
Hello World!
```

x. To Consider

1. What rules are there for naming variables?
    1. Python Keywords and Identifiers (With Examples)
2. Is `my_variable` also an object? What if it were a number instead?

c. Functions

Functions are objects that take any number of inputs (including zero) and return (at most) one output. Functions can be used to condense multiple statements into a single named object that can be run anywhere and any number of times. Running a function is known as *calling* the function.

`print` is an example of a function. Behind the scenes, `print` is ~80 lines of C code. The details of how `print` works is something that we do not have to, or want to, think about.

Besides functions that are provided to us by Python, we can write our own functions, as well. An example is shown below.

```
def quadratic(a, b, c, x):
        term0 = c
        term1 = b * x
        term2 = b * x * x
        return term0 + term1 + term2
```

The first line is called the **function signature**. This begins with the `def` keyword, is followed by the function identifier and a pair of parenthesis which may enclose the names of any number of parameters (inputs). Finally, the colon signals the end of this line. Next follows any number of statements and you might output a value with the `return` keyword. Notice that all statements belonging to the function are indented.

Parameters to a function and variables defined within a function are **local** to the function. This means that they only exist within the **scope** of the function. All variables have the scope of the function that they are declared in. An example of this is shown below:

```
x = 1

def f(x):
    print('x is', x)
    x = 2
    print('Changed local x to', x)

f(x)
print('x is still', x)
```

```
x is 1
Changed local x to 2
x is still 1
```

x. To Consider

1. What if we want to declare a variable in a function that is not necessarily local to just the function? What if we want to give a function default argument values? What if we want to make the number of parameters to a function variable?
    1. Functions · A Byte of Python

d. Input and Output

The `print` function is the most basic form of output. If multiple values are given to `print` as an input, they will be printed in the same order, separated by a single space.

The `input` function is the most basic form of input. `input` reads a single line of input from the user and outputs the input as a string. A prompt can be given to the function, which it will display before accepting input. Often, the output of `input` is stored in a variable for later user.

Here is an example of using `input` and `print` together. After running the code, "Hello" is typed, before `[Enter]` is pressed and "World!" is typed, before `[Enter]` is pressed. Try running the code yourself to see how this input works.

Code:

```
a = input("Type something: ")
b = input("Type something: ")
print(a, b)
```

Output:

```
Type something: Hello
Type something: World!
Hello World!
```

x. To Consider

1. What if I want to format my output differently (e.g. display something besides a space between values or don't start a new line after a print)?
    1. 1.2 Input/output - Introduction to Python Programming | OpenStax
2. Why does the following happen?

(From now on, the output will be the block succeeding the code (if any))
Case 1: 1 + 2 from literals

```
a = 1
b = 2
print(a + b)
```

```
3
```

Case 2: 1 + 2 from an input

```
a = input("Type something: ")
b = input("Type something: ")
print(a + b)
```

```
Type something: 1
Type something: 2
12
```

x. To Consider

1. I want more details!
    1. Basics · A Byte of Python

5. Expressions

In this chapter, we will explore **expressions**, a combination of operators and values that produce a resulting value when evaluated. An example of an expression is `1 + 2`. Here, `+` is an operator that acts on the values `1` and `2`. When evaluated, this expression will produce the resulting value `3`.

a. The Python shell

Earlier, we saw how to run Python code by typing it into an editor and running it all at once. This is convenient for working with large amounts of code where you want each evaluation of the code to be independent of another. However, typing an expression into the editor within a print

statement, saving the file, then running the file can be be cumbersome for exploring how expressions work where we might want to evaluate many snippets and see their result.

The **Python shell** allows one to experiment with code interactively. While an entire program is ran at once with IDLE, the shell allows as to run a single line of code at a time. We can use the Python shell simply by typing into the `IDLE Shell` window after the `>>>`. Try typing in some expression. Even without explicitly printing it, the result will be printed.

