Lecture Book
Python for Eng. and Eco.

D. T. McGuiness, PhD

Version: 2024

Topics on Computer Science

Current version is 2024.

This document includes the contents of Python for Eng. and Eco. taught at MCI. This document is the part of Interdisciplinary Elective.

All relevant code of the document is done using *SageMath* v10.3 and Python v3.12.5.

This document was compiled with LuaTEX and all editing were done using
GNU Emacs using AUCTEX and org-mode package.

This document is based on resources on Python programming, which includes *Defining Your Own Python Function* by J. Sturtz.

The current maintainer of this work along with the primary lecturer
is D. T. McGuiness, PhD (dtm@mci4me.at).

Contents

Part I.

Fundamentals

Chapter 1

Welcome

Table of Contents

Python is a multiplatform programming language, written in **C**, under a free license. It is an interpreted language, i.e., it requires an interpreter to execute commands, and has **no compilation phase**. Its first public version dates from 1991.

Figure 1.1.: The name Python comes from the cult classic "Monthy Python's Flying Circus"

Compiled v. Interpreted

In a compiled language, the target machine directly translates the program. In an interpreted language, the source code is not directly translated by the target machine. Instead, a different program, aka the interpreter, reads and executes the code

The main programmer, Guido van Rossum, had started working on this programming language in the late 1980s. The name given to the Python language comes from the interest of its main creator in a British television series broadcast on the BBC called Monty Python's Flying Circus.

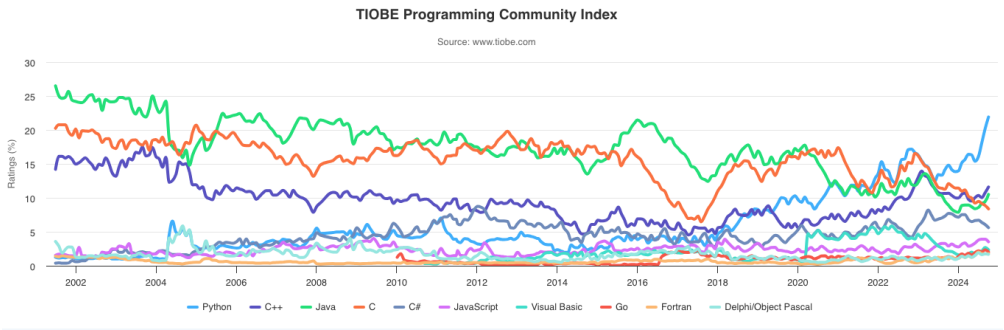
The popularity of Python has grown strongly in recent years, as confirmed by the survey results provided since 2011 by Stack Overflow. Stack Overflow offers its users the opportunity to complete a survey in which they are asked many questions to describe

their experience as a developer. The results of the 2019 survey show a new breakthrough in the use of Python by developers.

1.1 Features

1. **Simple and Easy to Learn:** Python has a simple syntax, which makes it easy to learn and read. It's a great language for beginners who are new to programming.
2. **Interpreted:** Python is an interpreted language, which means that the Python code is executed line by line. This makes it easy to test and debug code.
3. **High-Level:** Python is a high-level language, which means that it abstracts away low-level details like memory management and hardware interaction. This makes it easier to write and understand code.
4. **Dynamic Typing:** Python is dynamically typed, which means that you don't need to declare the data type of a variable explicitly. Python will automatically infer the data type based on the value assigned to the variable.
5. **Strong Typing:** Python is strongly typed, which means that the data type of a variable is enforced at runtime. This helps prevent errors and makes the code more robust.
6. **Extensive Standard Library:** Python comes with a large standard library that provides tools and modules for various tasks, such as file I/O, networking, and more. This makes it easy to build complex applications without having to write everything from scratch.
7. **Cross-Platform:** Python is a cross-platform language, which means that Python code can run on different operating systems without modification. This makes it easy to develop and deploy Python applications on different platforms.
8. **Community and Ecosystem:** Python has a large and active community, which contributes to its ecosystem. There are many third-party libraries and frameworks available for various purposes, making Python a versatile language for many applications.
9. **Versatile:** Python is a versatile language that can be used for various purposes, including web development, data science, artificial intelligence, game development, and more.

   Since 2003, Python has consistently ranked in the top ten most popular programming languages in the TIOBE Programming Community Index where as of December 2022 it was the most popular language (ahead of C, C++, and Java). It was selected as Programming Language of the Year (for "the highest rise in ratings in a year") in 2007, 2010, 2018, and 2020 (the only language to have done so four times as of 2020).



Large organizations that use Python include Wikipedia, Google, Yahoo!, CERN, NASA, Facebook, Amazon, Instagram, Spotify, and some smaller entities like Industrial Light & Magic and ITA. The social news networking site Reddit was written mostly in Python. Organizations that partially use Python include Discord and Baidu.

1.2 Hello, World !

A ”Hello, World!” program is generally a simple computer program that emits (or displays) to the screen (often the console) a message similar to ”Hello, World!”. A small piece of code in most general-purpose programming languages, this program is used to illustrate a language’s basic syntax. A ”Hello, World!” program is often the first written by a student of a new programming language, but such a program can also be used as a sanity check to ensure that the computer software intended to compile or run source code is correctly installed, and that its operator understands how to use it.

Python is known to be exceptionally user-friendly compared to other languages. To show, here is a Hello, World! written in C++.

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!";
    return 0;
}
```

And here is in Java, another major programming language:

```java
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

And finally, here is in Python:

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

From here we will start to work on the concepts of programming. While some parts are unique to Python and explained in its notation, the things you learn here will be applicable to the majority of the programming languages you will encounter in academic or industrial environments.
Here is a tiny code to get us started.

```python
username = input("Enter username:")
print("Welcome to the class: " + username + "!")
```

Chapter 2

Functions

Table of Contents

2.1 Introduction

We may be familiar with the mathematical concept of a function. A function is a relationship or mapping between one or more inputs and a set of outputs. In mathematics, a function is typically represented like this:

$$z = f(x, y),$$

where, f is a function operating on inputs x and y. The output of the function is z. However, programming functions are much more generalised and versatile compared to its mathematical counterpart. In fact, appropriate function definition and use is so

critical to proper software development that virtually all modern programming languages support both built-in and user-defined functions.

In programming, a function is defined as:

> *self-contained block of code encapsulating a specific task or a group of tasks.*

Previously, we have worked with some of the *built-in functions* provided by Python.

`id()` Takes one (1) argument and returns the object's unique integer identifier:

```python
s = 'foobar'
id(s)
```

`len()` Returns the length of the argument passed to it:

```python
a = ['foo', 'bar', 'baz', 'qux']
print(len(a))
```

```text
4
```

`any()` Takes an iterable as its argument and returns `True` if any of the items in the iterable are truthy and `False` otherwise:

```python
print(any([False, False, False]))
print(any([False, True, False]))
print(any(['bar' == 'baz', len('foo') == 4, 'qux' in {'foo', 'bar', 'baz'}]))
print(any(['bar' == 'baz', len('foo') == 3, 'qux' in {'foo', 'bar', 'baz'}]))
```

```text
False
True
False
True
```

Each of these built-in functions performs a specific task. The code that accomplishes the task is defined somewhere, but we don't need to know where or even how the code works. All we need to know about is the function's **interface**:

1. What **arguments** (if any) it takes
2. What **values** (if any) it returns

Then we call the function and pass the appropriate arguments. Program execution goes off to the designated body of code and does its useful thing. When the function is finished, execution returns to our code where it left off.

> The function may or may not return data for our code to use.

