Python Notebook from Zero to Advanced

None

Zhenyuan Lu

Table of contents

1. Python Notebook from Zero to Advanced

Author: Zhenyuan Lu

Version: 0.9.0

Created: 12/05/2022, *last modified on 06/20/2023*

Welcome to the Python Notebook from Zero to Advanced, an simplistic place on the internet to learn Python (besides the best place official Python documentation, of course 😉)

1.0.1 About This Book

This book is designed for those with little or no Python programming experience, and it is filled with concise, easy-to-understand examples that will help you learn quickly and effectively.

Throughout this comprehensive guide, we'll cover a wide range of topics, including data types, control structures, functions, and more.

2. Disclaimer

I am pretty sure there are some typing errors, spelling mistakes, and other inaccuracies. If you find any such issues, please do not hesitate to contact me via lu [dot] zhenyua [at] northeastern [dot] edu.

This tutorial is aimed to those who have zero or less Python programming experience with concise and simple examples throughout the entire tutorial. The content has been inspired by official Python documentation, Corey Schafer's tutorial. If you believe any content has been used inappropriately, please let me know, and I will address the issue.

This work is licensed under the **MIT License**.

3. Get Started

Welcome! Python is a versatile and powerful programming language, widely used for web development, data analysis, artificial intelligence, and more. This tutorial will guide you through the installation of Python and setting up a Python development environment in PyCharm, a popular integrated development environment (IDE) for Python.

To get started with Python, you'll first need to install it on your computer. Follow the instructions below for your operating system:

3.1 1.1. Installation with Anaconda

Anaconda is a popular distribution of Python and R programming languages, which simplifies package management and deployment. It comes with many pre-installed packages and tools for data science and machine learning.

For Windows:

Visit the Anaconda website at https://www.anaconda.com/products/distribution and download the installer for your operating system. Run the installer and follow the installation instructions. After the installation is complete, you can verify the installation by opening a terminal (or Anaconda Prompt on Windows) and typing conda --version. You should see the installed Anaconda version displayed. To check the Python version, type `python --version`.

For macOS:

Visit the Anaconda website at https://www.anaconda.com/products/distribution and download the installer for macOS. Open the downloaded package (.pkg file) and follow the installation instructions. After the installation is complete, you can verify the installation by opening Terminal and typing conda --version. You should see the installed Anaconda version displayed. To check the Python version, type `python --version`.

3.2 1.2. Standard Installation

For Windows:

Visit the official Python website at https://www.python.org/downloads/ and download the latest version of Python. Run the installer. Be sure to check the box "Add Python to PATH" before clicking "Install Now." This will make it easier to run Python from the command prompt. After the installation is complete, you can verify the installation by opening a command prompt and typing `python --version`. You should see the installed Python version displayed.

For macOS:

Visit the official Python website at https://www.python.org/downloads/ and download the latest version of Python. Open the downloaded package and follow the installation instructions. After the installation is complete, you can verify the installation by opening Terminal and typing `python --version`. You should see the installed Python version displayed.

3.3 1.2. Setting up a Python Development Environment in PyCharm

Now that Python is installed, let's set up a development environment in PyCharm.

1. Download and install PyCharm from https://www.jetbrains.com/pycharm/download/. There are two editions available: Community Edition (free) and Professional Edition (paid). For this tutorial, the Community Edition is sufficient.
2. Open PyCharm and create a new project by clicking "Create New Project" on the welcome screen.
3. Choose a location for your project and make sure the "Python Interpreter" field is set to the Python version you installed earlier. If not, click the gear icon next to the field and select "Add Interpreter." Choose "System Interpreter" and select the Python executable from the list. Click "Create" to create your new Python project.
4. You're now ready to start writing Python code! In the next chapter, we'll dive into Python basics, including syntax, variables, and data types.

4. 2. Data Types

4.1 2.1. Numbers

In this section, we will cover the different number types in Python, such as integers and floating-point numbers, and how to work with them.

4.1.1 2.1.1. Integers and Floats

Python has two primary numeric types: integers (int) and floating-point numbers (float).

Assign an integer `5` to a variable named `num`. The `print()` function is then used to output the value of `num` to the console.

Input

```
# Integers
num = 5
print(num)
```

Output

```
5
```

Assign a floating-point number `5.2` to a variable named `num`. The `print()` function is then used to output the value of `num` to the console.

Input

```
# Floats
num = 5.2
print(num)
```

Output

```
5.2
```

4.1.2 2.1.2. `type()` and `__class__`

Info

`type()` is a built-in function that returns the type of an object. It is the same as calling the object's `__class__` attribute, e.g. `object.__class__`, but which is less commonly used.

Use `type()` to check the type of a variable.

Input

```
num = 5
print(type(num))
```

Output

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

The num variable is a floating-point number, so the `type()` function returns `<class 'float'>`.

Input

```
# Floats
num = 5.2
print(type(num))
```

Output

```
<class 'float'>
```

Floats with scientific notation.