```
>>> 1 + 2 * 3
7
```

Use the shell to experiment with the expressions we'll encounter in the upcoming sections!

b. Math Basics

Numeric Data Types

Multiple data types exist in Python. So far, we have worked with integers and strings and have seen how they behave differently. Multiple data types also exist for numbers.

Recall that the integer set is made up of natural numbers ($1, 2, \ldots$), zero, and the additive inverses of the natural numbers ($-1, -2, \ldots$). We refer to this set when referring to integers in Python, as well.

Floats (floating-point numbers) are the data type used to represent numbers with greater precision. (Almost) any finite decimal number can be represented with floats (floating-point math has its quirks, see below).

You can check the data type of any object, with the `type` function. Try evaluating `type(1)`, `type(1.0)`, and `type("1")`!

Order of Operations

The order of operations in Python is similar to the order of operations you are familiar with from high school maths. The full list can be found in the reference manual linked below.

You can change (or assert) the order of evaluation to your liking using parenthesis. Using parenthesis can improve readability but can certainly be overdone. For example, `1 + 2 * 3` is more easily read as `1 + (2 * 3)` but `((1) + ((2) * (3)))` is redundant and difficult to read.

Arithmetic Operations

The arithmetic operators available in Python are given in the table below as reference.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation
%	Modulus (remainder)
//	Floor division (quotient)

Comparison Operations

The following operators can be used to compare two numbers. You should be familiar with all notations except for equal, where two equal signs are used instead of just one. This is to ensure the distinction between itself and the assignment operator, `=`.

Operator	Condition
==	Equal
!=	Not equal
>	Greater than
<	Lesser than
>=	Greater than or equal to
<=	Less than or equal to

Unlike arithmetic operators, these comparison operators will evaluate to either a `True` or a `False` value. These values are of a new data type, known as **booleans**, something that can either be "true" or "false". We will explore booleans in a later section.

The `math` module

Python has an extensive **standard library** of **modules**. A module is code that can be used in another program. This allows you to use code without having to write it out again. `print`, `input`, `int`, etc. are all built-in functions defined by the standard library. Built-in functions are those that are always available.

A commonly used module in the standard library is the `math` module. This module defines a variety of useful functions, including `sqrt`, `factorial`, and `log`. These functions are not built-in and must be explicitly included using the `import` keyword.

```
import math

x2 = float(input("Enter a number you would like to know the square root of:
"))
x = math.sqrt(x2)
print(x)
```

```
Enter a number you would like to know the square root of: 2
1.4142135623730951
```

x. To Consider

1. What are all operators available in Python? What is the order of precedence of operations in Python?
    1. Python Operators
    2. 6.16. Operator Precedence — Python 3.12.4 documentation
2. `0.1 + 0.2 != 0.3`
    1. Is floating-point math broken? - Stack Overflow
3. What would happen if you multiplied a string and a number?
    1. Try it yourself! See what `"Hello!" * 3` evaluates to.
4. What would happen if you divided integers? Does implicit type conversion occur? For example, would `7 / 4` result in `1.75` or `1`?
    1. Try it yourself! Note that another operator, floor division (`//`) exists to compute the quotient without the remainder.
5. What are all of the built-in functions?
    1. Built-in Functions — Python 3.12.4 documentation

c. Boolean Operations

Logical operators take boolean operand(s) and output one boolean result. Logical operators have an associated *truth table*, where combinations of operands are mapped to the corresponding result. Three logical operators are available in Python: `and`, `or`, and `not`.

`and`

The `and` operator tests that (returns `True` if) both operands are `True`.

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

`or`

The `and` operator tests that at least one operand is `True`.

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

`not`

The `or` operator takes in only one operand and outputs the reverse.

A	not A
False	True
True	False

d. Type Conversion

What would happen if we evaluated `1 + "2"`? Should the integer be converted to a string, resulting in `"12"`? Or should the string be converted to an integer, resulting in `3`? Fortunately, we don't have to guess, as evaluating the expression will result in the following error: `TypeError: unsupported operand type(s) for +: 'int' and 'str'`.

How about if we evaluated `1 + 2.0`? Again, we are mixing two different data types, an integer and a float. In this case, the integer is converted to a float and the expression evaluates to `3.0`.

This is an example of **implicit type conversion**, where the Python interpreter automatically and silently converts one data type to another.

How could we make the first example, `1 + "2"` work? We can use **explicit type conversion** to explicitly convert one data type to another. `int`, `float`, `str`, and `bool` are all functions which convert the data type of their input to their respective data type. With this, we can specify what result we want. If we want the expression to evaluate to `"12"`, we can use `str(1) + "2"`. If we want the expression to evaluate to `3`, we can use `1 + int("2")`. However, `bool` returns the "truthiness" of values. In Python, all values are "truthy" except for some that are defined as "falsy". One example of a "falsy" value is the integer, `0`.

A verb exists to describe the action of converting one type to another, "cast". For example, "Explicitly convert the data type of the variable, `x` to a string" can be said as "Explicitly cast `x` to a string".

x. To Consider

1. What are all of the "falsy" values?
    1. Truth Value Testing — Python 3.12.4 documentation

6. Control Flow

So far, all of the code that we have written in our programs have been executed from top to bottom. What if we want to selectively skip or repeat some code? With control flow statements, we can control the flow of execution with three control flow statements - `if`, `for`, and `while`.

a. The `if` statement

The `if` statement checks a condition and runs the code within its block *if* the condition is true. It can also have an optional `else` or `elif` clause. An example is given below, in the form of a guessing game.