When we define our **own** function, it works just the same. From somewhere in our code, we'll call our Python function and program execution will transfer to the body of code that makes up the function.

When the function is finished, execution returns to the location where the function was called. Depending on how we designed the function's interface, data may be passed in when the function is called, and return values may be passed back when it finishes.

2.2 Importance of Python Functions

Almost all programming languages used today support a form of **user-defined functions**, although they aren't always called functions. In other languages, we may see them referred to as one of the following:

- Subroutines
- Procedures
- Methods
- Subprograms

We may start to wonder what is the big deal of defining our own function? There are several very good reasons. Let's go over a few now.

2.2.1 Abstraction and Re-usability

Suppose we write some code that does something useful. As we continue development, we find the task performed by that code is one we need often, in many different locations within our application.

What should we do?

Well, we could just replicate the code over and over again, using Ctrl-C & Ctrl-V.

Later on, we'll probably decide that the code in question needs to be modified. We'll either find something wrong with it that needs to be fixed, or we'll want to enhance

it in some way. If copies of the code are scattered all over our application, then we'll need to make the necessary changes in every location.

> At first, it may be a reasonable solution, but it's likely to be a maintenance nightmare. While our code editor may help by providing a search-and-replace function (such as Emacs replace-string), this method is error-prone, and we could easily introduce bugs into our code that will be difficult to find.

A better solution is to define a Python function that performs the task. Anywhere in our application that we need to accomplish the task, we simply call the function. Down the line, if we decide to change how it works, then we only need to change the code in one location, which is the place where the function is defined. The changes will automatically be picked up anywhere the function is called.

The abstraction of functionality into a function definition is an example of the **Don't Repeat Yourself** (DRY) Principle of software development.

This is arguably the strongest motivation for using functions.

2.2.2 Modularity

Functions allow complex processes to be broken up into smaller steps. For example, we have a program that reads in a file, processes the file contents, and then writes an output file. Our code could look like this:

```python
# Main program

# Code to read file in
<statement>
<statement>

# Code to process file
<statement>
<statement>

# Code to write file out
<statement>
<statement>
```

In this example, the main program is a bunch of code strung together in a long sequence, with whitespace and comments to help organize it. However, if the code were to get much lengthier and more complex, then we'd have an increasingly difficult time wrapping our head around it.

Or, we could structure the code more like the following:

```python
def read_file():
    # Code to read file in
    <statement>
    <statement>

def process_file():
    # Code to process file
    <statement>
    <statement>

def write_file():
    # Code to write file out
    <statement>
    <statement>

# Main program
read_file()
process_file()
write_file()
```

The above example is modularised. Instead of all the code being strung together, it's broken out into **separate functions**, each of which focuses on a specific task. Those tasks are read, process, and write. The main program now simply needs to call each of these in turn.

> The `def` keyword introduces a new Python function definition.

In life, we do this sort of thing all the time, even if we don't explicitly think of it that way. If we wanted to tackle any problem we'd divide the job into manageable steps.

Breaking a large task into smaller, bite-sized sub-tasks helps make the large task easier to think about and manage. As programs become more complicated, it becomes increasingly beneficial to modularize them in this way.

2.2.3 Namespace Separation

A **namespace** is a region of a program in which identifiers have meaning. As we'll see below, when a Python function is called, a new namespace is created for that function, one that is distinct from all other namespaces that already exist.

The practical upshot of this is that variables can be defined and used within a Python

function even if they have the same name as variables defined in other functions or in the main program. In these cases, there will be no confusion or interference because they're kept in separate namespaces.

This means that when we write code within a function, we can use variable names and identifiers without worrying about whether they're already used elsewhere outside the function. This helps minimize errors in code considerably.

2.3 Function Calls and Definition

The usual syntax for defining a Python function is as follows:

```python
def <function_name>([<parameters>]):
    <statement(s)>
```

The components of the definition are explained in the table below:

Component	Meaning
`def`	The keyword that informs Python that a function is being defined
`<function_name>`	A valid Python identifier that names the function
`<parameters>`	An optional, comma-separated list of parameters that may be passed to the function
`:`	Punctuation that denotes the end of the Python function header (the name and parameter list)
`<statement(s)>`	A block of valid Python statements

Table 2.1.: The components of what makes a function.

The final item, `<statement(s)>`, is called the **body of the function**. The body is a block of statements that will be executed when the function is called. The body of a Python function is defined by indentation in accordance with the off-side rule.

> This is the same as code blocks associated with a control structure, similar to an `if` or `while` statement.

The syntax for calling a Python function is as follows:

```python
<function_name>([<arguments>])
```

`<arguments>` are the values passed into the function. They correspond to the `<parameters>` in the Python function definition. We can define a function that doesn't take any arguments, but the parentheses are still required. Both a function definition and a function

call must **ALWAYS** include parentheses, even if they're empty.

As usual, we will start with a small example and add complexity from there. Keeping the mathematical tradition in mind, we will call our first Python function `f()`. Here's a code snippet defining and calling `f()`:

```python
def f():
    s = '-- Inside f()'
    print(s)

print('Before calling f()')
f()
print('After calling f()')
```

```text
Before calling f()
-- Inside f()
After calling f()
```

Here's how this code works:

1. First line uses the `def` keyword to indicate that a function is being defined. Execution of the `def` statement merely creates the definition of `f()`.
2. All the following lines that are indented become part of the body of `f()` and are stored as its definition, but they aren't executed yet.
3. Empty line is a bit of whitespace between the function definition and the first line of the main program. While it isn't syntactically necessary, it is nice to have.
4. This is the first statement that isn't indented because it isn't a part of the definition of `f()`. It's the start of the main program. When the main program executes, this statement is executed first.
5. This is a call to `f()`. Note that empty parentheses are always required in both a function definition and a function call, even when there are no parameters or arguments. Execution proceeds to `f()` and the statements in the body of `f()` are executed.
6. Here execute once the body of `f()` has finished. Execution returns to this `print()` statement.

Occasionally, we may want to define an empty function that does **nothing**. This is referred to as a **stub**, which is usually a temporary placeholder for a Python function that will be fully implemented at a later time. Just as a block in a control structure can't be empty, neither can the body of a function. To define a stub function, use the passstatement:

```python
def f():
    pass

f()
```

As we can see above, a call to a stub function is syntactically valid but doesn't do anything.

Example **Sum of Two Number** 1

Write a function which takes two values (say `a`, and `b`) and returns their sum.

Solution **Sum of Two Number**

```python
def sum_values(a, b):
    result = a + b
    return result

print(sum_values(1, 2))
```

Example **Maximum of Three Numbers** 2

Write a Python function to find the maximum of three numbers.

Solution **Maximum of Three Numbers**

```python
# Define a function that returns the maximum of two numbers
def max_of_two(x, y):
    # Check if x is greater than y
    if x > y:
        # If x is greater, return x
        return x
    # If y is greater or equal to x, return y
    return y

# Define a function that returns the maximum of three numbers
def max_of_three(x, y, z):
    # Call max_of_two function to find the maximum of y and z,
    # then compare it with x to find the overall maximum
    return max_of_two(x, max_of_two(y, z))

# Print the result of calling max_of_three function with arguments 3, 6, and -5
print(max_of_three(3, 6, -5))
```

Example **Sum of a list** 3

Write a Python function to sum all the numbers in a list.