Input

```
num = 5.2e3
print(type(num))
```

Output

```
<class 'float'>
```

Use `__class__` to check the type of a variable.

Input

```
print(num.__class__)
```

Output

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

4.1.3 2.1.3. Math Functions

Python supports various mathematical operations that can be performed on numbers, such as addition, subtraction, multiplication, division, and more. Here's a list of common mathematical operations and their corresponding symbols:

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Floor Division: `//`
- Exponentiation: `**`
- Modulus: `%`

Let's see some examples of using these mathematical operations:

Addition Subtraction Multiplication Division Floor Division Exponentiation Modulus

Input

```
num = 5
print(num + 2)
```

Output

```
7
```

Input

```
num = 5
print(num - 2)
```

Output

```
3
```

Input

```
num = 5
print(num * 2)
```

Output

```
10
```

Input

```
num = 5
print(num / 2)
```

Output

```
2.5
```

Input

```
num = 5
print(num // 2)
```

Output

```
2
```

Input

```
num = 5
print(num ** 2)
```

Output

```
25
```

Input

```
num = 5
print(num % 2)
```

Output

```
1
```

These operations can be used in combination, following the standard order of operations (PEMDAS), to perform more complex calculations. Parentheses can be used to specify the order of operations explicitly.

`abs()` and `round()`

`abs()` and `round()` are two built-in functions that can be used to perform mathematical operations on numbers.

The `abs()` function returns the absolute value of a number.

Input

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

Output

```
5
```

The `round()` function rounds a number to a specified number of decimal places.

Input

```
# round() function
print(round(5.75))
```

Output

```
6
```

`round()` with 2nd argument to specify the number of decimal places. Here we round `5.75` to 1 decimal place.

Input

```
# round() with 2nd argument
print(round(5.75, 1))
```

Output

```
5.8
```

4.1.4 2.1.4. Increment and Decrement

In Python, you can increment or decrement the value of a variable using the `+=` and `-=` operators, respectively. These operators are shorthand for adding or subtracting a value to the variable and then assigning the result back to the variable.

We use `num = num + 1` to increment the value of `num` by 1.

Input

```
# Increment
num = 5
num = num + 1
print(num)
```

Output

```
6
```

Increment using shorthand `+=` operator. The `+=` operator is equivalent to `num = num + 1`.

Input

```
# Increment using shorthand +=
num = 5
num += 1
print(num)
```

Output

```
6
```

The `num = num - 1` is used to decrement the value of `num` by 1.

Input

```
# Decrement
num = 5
num = num - 1
print(num)
```

Output

```
4
```

Decrement using shorthand `-=` operator. The `-=` operator is equivalent to `num = num - 1`.

Input

```
# Decrement using shorthand -=
num = 5
num -= 1
print(num)
```

Output

```
4
```

Using the `+=` and `-=` operators can make your code shorter and more readable, especially when performing multiple increment or decrement operations on the same variable.

4.1.5 2.1.5. Comparison Operators

- Equal: `==`
- Not Equal: `!=`
- Greater Than: `>`
- Less Than: `<`
- Greater or Equal: `>=`
- Less or Equal: `<=`

Setting

```
num_1 = 5
num_2 = 2
```

Equal: == **Not Equal: !=** **Greater Than: >** **Less Than: <** **Greater or Equal: >=** **Less or Equal: <=**

Input

```
print(num_1 == num_2)
```

Output

```
False
```

Input

```
print(num_1 != num_2)
```

Output

```
True
```

Input

```
print(num_1 > num_2)
```

Output

```
True
```

Input

```
print(num_1 < num_2)
```

Output

```
False
```

Input

```
print(num_1 >= num_2)
```

Output

```
True
```

Input

```
print(num_1 <= num_2)
```

Output

```
False
```

4.1.6 2.1.6. Casting

Casting is the process of converting a value from one data type to another. In Python, casting is achieved using built-in functions like `int()`, `float()`, and `complex()`.

For example, when working with numbers, you might need to convert a string to an integer or a float. This is useful when you want to perform mathematical operations on string representations of numbers.

`int()`: Convert a value to an integer `float()`: Convert a value to a float `complex()`: Convert a value to a complex number

Check the type of variable `num_1`.

Input

```
num_1 = '5'
print(type(num_1))
```

Output

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

If we have two numbers as strings, when we `+` them, they are concatenated instead of added.

Input

```
num_1 = '5'
num_2 = '2'
print(num_1 + num_2)
```

Output

```
52
```

If we want to add them, we need to convert them to integers first.

Input

```
# Convert string to int
num_1 = int(num_1)
num_2 = int(num_2)
print(num_1 + num_2)
```

Output

```
7
```

If we convert a floating number to an integer, the decimal part will be removed.

Input

```
num = 5.2
print(int(num))
```

Output

```
5
```

If we convert an integer to a float, the result will be a float with `.0` at the end.