```
number = 4
guess = int(input("Guess a number 0-5: "))

if guess == number:
        print("Correct!")
elif guess < number:
        print("Incorrect...try higher!")
```

```
else:
        print("Incorrect...try lower!")
```

In the first two lines, we set the number to be guessed, `number`, to `4` and take an input from the user that is converted to an `int`, the same type as `number`, and store it in `guess`.
We test the first condition, `guess == number`. The colon (`:`) indicates that a block of statements will follow. If the condition is true, the program will print "Correct!".
The second condition, `guess < number`, is preceded by the keyword `elif`, which is "else-if" abbreviated. This condition is only checked if the first one fails and, if met, the program will print "Incorrect...try higher!".
If neither of the preceding conditions are met, the code in the `else` block will be executed, printing "Incorrect...try lower!"

x. To Consider

1. Could the `else` be replaced with `elif guess > number`? Why?
2. Would the print statement in the following code run?

```
x = 1
if x:
        print("Hello!")
```

b. The `while` loop

The `while` statement repeatedly executes a block of code as long as a condition is true. It can also have an optional `else` clause, which will run when the condition is `False`. One cycle of a loop is called an *iteration*. We can expand on our guessing game with this statement to allow a user to make multiple guesses without having to restart the program.

```
number = 4
guess = -1

while guess != number:
        guess = int(input("Guess a number 0-5: "))
        if guess < number:
                print("Incorrect...try higher!")
        elif guess > number:
                print("Incorrect...try lower!")

print("Correct!")
```

The first two lines are the same, except the initial guess is set to `-1`. This is done so that the condition is `False` on the first evaluation. Really, the initial guess could be any value besides the number.
We check whether the guess is not equal to the number because we want to accept guess until a correct one is made. The first iteration will always run because the number and guess are set to be unequal to each other. We accept a guess from the user and check whether it is less than or greater than the number. Here, we have to explicitly test `guess > number` because, unlike before, we don't already know what this condition will be
Once a correct guess is made, we will exit the while loop and the program will continue, printing "Correct!".

c. The `for` loop

The `for...in` statement *iterates* over a sequence of objects (i.e. go through them, one by one in order). It can also have an optional `else` clause. We can change our guessing game to only accept a certain number of guesses.

```
number = 4
guess = -1
n_guesses = 5
guessed = False

for i in range(1, 1 + n_guesses):
        guess = int(input("Guess a number 0-5: "))
        if guess == number:
                guessed = True
                break
        elif guess < number:
                print("Incorrect...try higher!")
        else:
                print("Incorrect...try lower!")
else:

if guessed:
        print("Correct!")
else:
        print("You ran out of guesses, try again!")
```

Here, we have a couple of more variables. `n_guesses` defines the number of guesses the user will have. `guessed` is a boolean variable that lets us track whether the number has been guessed or not.
`range(a, b)` is a function that returns a sequence of numbers from `a` to `b`, exclusive (i.e.

integers in $[a, b)$). In each iteration of the for loop, `i` will be assigned to each number in order. While it isn't relevant in this example, `i` will be `1` ... `5`, the 1st...5th iteration, respectively. Each iteration, the code in the for block will be run. The only thing we have changed from the first example is that, instead of printing "Correct" when the correct guess is made, we set `guessed = True` and `break` (we will see what it does in the next section).
Finally, we check whether the number was guessed or not and print a relevant statement.

d. The `break` statement

The `break` statement is used to *break* out of a loop (i.e. prematurely stop the execution of a loop). Note that, if you break out of a loop, the corresponding `else` will not be executed. An example of its usage is in 7.c.

e. The `continue` statement

The `continue` statement is used to *continue* on to the next loop (i.e. prematurely begin the next iteration, skipping all remaining statements).

7. Data Structures

Data structures are structures which hold data (`int`, `bool`, etc.) together. For example, how can we hold a collection of grades, represented as `float`'s without having to define a new variable for every grade? With a data structure.

There are four built-in data structures in Python - *list, tuple dictionary, and set*.

These data structures include a *subscript* operator that allow you to access elements within them. This operator is done by using square brackets after the variable name.
Recall that a subscript to a variable in math refers to some element at that position (e.g. number of a series, vector of a set, column of a matrix, etc.). Then, x_0, in math and `x[0]`, in Python, both represent the first element of x, whatever that might be.

a. List

A `list` is a data structure that holds an ordered collection of items. This would allow us to hold a collection of grades in a single structure. A list is defined by listing items with a comma in between, and wrapping the list in square brackets. An example is given below.