- Sample List : (8, 2, 3, 0, 7)
- Expected Output : 20

Sum of a list **Solution**

```python
# Define a function named 'sum' that takes a list of numbers as input
def sum(numbers):
    # Initialize a variable 'total' to store the sum of numbers, starting at 0
    total = 0

    # Iterate through each element 'x' in the 'numbers' list
    for x in numbers:
        # Add the current element 'x' to the 'total'
        total += x

    # Return the final sum stored in the 'total' variable
    return total

# Print the result of calling the 'sum' function with a tuple of numbers (8, 2, 3, 0, 7)
print(sum((8, 2, 3, 0, 7)))
```

Multiply Values in a List 4 **Example**

Write a Python function to multiply all the numbers in a list.

- Sample List: (8, 2, 3, -1, 7)
- Expected Output : -336

Multiply Values in a List **Solution**

```python
# Define a function named 'multiply' that takes a list of numbers as input
def multiply(numbers):
    # Initialize a variable 'total' to store the multiplication result, starting at 1
    total = 1

    # Iterate through each element 'x' in the 'numbers' list
    for x in numbers:
        # Multiply the current element 'x' with the 'total'
        total *= x

```

```python
    # Return the final multiplication result stored in the 'total' variable
    return total

# Print the result of calling the 'multiply' function with a tuple of numbers (8, 2, 3, -1, 7)
print(multiply((8, 2, 3, -1, 7)))
```

Example **Printing a Sentence Multiple Times** 5

Write a Python function that prompts the user to enter a number. Then, given that number, it prints the sentence "Hello, Python!" that many times.

Solution **Printing a Sentence Multiple Times**

```python
def print_sentence_multiple_times():
    num = int(input("Enter a number: "))
    for _ in range(num):
        print("Hello, Python!")


print_sentence_multiple_times()
```

Example **Function to Make a List** 6

Write a function that accepts different values as parameters and returns a list.

Solution **Function to Make a List**

```python
def myFruits(f1,f2,f3,f4):
    FruitsList = [f1,f2,f3,f4]
    return FruitsList

output = myFruits("Apple","Bannana","Grapes","Orange")
print(output)
```

Example **Reversing a String** 7

Write a Python program to reverse a string.

- Sample String : "1234abcd"
- Expected Output : "dcba4321"

Solution **Reversing a String**

```python
# Define a function named 'string_reverse' that takes a string 'str1' as input
def string_reverse(str1):
    # Initialize an empty string 'rstr1' to store the reversed string
    rstr1 = ''

    # Calculate the length of the input string 'str1'
    index = len(str1)

    # Execute a while loop until 'index' becomes 0
    while index > 0:
        # Concatenate the character at index - 1 of 'str1' to 'rstr1'
        rstr1 += str1[index - 1]

        # Decrement the 'index' by 1 for the next iteration
        index = index - 1

    # Return the reversed string stored in 'rstr1'
    return rstr1

# Print the result of calling the 'string_reverse' function with the input string '1234abcd'
print(string_reverse('1234abcd'))
```

Counting Cases 8 **Example**

Write a Python function that accepts a string and counts the number of upper and lower case letters.

- Sample String : 'The quick Brow Fox'
- Expected Output :
- No. of Upper case characters : 3
- No. of Lower case Characters : 12

Counting Cases **Solution**

```python
# Define a function named 'string_test' that counts the number of upper and lower case characters in a string 's'
def string_test(s):
    # Create a dictionary 'd' to store the count of upper and lower case characters
    d = {"UPPER_CASE": 0, "LOWER_CASE": 0}

    # Iterate through each character 'c' in the string 's'
    for c in s:
        # Check if the character 'c' is in upper case
        if c.isupper():
```

```python
            # If 'c' is upper case, increment the count of upper case characters in
            ↪ the dictionary
            d["UPPER_CASE"] += 1
        # Check if the character 'c' is in lower case
        elif c.islower():
            # If 'c' is lower case, increment the count of lower case characters in
            ↪ the dictionary
            d["LOWER_CASE"] += 1
        else:
            # If 'c' is neither upper nor lower case (e.g., punctuation, spaces),
            ↪ do nothing
            pass

    # Print the original string 's'
    print("Original String: ", s)

    # Print the count of upper case characters
    print("No. of Upper case characters: ", d["UPPER_CASE"])

    # Print the count of lower case characters
    print("No. of Lower case Characters: ", d["LOWER_CASE"])

# Call the 'string_test' function with the input string 'The quick Brown Fox'
string_test('The quick Brown Fox')
```

2.4 Argument Passing

So far, the functions we have defined haven't taken any arguments. That can sometimes be useful, and we'll occasionally write such functions. More often, though, we'll want to pass data into a function so that its behaviour can vary from one invocation to the next. Let's see how to do that.

2.4.1 Positional Arguments

The most straightforward way to pass arguments to a Python function is with positional arguments (it is also called required arguments). In the function definition, we specify a comma-separated list of parameters inside the parentheses:

```python
def f(qty, item, price):
    print(f'{qty} {item} cost ${price:.2f}')
```

When the function is called, we specify a corresponding list of arguments:

```python
f(6, 'bananas', 1.74)
```

```text
6 bananas cost $1.74
```

The parameters (`qty`, `item`, and `price`) behave like variables that are defined locally to the function. When the function is called, the arguments that are passed are bound to the parameters in order, as though by variable assignment:

Parameter	Argument
qty	6
item	bananas
price	1.74

Table 2.2.: A closer look at the parameters.

> In some programming texts, the parameters given in the function definition are referred to as formal parameters, and the arguments in the function call are referred to as actual parameters

Although positional arguments are the most straightforward way to pass data to a function, they also afford the least flexibility. To start, the order of the arguments in the call must match the order of the parameters in the definition. There's nothing to stop us from specifying positional arguments out of order, of course:

```python
f('bananas', 1.74, 6)
```

The function may even still run, as it did in the example above, but it's very unlikely to produce the correct results. It's the responsibility of the programmer who defines the function to document what the appropriate arguments should be, and it's the responsibility of the user of the function to be aware of that information and abide by it.

With positional arguments, the arguments in the call and the parameters in the definition must agree not only in order but in number as well. That's the reason positional arguments are also referred to as **required arguments**. We can't leave any out when calling the function:

```python
# Too few arguments and would give an error
f(6, 'bananas')
```

We can't also specify extra ones:

```python
# Too many arguments
f(6, 'bananas', 1.74, 'kumquats')
```

Positional arguments are conceptually straightforward to use, but they're not very forgiving.

> We must specify the same number of arguments in the function call as there are parameters in the definition, and in exactly the same order.

2.4.2 Keyword Arguments

When we're calling a function, we can specify arguments in the form `<keyword>=<value>`. In that case, each `<keyword>` must match a parameter in the Python function definition. For example, the previously defined function `f()` may be called with keyword arguments as follows:

```python
f(qty=6, item='bananas', price=1.74)
```

Referencing a keyword that doesn't match any of the declared parameters generates an exception:

```python
f(qty=6, item='bananas', cost=1.74)
```

> Using keyword arguments lifts the restriction on argument order.

Each keyword argument explicitly designates a specific parameter by name, so we can specify them in any order and Python will still know which argument goes with which parameter:

```python
f(item='bananas', price=1.74, qty=6)
```

Like with positional arguments, though, the number of arguments and parameters must **still match**:

```python
# Still too few arguments
f(qty=6, item='bananas')
```

So, keyword arguments allow flexibility in the order that function arguments are specified, but the number of arguments is still rigid.

We can call a function using both positional and keyword arguments:

```python
print(f(6, price=1.74, item='bananas'))
print(f(6, 'bananas', price=1.74))
```

When positional and keyword arguments are both present, all the positional arguments must come first:

```python
f(6, item='bananas', 1.74)
```

Once we've specified a keyword argument, there can't be any positional arguments to the right of it.

2.4.3 Default Parameters

If a parameter specified in a Python function definition has the form `<name>=<value>`, then `<value>` becomes a default value for that parameter. Parameters defined this way are referred to as **default** or **optional** parameters. An example of a function definition with default parameters is shown below:

```python
def f(qty=6, item='bananas', price=1.74):
    print(f'{qty} {item} cost ${price:.2f}')
```

When this version of `f()` is called, any argument that's left out assumes its default value:

```python
print(f(4, 'apples', 2.24))
print(f(4, 'apples'))
print(f(4))
print(f())
print(f(item='kumquats', qty=9))
print(f(price=2.29))
```

2.4.4 Mutable Default Parameter Values

Things can get weird if we specify a default parameter value that is a **mutable** object. Consider this Python function definition:

mutable is the ability of objects to change their values

```python
def f(my_list=[]):
    my_list.append('###')
    return my_list
```

Here, `f()` takes a single list parameter, appends the string '###' to the end of the list, and returns the result:

```python
print(f(['foo', 'bar', 'baz']))
print(f([1, 2, 3, 4, 5]))
```

```text
['foo', 'bar', 'baz', '###']
[1, 2, 3, 4, 5, '###']
```

The default value for parameter `my_list` is the empty list, so if `f()` is called without any arguments, then the return value is a list with the single element '###':

```python
print(f())
```

```text
['###']
```

Everything makes sense so far. Now, what would we expect to happen if `f()` is called without any parameters a second and a third time? Let's see:

```python
print(f())
print(f())
```

```text
['###', '###']
['###', '###', '###']
```

We expected each subsequent call to also return the singleton list ['###'], just like the first. Instead, the return value keeps growing. What happened?

In Python, default parameter values are defined only once when the function is defined (that is, when the def statement is executed). The default value isn't re-defined each time the function is called.

Thus, each time we call `f()` without a parameter, we're performing `.append()` on the same list.
We can demonstrate this with `id()`:

```python
def f(my_list=[]):
    print(id(my_list))
    my_list.append('###')
    return my_list
```

```python
print(f())
print(f())
print(f())
```

```text
4310647808
['###']
4310647808
['###', '###']
4310647808
['###', '###', '###']
```

The object identifier displayed confirms , when `my_list` is allowed to default, the value is the same object with each call. As lists are mutable, each subsequent `.append()` call causes the list to get longer. This is a common and pretty well-documented pitfall when we're using a mutable object as a parameter's default value. It potentially leads to confusing code behavior, and is probably best avoided.

As a workaround, consider using a default argument value that signals no argument has been specified. Most any value would work, but None is a common choice. When the sentinel value indicates no argument is given, create a new empty list inside the function:

```python
def f(my_list=None):
    if my_list is None:
        my_list = []
    my_list.append('###')
    return my_list

print(f())
print(f())
print(f())
print(f(['foo', 'bar', 'baz']))
print(f([1, 2, 3, 4, 5]))
```

```text
['###']
['###']
['###']
['foo', 'bar', 'baz', '###']
[1, 2, 3, 4, 5, '###']
```

This ensures that my_{list} now truly defaults to an empty list whenever `f()` is called without an argument.

Example **Celcius to Fahrenheit** 9

Write a program which converts celcius to fahrenheit.

$$F = \left(C\frac{9}{5}\right) + 32$$

Solution **Celcius to Fahrenheit**

```python
# creating a function that converts the given celsius degree temperature
# to Fahrenheit degree temperature
def convertCelsiustoFahrenheit(c):
    # converting celsius degree temperature to Fahrenheit degree temperature
    f = (9/5)*c + 32
    # returning Fahrenheit degree temperature of given celsius temperature
    return (f)
# input celsius degree temperature
celsius_temp = 80
print("The input Temperature in Celsius is ",celsius_temp)
# calling convertCelsiustoFahrenheit() function by passing
# the input celsius as an argument
fahrenheit_temp = convertCelsiustoFahrenheit(celsius_temp)
# printing the Fahrenheit equivalent of the given celsius degree temperature
print("The Fahrenheit equivalent of input celsius degree = ", fahrenheit_temp)
```

2.5 The return Statement

As a tradition to let the user know everything worked, functions return 0 and if an error has occurred, they would return 1. This originally comes from UNIX computers.

What's a Python function to do then? After all, in many cases, if a function doesn't cause some change in the calling environment, then there isn't much point in calling it at all. How should a function affect its caller?

Well, one possibility is to use function **return** values. A return statement in a Python function serves two purposes:

- It immediately terminates the function and passes execution control back to the caller.
- It provides a mechanism by which the function can pass data back to the caller.

2.5.1 Exiting a Function

Within a function, a return statement causes immediate exit from the Python function and transfer of execution back to the caller:

```python
def f():
    print('foo')
    print('bar')
    return

print(f())
```

```text
foo
bar
None
```

In this example, the return statement is actually superfluous (i.e., unnecessary). A function will return to the caller when it falls off the end—that is, after the last statement of the function body is executed.

> So, this function would behave identically without the return statement.

However, return statements don't need to be at the end of a function. They can appear anywhere in a function body, and even multiple times.
Consider the following example:

```python
def f(x):
    if x < 0:
        return
    if x > 100:
        return
    print(x)

print(f(-3))
print(f(105))
print(f(64))
```

```text
None
None
64
None
```

The first two calls to `f()` don't cause any output, because a return statement is executed and the function exits prematurely, before the `print()` statement is reached.
This sort of paradigm can be useful for error checking in a function. We can check several error conditions at the start of the function, with return statements that bail out if there's a problem:

```python
def f():
    if error_cond1:
        return
    if error_cond2:
        return
    if error_cond3:
        return

    <normal processing>
```

If none of the error conditions are encountered, then the function can proceed with its normal processing.

2.5.2 Returning Data to the Caller

In addition to exiting a function, the return statement is also used to pass data back to the caller. If a return statement inside a Python function is followed by an expression, then in the calling environment, the function call evaluates to the value of that expression:

```python
def f():
    return 'foo'

s = f()
s
```

Here, the value of expression `f()` is `foo`, which is subsequently assigned to variable `s`.

You might be wondering the use of foo and bar. The terms "foo" and "bar" come from FUBAR, a military acronym meaning Fucked Up Beyond All Repair, a phrase which applies to many pieces of software. Leave it to programmers to have a dry sense of humour!

A function can return any type of object. In Python, that means pretty much anything whatsoever. In the calling environment, the function call can be used syntactically in any way that makes sense for the type of object the function returns.

For example, in the code below, `f()` returns a dictionary. In the calling environment then, the expression `f()` represents a dictionary, and `f()['baz']` is a valid key reference into that dictionary:

```python
def f():
    return dict(foo=1, bar=2, baz=3)

print(f())
print(f()['baz'])
```

```
{'foo': 1, 'bar': 2, 'baz': 3}
3
```
text

In the next example, `f()` returns a string that we can slice like any other string:

```python
def f():
    return 'foobar'

print(f()[2:4])
```

```
ob
```
text

Here, `f()` returns a list that can be indexed or sliced:

```python
def f():
    return ['foo', 'bar', 'baz', 'qux']


print(f())
print(f()[2])
print(f()[::-1])
```

```
['foo', 'bar', 'baz', 'qux']
baz
['qux', 'baz', 'bar', 'foo']
```
text

If multiple comma-separated expressions are specified in a return statement, then they're packed and returned as a `tuple`:

```python
def f():
    return 'foo', 'bar', 'baz', 'qux'

print(type(f()))

t = f()
print(t)

a, b, c, d = f()
print(f'a = {a}, b = {b}, c = {c}, d = {d}')
```

```
<class 'tuple'>
('foo', 'bar', 'baz', 'qux')
```
text

```
a = foo, b = bar, c = baz, d = qux
```

When no return value is given, a Python function returns the special Python value None:

```python
def f():
    return

print(f())
```

```text
None
```

The same thing happens if the function body doesn't contain a return statement at all and the function falls off the end:

```python
def g():
    pass

print(g())
```

```text
None
```

> None is treated as `False` when evaluated in a Boolean context.