```
grades = [91, 87, 97]
```

Items can be added and removed from a list, meaning that it is a *mutable* (can be mutated, changed) data type. Some examples of things you can do to lists are given below:

```
print(grades[0]) # print the first element in the list
print(len(grades)) # print the number of element in the list
# replace first grade with 94
grades.remove(0) # remove the first element in the list
grades.insert(0, 94) # insert the integer 92 into the list at index 0 (the
front)
# revert the replacement
grades[0] = 91 # set the first element of the list to be 91
```

x. To Consider

1. I want to know more about lists.
    1. Ch. 9 Introduction - Introduction to Python Programming | OpenStax

b. Tuple

A `tuple` is essentially an immutable `list`. They are defined by wrapping items, separated by commas, with parenthesis.

c. Dictionary

A `dict` is a data structure that maps *keys* to *values*. For example, an actual dictionary is a `dict` that maps words to their definitions. Duplicate keys cannot exist and keys can only be immutable objects (an `int`,a `str`, and even a `tuple` would be valid but **not** a `list`).
A dictionary is defined with key-value pairs denoted by `key : value`, separated by commas, and all pairs should be wrapped in curly brackets.
Note that dictionaries in Python are unordered.

```
phone_book = {
        "John" : "404-789-123",
        "Jane" : "678-345-901"
}
```

```
print("John's number is:", phone_book["John"])
print("And Jane's number is:", phone_book["Jane"])
```

d. Set

A `set` is a data structure that that holds an *unordered* collection of *unique* elements. They are used when the existence of an object in a collection is more important than the order or number of times it occurs.
Sets are defined by wrapping items, separated by commas, in curly brackets. Be careful not to mix up set and dictionary definitions!

If you remember basic set theory from school, sets and the things you can do with them should be self-explanatory. You should also understand why elements of a set are unordered and unique.

x. To Consider

1. What can you do with sets?
    1. Everything you can do with sets in math (union, intersection, etc.)
    2. Sets in Python - GeeksforGeeks

8. Objects and Classes

As mentioned earlier, objects are any "something" in programming. Specifically, the term refers refer an **instance** of a class. For example, for `x=10`, `x` is an object of the `int` class, or type (you can imagine the type to be the same as the class name for now).

Besides a name, classes also have **methods** (i.e. functions that belong to the class). For example, in `my_list.remove(0)`, `remove` is a method of the `list` class that `my_list` refers to.

A class can also have **fields** (i.e. variables that belong to the class). A field can either be an instance variable or a class variable, which belong to each instance of the class or to the class itself, respectively. *We will not discuss class variables.*

Collectively, the methods and fields of a class can be referred to as the **attributes** of that class. All attributes of a variable can be checked by passing variable into the `dir` function.

All of the types that we have seen so far, from `int` to `set`, are all defined by a class.

A class is created using the `class` keyword. The fields and methods of the class are defined in the indented block that follows.

We will explore the creation of a class with an example, where we create a representation of a person who has attributes and can do things.

a. `self`

Within a class, the first parameter of all methods must be a variable representing *itself*. By convention, this parameter is named `self`.
When calling a method, you never provide an argument for `self`, as Python automatically provides it.

x. To Consider

1. How does Python provide the value for `self`?

b. Basics

Let's start with the simplest class possible:

```
class Person:
        pass # do nothing

john = Person()
print(john)
```

```
<__main__.Person instance at 0x10171f518>
```

We define the class and create an instance of the class using the name of the class followed by a pair of parenthesis.
Printing our class prints information about the object (where is the class defined, where in memory is the instance). This behavior is the default. We will see that we can change what is printed when we print an instance of a class.

c. The `__init__` method

```
class Person:
        def __init__(self, name):
                self.name = name

john = Person("John")
print(john.name)
```

```
John
```

The `__init__` method is run when an instance of the class is instantiated (created). The method handles *initialization* of the object. Here, we set a field of the class to the parameter `name`.
We access the field with a period, and our program prints the name that we gave to our person.

d. Inheritance