Since functions that exit through a bare return statement or fall off the end return `None`, a call to such a function can be used in a Boolean context:

```python
def f():
    return

def g():
    pass

if f() or g():
    print('yes')
else:
    print('no')
```

```text
no
```

In the code above, calls to both `f()` and `g()` are falsy, so `f()` or `g()` is as well, and the

`else` clause executes.

2.6 Variable-Length Argument Lists

In some cases, when we're defining a function, we may not know beforehand how many arguments we'll want it to take. Suppose, for example, that we want to write a Python function that computes the average of several values. We could start with something like this:

```python
def avg(a, b, c):
    return (a + b + c) / 3
```

All is well if we want to average just three (3) values:

```python
print(avg(1, 2, 3))
```

```text
2.0
```

However, as we've already seen, when **positional arguments** are used, the number of arguments passed must agree with the number of parameters declared. Clearly then, all isn't well with this implementation of `avg()` for any number of values that is not three:

```python
avg(1, 2, 3, 4)
```

We could try to define `avg()` with optional parameters:

```python
def avg(a, b=0, c=0, d=0, e=0):
    .
    .
    .
```

This allows for a variable number of arguments to be specified. The following calls are at least **syntactically** correct:

```python
avg(1)
avg(1, 2)
avg(1, 2, 3)
avg(1, 2, 3, 4)
avg(1, 2, 3, 4, 5)
```

But this approach still suffers from a couple of problems. For starters, it still only allows up to five arguments, not an arbitrary number. Worse yet, there's no way to distinguish between the arguments that were specified and those that were allowed to default.

The function has no way to know how many arguments were actually passed, so it doesn't know what to divide by:

```python
def avg(a, b=0, c=0, d=0, e=0):
    return (a + b + c + d + e) / # Divided by what???
```

Evidently, this won't do either.
We could write `avg()` to take a single list argument:

```python
def avg(a):
    total = 0
    for v in a:
            total += v
    return total / len(a)


print(avg([1, 2, 3]))
print(avg([1, 2, 3, 4, 5]))
```

This works. It allows an arbitrary number of values and produces a correct result. As an added bonus, it works when the argument is a tuple as well:

```python
t = (1, 2, 3, 4, 5)
print(avg(t))
```

The drawback is that the added step of having to group the values into a list or tuple is probably not something the user of the function would expect, and it isn't very elegant. Whenever we find Python code that looks inelegant, there's probably a better option.

In this case, Python provides a way to pass a function a variable number of arguments with argument tuple packing and unpacking using the asterisk (*) operator.

2.6.1 Argument Tuple Packing

When a parameter name in a Python function definition is preceded by an asterisk (*), it indicates argument tuple packing. Any corresponding arguments in the function

call are packed into a tuple that the function can refer to by the given parameter name.

Here's an example:

```python
def f(*args):
    print(args)
    print(type(args), len(args))
    for x in args:
            print(x)

print(f(1, 2, 3))
print(f('foo', 'bar', 'baz', 'qux', 'quux'))
```

```text
(1, 2, 3)
<class 'tuple'> 3
1
2
3
None
('foo', 'bar', 'baz', 'qux', 'quux')
<class 'tuple'> 5
foo
bar
baz
qux
quux
None
```

In the definition of `f()`, the parameter specification `*args` indicates tuple packing. In each call to `f()`, the arguments are packed into a tuple that the function can refer to by the name args. Any name can be used, but `args` is so commonly chosen that it's practically a standard.

Using tuple packing, we can clean up `avg()` like this:

```python
def avg(*args):
    total = 0
    for i in args:
        total += i
    return total / len(args)

print(avg(1, 2, 3))
print(avg(1, 2, 3, 4, 5))
```

```python
def avg(*args):
    total = 0
    for i in args:
        total += i
    return total / len(args)

print(avg(1, 2, 3))
print(avg(1, 2, 3, 4, 5))
```

Better still, we can tidy it up even further by replacing the for loop with the built-in Python function `sum()`, which sums the numeric values in any iterable:

```python
def avg(*args):
    return sum(args) / len(args)

print(avg(1, 2, 3))
print(avg(1, 2, 3, 4, 5))
```


```python
def avg(*args):
    return sum(args) / len(args)

print(avg(1, 2, 3))
print(avg(1, 2, 3, 4, 5))
```


Now, `avg()` is concisely written and works as intended.

Still, depending on how this code will be used, there may still be work to do. As written, `avg()` will produce a TypeError exception if any arguments are non-numeric:

TypeError is raised whenever an operation is performed on an incorrect/unsupported object type.

```python
avg(1, 'foo', 3)
```

To be as robust as possible, we could add code to check that the arguments are of the proper type.

2.6.2 Argument Tuple Unpacking

An analogous operation is available on the other side of the equation in a Python function call. When an argument in a function call is preceded by an asterisk (*), it indicates that the argument is a tuple that should be unpacked and passed to the function as separate values:

```python
def f(x, y, z):
    print(f'x = {x}')
    print(f'y = {y}')
    print(f'z = {z}')

print(f(1, 2, 3))

t = ('foo', 'bar', 'baz')
print(f(*t))
```

```text
x = 1
y = 2
z = 3
None
x = foo
y = bar
z = baz
None
```

In this example, `*t` in the function call indicates that `t` is a tuple that should be unpacked. The unpacked values `'foo'`, `'bar'`, and `'baz'` are assigned to the parameters `x`, `y`, and `z`, respectively.

Although this type of unpacking is called tuple unpacking, it doesn't only work with tuples. The asterisk (`*`) operator can be applied to any iterable in a Python function call. For example, a list or set can be unpacked as well:

```python
a = ['foo', 'bar', 'baz']

print(type(a))
print(f(*a))

s = {1, 2, 3}
print(type(s))
print(f(*s))
```

```text
<class 'list'>
x = foo
y = bar
z = baz
None
<class 'set'>
x = 1
y = 2
```

```
z = 3
None
```

2.6.3 Argument Dictionary Unpacking

Argument dictionary unpacking is analogous to argument tuple unpacking. When the double asterisk (`**`) precedes an argument in a Python function call, it specifies that the argument is a dictionary that should be unpacked, with the resulting items passed to the function as keyword arguments:

```python
def f(a, b, c):
    print(F'a = {a}')
    print(F'b = {b}')
    print(F'c = {c}')

d = {'a': 'foo', 'b': 25, 'c': 'qux'}
f(**d)
```

The items in the dictionary `d` are unpacked and passed to `f()` as keyword arguments. So, `f(**d)` is equivalent to `f(a='foo', b=25, c='qux')`:

```python
f(a='foo', b=25, c='qux')
```

In fact, we can see it in the code below

```python
f(**dict(a='foo', b=25, c='qux'))
```

```text
a = foo
b = 25
c = qux
```

2.6.4 Putting it all together

Think of `*args` as a variable-length positional argument list, and `**kwargs` as a variable-length keyword argument list.

All three—standard positional parameters, `*args`, and `**kwargs`—can be used in one Python function definition. If so, then they should be specified in that order:

```python
def f(a, b, *args, **kwargs):
    print(F'a = {a}')
    print(F'b = {b}')
    print(F'args = {args}')
    print(F'kwargs = {kwargs}')

f(1, 2, 'foo', 'bar', 'baz', 'qux', x=100, y=200, z=300)
```

```text
a = 1
b = 2
args = ('foo', 'bar', 'baz', 'qux')
kwargs = {'x': 100, 'y': 200, 'z': 300}
```

This provides just about as much flexibility as you could ever need in a function interface.

Chapter

3

Object Oriented Programming

Table of Contents

3.1 Introduction

Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviours into individual objects. In this chapter, we'll focus the basics of object-oriented programming in Python.

Terminology invoking "objects" in the modern sense of object-oriented programming made its first appearance at the artificial intelligence group at MIT in the late 1950s and early 1960s. "Object" referred to LISP atoms with identified properties (attributes)

Conceptually, objects are like the components of a system. Think of a program as a factory assembly line of sorts. At each step of the assembly line, a system component processes some material, ultimately transforming raw material into a finished product.

An object contains data, like the raw or pre-processed materials at each step on an assembly line. In addition, the object contains behaviour, like the action that each assembly line component performs.

3.2 Defining Object Oriented Programming

Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviours are bundled into individual objects.

> Up to know what we were writing is what is called **imperative programming**.

For example, an object could represent a person with properties like a name, age, and address and behaviours such as walking, talking, breathing, and running. Or it could represent an email with properties like a recipient list, subject, and body and behaviours like adding attachments and sending.

Put another way, object-oriented programming is an approach for modelling concrete, real-world things, like cars, as well as relations between things, like companies and employees or students and teachers. OOP models real-world entities as software objects that have some data associated with them and can perform certain operations.

> Objects are at the centre of object-oriented programming in Python. In other programming paradigms, objects only represent the data. In OOP, they additionally inform the overall structure of the program.

3.3 Defining a Class

In Python, we define a class by using the `class` keyword followed by a name and a colon (`:`). Then we use `.__init__()` to declare which attributes each instance of the class we would like to have have:

```python
class Employee:
    def __init__(self, name, age):
        self.name =  name
        self.age = age
```

But what does all of that mean? And why do we even need classes in the first place? Lets' take a step back and consider using built-in, primitive data structures as an alternative.

Primitive data structures (numbers, strings, and lists) are designed to represent straightforward pieces of information, such as the cost of an apple, the name of a poem, or your favourite colours, respectively.

What if we want to represent something more complex?

For example, we might want to track employees in an organisation. You need to store some basic information about each employee, such as their name, age, position, and the year they started working.

One way to do this is to represent each employee as a list:

```python
kirk = ["James Kirk", 34, "Captain", 2265]
spock = ["Spock", 35, "Science Officer", 2254]
mccoy = ["Leonard McCoy", "Chief Medical Officer", 2266]
```

There are a number of issues with this approach.

- it can make larger code files more difficult to manage. If we reference `kirk[0]` several lines away from where we declared the kirk list, will we remember that the element with index 0 is the employee's name?
- it can introduce errors if employees don't have the same number of elements in their respective lists. In the mccoy list above, the age is missing, so `mccoy[1]` will return "Chief Medical Officer" instead of Dr. McCoy's age.

A great way to make this type of code more manageable and more maintainable is to use classes.

3.4 Classes v. Instances

Classes allow we to create **user-defined data structures**. Classes define functions called methods, which identify the behaviours and actions that an object created from the class can perform with its data.

In this chapter, we'll create a Dog class that stores some information about the characteristics and behaviours that an individual dog can have.

A class is a **blueprint** for how to define something. It doesn't actually contain any data. The Dog class specifies that a name and an age are necessary for defining a dog, but it doesn't contain the name or age of any specific dog.

While the class is the blueprint, an instance is an object that's built from a class and contains real data. An instance of the Dog class is not a blueprint anymore. It's an actual dog with a name, like Miles, who's four years old.

> a class is like a form or questionnaire. An instance is like a form that we've filled out with information. Just like many people can fill out the same form with their own unique information, we can create many instances from a single class.

3.5 Class Definition

We start all class definitions with the `class` keyword, then add the name of the class and a colon (`:`). Python will consider any code that we indent below the class definition as part of the class's body.

Here's an example of a Dog class:

```python
class Dog:
    pass
```

The body of the Dog class consists of a single statement:

the `pass` keyword.

Python programmers often use pass as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error.

> Python class names are written in **CapitalizedWords** notation by convention. For example, a class for a specific breed of dog, like the Jack Russell Terrier, would be written as `JackRussellTerrier`.

The Dog class isn't very interesting right now, so we'll spruce it up a bit by defining some properties that all Dog objects should have. There are several properties that you can choose from, including:

- name,
- age,
- coat color,
- breed, ...

To keep the example small in scope, we'll just use **name** and **age**.

We define the properties all Dog objects must have in a method called `.__init__()`. Every time we create a new Dog object, `.__init__()` sets the initial state of the object by assigning the values of the object's properties.

`.__init__()` initializes each new instance of the class.

The Role of `.__init__()`

`.__init__()` doesn't initialize a class, it initializes an instance of a class or an object. Each dog has colour, but dogs as a class don't. Each dog has four or fewer feet, but the class of dogs doesn't. The class is a concept of an object. When you see Fido and Spot, you recognise their similarity, their doghood. That's the class.

The `__init__()` function is called a constructor, or initializer, and is automatically called when you create a new instance of a class. Within that function, the newly created object is assigned to the parameter `self`. The notation `self.name` is an attribute called name of the object in the variable self. Attributes are kind of like variables, but they describe the state of an object, or particular actions (functions) available to the object.

You can give `.__init__()` any number of parameters, but the first parameter will **always** be a variable called `self`. When you create a new class instance, then Python automatically passes the instance to the self parameter in `.__init__()` so that Python can define the new attributes on the object.

The role of `self`

The reason you need to use `self.` is because Python does not use special syntax to refer to instance attributes. Python decided to do methods in a way that makes the instance to which the method belongs be passed automatically, but not received automatically: the first parameter of methods is the instance the method is called on. That makes methods entirely the same as functions, and leaves the actual name to use up to the programmer (although self is the convention, and people will generally frown at you when you use something else.) self is not special to the code, it's just another object.

We update the Dog class with an `.__init__()` method that creates `.name` and `.age` attributes:

```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Make sure that you indent the `.__init__()` method's signature by four spaces, and the body of the method by eight spaces.

> This indentation is vitally important. It tells Python that the `.__init__()` method belongs to the Dog class.

In the body of `.__init__()`, there are two (2) statements using the `self` variable:

1. `self.name = name` creates an attribute called name and assigns the value of the name parameter to it.
2. `self.age = age` creates an attribute called age and assigns the value of the age parameter to it.

Attributes created in `.__init__()` are called **instance attributes**. An instance attribute's value is specific to a particular instance of the class. All Dog objects have a name and an age, but the values for the name and age attributes will vary depending on the Dog instance.

On the other hand, class attributes are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of `.__init__()`.

For example, the following Dog class has a class attribute called species with the value *Canis familiaris*:

```python
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

You define class attributes directly beneath the first line of the class name and indent them by four spaces. You always need to assign them an initial value. When you create an instance of the class, then Python automatically creates and assigns class attributes to their initial values.

> Use class attributes to define properties that should have the same value for every class instance. Use instance attributes for properties that vary from one instance to another.

Now that you have a Dog class, it's time to create some dogs!

Creating a Class with no Attributes 10 **Example**

Create a Vehicle class without any variables and methods.