```
class Person:
        def __init__(self, name, occupation):
                self.name = name
                self.occupation = occupation

        def self_introduction(self):
                print("Hello! My name is " + self.name + ". Currently I am a
" + self.occupation + ".")

john = Person("John", "secretary")
john.self_introduction()
```

```
Hello! My name is John. Currently I am a secretary.
```

We gave our person an additional instance variable, `self.occupation` and a new method where they can introduce themselves.

Now, what if we want to give secretaries a `make_appointment` method, where they set up an appointment for their boss? Well, if we added that method to `Person`, we would be giving all instances of `Person` that method. However, it wouldn't make sense for people who aren't secretaries to be making appointments for their boss.

We can create a new secretary class which has this method, so that we don't bloat all instances of `Person` with the method.

```
class Person:
        def __init__(self, name, occupation):
                self.name = name
                self.occupation = occupation

        def self_introduction(self):
                print("Hello! My name is " + self.name + ". Currently I am a
" + self.occupation + ".")

class Secretary:
        def __init__(self, name):
                self.name = name
                self.occupation = "secretary"

        def self_introduction(self):
                print("Hello! My name is " + self.name + ". Currently I am a
" + self.occupation + ".")

        def make_appointment(self):
                pass # do whatever it would do

john = Secretary("John")
john.self_introduction()
```

This code outputs the same thing as before but now `john`, being an instance of `Secretary` has the `make_appointment` method.

Notice that a lot of the same code is being reused, like the initialization function and `self_introduction`. We can use **inheritance** to avoid repeating ourselves like this.

```
class Person:
        def __init__(self, name, occupation):
                self.name = name
                self.occupation = occupation

        def self_introduction(self):
                print("Hello! My name is " + self.name + ". Currently I am a
" + self.occupation + ".")

class Secretary(Person):
        def __init__(self, name):
                super().__init__(name, "secretary")

        def make_appointment(self):
                pass # do whatever it would do
```

```
john = Secretary("John")
john.self_introduction()
```

By following the name of `Secretary` with `Person`, wrapped in parenthesis, we make the subclass `Secretary` inherit from the superclass `Person`. This means that all fields and methods of `Person` are available in `Secretary`. This is why we can call `self_introduction` through `john` even though `john` is an instance of `Secretary`.

When we define the `__init__` method in `Secretary` again, we override the `__init__` method of `Person`. Otherwise, `Secretary` would be initialized using the initialization method defined in `Person`. To still use `__init__` of `Person`, we use `super().__init__`, where `super()` is simply the superclass. `Person.__init__` works, as well, but you will have to remember to update the statement if you ever change the identifier of `Person`.

Using the superclass's `__init__`, we set the name to a parameter and the occupation to `"secretary"`, as we already know that.

Appendix

To Consider - Answers

5

b

2. Yes, as stated, everything used in a program can be considered objects.

d

2. See Crash Course in Python > 6. Expressions. In the first code block, two **integers** are added together, resulting in behavior that we would expect. In the second code block, we read in two numbers as inputs. However, the output of `input` is always a **string**, regardless of if the input was a digit. So `a="1"` and `b="2"` rather than `a=1` and `b=2` The behavior of adding two strings together is to concatenate (put together) the two strings. Thus, `"1"+"2"` behaves much like `"Hello " + "World!"`, evaluating to `"12"` and `"Hello World!"`, respectively.

7

a

1. If the first two conditions have not been met, we know that `guess == number` is `False` and `guess < number` is also `False`. Equivalently, we know that `guess != number` and `guess >= number`, or `guess > number` together. Therefore, we already know that `guess > number` is `True` and we don't have to test this condition.
2. Yes. `if` and `elif` implicitly cast non-boolean values to booleans. The same is done when using logical operators on non-boolean values.

8

a

1. From Object Oriented Programming · A Byte of Python:
    1. Say you have a class called `MyClass` and an instance of this class called `myobject`. When you call a method of this object as `myobject.method(arg1, arg2)`, this is automatically converted by Python into `MyClass.method(myobject, arg1, arg2)` - this is all the special `self` is about.

References

1. https://python.swaroopch.com
2. https://www.learncpp.com
3. https://www.codecademy.com/article/what-is-programming
4. https://openstax.org/details/books/introduction-python-programming
5. https://wiki.python.org/moin/BeginnersGuide/NonProgrammers