Creating a Class with no Attributes **Solution**

```python
class Vehicle:
    pass
```

3.6 To Instance a Class

Creating a new object from a class is called **instantiating a class**. You can create a new object by typing the name of the class, followed by opening and closing parentheses:

```python
class Dog:
    pass

print(Dog())
```

We first create a new Dog class with **NO** attributes or methods, and then we instantiate the Dog class to create a Dog object.

```text
<__main__.Dog object at 0x100dda240>
```

In the output above, you can see that you now have a new Dog object at `0x100dda240`. This funny-looking string of letters and numbers is a memory address that indicates where Python stores the Dog object in your computer's memory.

> The address on your screen will be different.

Now instantiate the Dog class a second time to create another Dog object:

```python
print(Dog())
```

```text
<__main__.Dog object at 0x100cdbe60>
```

The new Dog instance is located at a **different memory address**. That's because it's an entirely new instance and is completely unique from the first Dog object that we created.

To see this another way, type the following:

```python
a = Dog()
b = Dog()
print(a == b)
```

```text
False
```

In this code, we create two (2) new Dog objects and assign them to the variables a and b. When you compare a and b using the == operator, the result is False. Even though a and b are both instances of the Dog class, they represent two distinct objects in memory.

Example **Creating a Class** 11

Write a Python program to create a `Vehicle` class with `max_speed` and mileage instance attributes.

Solution **Creating a Class**

```python
class Vehicle:
    def __init__(self, max_speed, mileage):
        self.max_speed = max_speed
        self.mileage = mileage

modelX = Vehicle(240, 18)
print(modelX.max_speed, modelX.mileage)
```

3.7 Class and Instance Attributes

```python
class Dog:
    species = "Canis familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

To instantiate this Dog class, we need to provide values for name and age.

> If we don't do it, then Python raises a `TypeError`.

To pass arguments to the name and age parameters, put values into the parentheses after the class name:

```python
miles = Dog("Miles", 4)
buddy = Dog("Buddy", 9)
```

This creates two (2) new Dog instances:

1. four-year-old dog named Miles
2. nine-year-old dog named Buddy

The Dog class's `.__init__()` method has three (3) parameters, so why are you only passing two arguments to it in the example?

When you instantiate the Dog class, Python creates a new instance of Dog and passes it to the first parameter of `.__init__()`. This essentially removes the self parameter, so you only need to worry about the name and age parameters.

> Behind the scenes, Python both creates and initializes a new object when you use this syntax.

After we create the Dog instances, you can access their instance attributes using dot (`.`) notation:

```python
print(miles.name)
print(miles.age)
print(buddy.name)
print(buddy.age)
```

```text
Miles
4
Buddy
9
```

You can access class attributes the same way:

```python
print(buddy.species)
```

```text
Canis familiaris
```

A great advantage of using classes to organise data is that instances are guaranteed to have the attributes you expect. All Dog instances have:

- `.species`

- `.name`
- `.age`

attributes, so you can use those attributes with confidence, knowing that they'll always return a value. Although the attributes are guaranteed to exist, their values can change dynamically:

```python
buddy.age = 10
print(buddy.age)

miles.species = "Felis silvestris"
print(miles.species)
```

```text
10
Felis silvestris
```

In this example, we changed the `.age` attribute of the buddy object to `10`. Then you change the `.species` attribute of the miles object to *Felis silvestris*, which is a species of cat.

The key takeaway here is that custom objects are **mutable by default**. An object is mutable if you can alter it dynamically. For example, lists and dictionaries are mutable, but strings and tuples are immutable.

3.7.1 Instance Methods

Instance methods are functions that you define inside a class and can only call on an instance of that class. Similar to `.__init__()`, an instance method always takes self as its first parameter. Let's start our code by typing in the following Dog class:

```python
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, sound):
```

```python
        return f"{self.name} says {sound}"
```

This Dog class has two (2) instance methods:

1. `.description()` returns a string displaying the name and age of the dog.
2. `.speak()` has one parameter called sound and returns a string containing the dog's name and the sound that the dog makes.

Then we type the following to see our instance methods in action:

```python
miles = Dog("Miles", 4)

print(miles.description())
print(miles.speak("Woof Woof"))
print(miles.speak("Bow Wow"))
```

```text
Miles is 4 years old
Miles says Woof Woof
Miles says Bow Wow
```

In the above Dog class, `.description()` returns a string containing information about the Dog instance miles. When writing your own classes, it's a good idea to have a method that returns a string containing useful information about an instance of the class.

However, `.description()` isn't the most Pythonic way of doing this.
When you create a list object, you can use `print()` to display a string that looks like the list:

```python
names = ["Miles", "Buddy", "Jack"]
print(names)
```

```text
['Miles', 'Buddy', 'Jack']
```

Go ahead and print the miles object to see what output you get:

```python
print(miles)
```

```text
<__main__.Dog object at 0x100cdbe60>
```

When you print miles, you get a cryptic-looking message telling you that miles is a Dog object at the memory address 0x00aeff70. This message isn't very helpful. You can change what gets printed by defining a special instance method called `.__str()__`. Let's change the name of the Dog class's `.description()` method to `.__str()__`:

```python
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old"

    def speak(self, sound):
         return f"{self.name} says {sound}"
```

Now, when we print miles, we get a much friendlier output:

```python
miles = Dog("Miles", 4)
print(miles)
```

```text
Miles is 4 years old
```

Methods like `.__init__()` and `.__str__()` are called **dunder methods** as they begin and end with double underscores. There are many dunder methods that you can use to customise classes in Python. Understanding dunder methods is an important part of mastering object-oriented programming in Python, but for our first exploration of the topic, you'll stick with these two dunder methods.

3.8 Inheritance

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called child classes, and the classes that you derive child classes from are called parent classes.

We inherit from a parent class by creating a new class and putting the name of the parent class into parentheses:

```python
class Parent:
    hair_color = "brown"

class Child(Parent):
    pass
```

In this minimal example, the child class `Child` inherits from the parent class Parent. Because child classes take on the attributes and methods of parent classes, `Child.hair_color` is also brown without your explicitly defining that.

Child classes can override or extend the attributes and methods of parent classes. In other words:

> Child classes inherit all of the parent's attributes and methods but can also specify attributes and methods that are unique to themselves.

Although the analogy isn't perfect, you can think of object inheritance sort of like genetic inheritance.

You may have inherited your hair color from your parents. It's an attribute that you were born with. But maybe you decide to color your hair purple. Assuming that your parents don't have purple hair, you've just overridden the hair color attribute that you inherited from your parents:

```python
class Parent:
    hair_color = "brown"

class Child(Parent):
    hair_color = "purple"
```

If you change the code example like this, then `Child.hair_color` will be "purple". You also inherit, in a sense, your language from your parents. If your parents speak English, then you'll also speak English. Now imagine you decide to learn a second language, like German. In this case, you've extended your attributes because you've added an attribute that your parents don't have:

```python
class Parent:
    speaks = ["English"]

class Child(Parent):
    def __init__(self):
        super().__init__()
        self.speaks.append("German")
```

You'll learn more about how the code above works in the sections below. But before you dive deeper into inheritance in Python, we'll take a walk to a dog park to better understand why we might want to use inheritance in our own code.

The `super()` Function

Used to refer to the parent class or superclass. It allows you to call methods defined in the superclass from the subclass, enabling you to extend and customize the functionality inherited from the parent class.

Example **Class Inheritance** 12

Create a Bus class that inherits from the Vehicle class. Give the capacity argument of `Bus.seating_capacity()` a default value of `50`.
Use the following code for your parent Vehicle class.

```python
class Vehicle:
    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage

    def seating_capacity(self, capacity):
        return f"The seating capacity of a {self.name} is {capacity} passengers"
```

Solution **Class Inheritance**

```python
class Vehicle:
    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage

    def seating_capacity(self, capacity):
        return f"The seating capacity of a {self.name} is {capacity} passengers"
```

3.9 Parent Classes v. Child Classes

In this section, we'll create a child class for each of the three (3) breeds mentioned above:

- Jack Russell terrier,
- dachshund,

- and bulldog.

For reference, here's the full definition of the Dog class that we're currently working with:

```python
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old"

    def speak(self, sound):
        return f"{self.name} says {sound}"
```

To create a child class, you create a new class with its own name and then put the name of the parent class in parentheses. Add the following lines to create three (3) new child classes of the Dog class:

```python
class JackRussellTerrier(Dog):
    pass

class Dachshund(Dog):
    pass

class Bulldog(Dog):
    pass
```

With the child classes defined, you can now create some dogs of specific breeds:

```python
miles = JackRussellTerrier("Miles", 4)
buddy = Dachshund("Buddy", 9)
jack = Bulldog("Jack", 3)
jim = Bulldog("Jim", 5)
```

Instances of child classes inherit all of the attributes and methods of the parent class:

```python
miles.species
buddy.name
print(jack)
jim.speak("Woof")
```

To determine which class a given object belongs to, you can use the built-in `type()`:

```
type(miles)
```

What if you want to determine if miles is also an instance of the Dog class? You can do this with the built-in `isinstance()`:

```python
print(isinstance(miles, Dog))
```

Notice that `isinstance()` takes two (2) arguments, an object and a class. In the example above, isinstance() checks if miles is an instance of the Dog class and returns True.

```text
True
```

The miles, buddy, jack, and jim objects are all Dog instances, but miles isn't a Bulldog instance, and jack isn't a Dachshund instance:

```python
print(isinstance(miles, Bulldog))
print(isinstance(jack, Dachshund))
```

More generally, all objects created from a child class are instances of the parent class, although they may not be instances of other child classes.

Now that you've created child classes for some different breeds of dogs, we can give each breed its own sound.

Example **Creating a Child Class** 13

Create a child class Bus that will inherit all of the variables and methods of the Vehicle class. Use the following code as example.

```python
class Vehicle:

    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage
```

Create a Bus object that will inherit all of the variables and methods of the parent Vehicle class and display it.

```
Vehicle Name: School Volvo Speed: 180 Mileage: 12
```

Solution **Creating a Child Class**

```python
class Vehicle:

    def __init__(self, name, max_speed, mileage):
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage

class Bus(Vehicle):
    pass

School_bus = Bus("School Volvo", 180, 12)
print("Vehicle Name:", School_bus.name, "Speed:", School_bus.max_speed, "Mileage:",
↪  School_bus.mileage)
```

3.9.1 Extending Parent Class Functionality

As different breeds of dogs have slightly different barks, we want to provide a default value for the sound argument of their respective `.speak()` methods. To do this, you want to override `.speak()` in the class definition for each breed.

To override a method defined on the parent class, we define a method with the same name on the child class. Here's what that looks like for the JackRussellTerrier class:

```python
class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return f"{self.name} says {sound}"
```

Now `.speak()` is defined on the JackRussellTerrier class with the default argument for sound set to `"Arf"`. Let's update our previous code with the new `JackRussellTerrier` class. You can now call `.speak()` on a `JackRussellTerrier` instance without passing an argument to sound:

```python
miles = JackRussellTerrier("Miles", 4)
miles.speak()
```

Sometimes dogs make **different** noises, so if Miles gets angry and growls, you can still call .speak() with a different sound:

```
miles.speak("Grrr")
```

One thing to keep in mind about class inheritance is that changes to the parent class

automatically propagate to child classes. This occurs as long as the attribute or method being changed isn't overridden in the child class.
For example, in the editor window, change the string returned by `.speak()` in the Dog class

```
# dog.py

class Dog:
    # ...

    def speak(self, sound):
        return f"{self.name} barks: {sound}"

# ...
```

Save the file and press F5. Now, when you create a new Bulldog instance named jim, jim.speak() returns the new string:

```
jim = Bulldog("Jim", 5)
jim.speak("Woof")
```

However, calling `.speak()` on a JackRussellTerrier instance won't show the new style of output:

```
miles = JackRussellTerrier("Miles", 4)
miles.speak()
```

Sometimes it makes sense to completely override a method from a parent class. But in this case, you don't want the JackRussellTerrier class to lose any changes that you might make to the formatting of the Dog.speak() output string.
To do this, you still need to define a `.speak()` method on the child JackRussellTerrier class. But instead of explicitly defining the output string, you need to call the Dog class's .speak() from inside the child class's `.speak()` using the same arguments that you passed to JackRussellTerrier.speak().
You can access the parent class from inside a method of a child class by using `super()`:

```
# dog.py

# ...

class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return super().speak(sound)
```

```
# ...
```

When you call super().speak(sound) inside JackRussellTerrier, Python searches the parent class, Dog, for a `.speak()` method and calls it with the variable sound. Update dog.py with the new JackRussellTerrier class. Save the file and press F5 so you can test it in the interactive window:

```
miles = JackRussellTerrier("Miles", 4)
miles.speak()
```

Now when you call miles.speak(), you'll see output reflecting the new formatting in the Dog class.

> In the above examples, the class hierarchy is very straightforward. The JackRussellTerrier class has a single parent class, Dog. In real-world examples, the class hierarchy can get quite complicated.

3.10 The classmethod() function

The classmethod() is an inbuilt function in Python, which returns a class method for a given function. This means that `classmethod()` is a built-in Python function that transforms a regular method into a class method.

When a method is defined using the `@classmethod` decorator (which internally calls classmethod()), the method is **bound to the class and not to an instance of the class**. As a result, the method receives the class (`cls`) as its first argument, rather than an instance (`self`).

3.10.1 Class v. Static Method

There are some differences between a class method and a static method which is worth mention:

- A class method takes class as the first parameter while a static method needs no specific parameters.
- A class method can access or modify the class state while a static method can't access or modify it.
- In general, static methods know nothing about the class state. They are utility-type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as a parameter.

- We use @classmethod decorator in Python to create a class method and we use @staticmethod decorator to create a static method in Python.

Example **An Example of @classmethod** 14

Create a simple classmethod

In this example, we are going to see how to create a class method in Python. For this, we created a class named "Geeks" with a member variable "course" and created a function named "purchase" which prints the object. Now, we passed the method Geeks.purchase into a class method using the @classmethod decorator, which converts the method to a class method. With the class method in place, we can call the function "purchase" without creating a function object, directly using the class name "Geeks."

```python
        self.name = name
        self.max_speed = max_speed
        self.mileage = mileage

    def seating_capacity(self, capacity):
        return f"The seating capacity of a {self.name} is {capacity} passengers"

class Bus(Vehicle):
    # assign default value to capacity
    def seating_capacity(self, capacity=50):
        return super().seating_capacity(capacity=50)

School_bus = Bus("School Volvo", 180, 12)
print(School_bus.seating_capacity())

#+end_src

** Practicals

#+NAME: PR-1
#+begin_src python :session :results output
class Geeks:
    course = 'DSA'
    list_of_instances = []

    def __init__(self, name):
        self.name = name
        Geeks.list_of_instances.append(self)

    @classmethod
```

Solution **An Example of @classmethod**

Part II.

Python For Scientific Applications