Input

```
# float()
num = 5
print(float(num))
```

Output

```
5.0
```

If we convert an integer to a complex number, the result will be a complex number with `j` at the end.

Input

```
# complex()
num = 5
print(complex(num))
```

Output

```
(5+0j)
```

`zfill()` method adds zeros (0) at the beginning of the string, until it reaches the specified length.

Input

```
# zfill method
num = 5
print(str(num).zfill(3))
```

Output

```
005
```

4.2 2.2. Strings

Strings are one of the most important and commonly used data types in Python. A string is simply a sequence of characters, such as letters, numbers, and symbols. In Python, strings are created using either single quotes `'<str>'` or double quotes `"<str>"`. This tutorial will cover the basics of string manipulation in Python, including string indexing, slicing, concatenation, formatting, and various string methods.

A string variable named sentence is defined and assigned the value `"Hello World"`. The `print()` function is then used to output the value of sentence to the console. This code demonstrates how to create and output a basic string in Python.

Input

```
sentence = 'Hello World'
print(sentence)
```

Output

```
Hello World
```

4.2.1 2.2.1. String Basics

Quotes

Single quotes are faster, more readable, and more commonly used than double quotes in Python. However, if a string itself contains a single quote, then it must be escaped using a backslash \ so that Python can properly interpret the string.

In the example code, a new string variable named sentence is defined using single quotes and contains the word `"Tub's World"`. To escape the single quote in the middle of the string, a backslash is used before it. The `print()` function is then used to output the value of sentence to the console.

Input

```
# Single quote
# Use backslash to escape single quote
sentence = 'Tub\'s World'
print(sentence)
```

Output

```
Tub's World
```

The following is the same example as above, but using double quotes instead of single quotes.

Input

```
# Use double quote
sentence = "Tub's World"
print(sentence)
```

Output

```
Tub's World
```

Triple quotes are used to create multi-line strings. In the example code, a new string variable named sentence is defined using triple quotes and contains the string `"Tub's World"` and `"is a good place to live in"`. The `print()` function is then used to output the value of sentence to the console.

Input

```
# Three double quotes for multi-line string
sentence = """Tub's World
is a good place to live in"""
print(sentence)
```

Output

```
Tub's World
is a good place to live in
```

Length of a String

The following code demonstrates how to use the `len()` function to get the length of a string.

Input

```
# Use len() to get the length of a string
sentence = "Tub's World"
print(len(sentence))
```

Output

```
12
```

Upper and Lower Case

The `lower()` method converts all the characters in a string to lowercase. This is useful for case-insensitive comparisons or normalization of text data.

Input

```
# Lowercase (all letters)
sentence = "Tub's World"
print(sentence.lower())
```

Output

```
tub's world
```

The `upper()` method converts all the characters in a string to uppercase.

Input

```
# Uppercase (all letters)
sentence = "Tub's World"
print(sentence.upper())
```

Output

```
TUB'S WORLD
```

Input

```
# Capitalize
sentence = "Tub's World"
print(sentence.capitalize())
```

Output

```
Tub's world
```

The `capitalize()` method capitalizes the first character of a string and makes the rest of the characters lowercase.

4.2.2 2.2.2. String Indexing

Indexing allows us to access individual characters within a string using their position, also known as the index. It is essential to understand that in Python, indexing starts from 0, meaning the first character in the string has an index of 0, the second character has an index of 1, and so on.

Here, we define a string sentence containing the text `"Tub's World"`. We then use indexing to access the character at position 0, which is `'T'`. The `print()` function displays the output, confirming that the first character in the string is indeed `'T'`.

Input

```
# String indexing
# Indexing starts from 0
sentence = "Tub's World"
print(sentence[0])
```

Output

```
T
```

Now, let's see how negative indexing works in Python:

Input

```
# If index is negative, it starts from the end
sentence = "Tub's World"
print(sentence[-1])
```

Output

```
d
```

Accessing an index that is out of range will result in an error.

Input

```
# If we input 11, it will throw an error
sentence = "Tub's World"
print(sentence[11])
```

Output

```
IndexError: string index out of range
```

4.2.3 2.2.3. String Slicing

String slicing is a technique used to extract a subset of characters from a string. Slicing is done by specifying the starting and ending indices of the slice, separated by a colon. The starting index is inclusive, and the ending index is exclusive. If either the starting or ending index is omitted, it defaults to the beginning or end of the string, respectively.

Basic Indexing

Input

```
# The first index is inclusive, the second index is exclusive
sentence = "Tub's World"
# Index: 012345678910
# Reverse index: 9876543210
```

Basic Syntax

```
# string[start:end:step]
sentence = "Tub's World"
# sentence[index:index:step]
```

Input

```
# The first index is inclusive, the second index is exclusive
sentence = "Tub's World"
# Index: 012345678910
print(sentence[0:3])
```

Output

```
Tub
```

Input

```
# We can also omit the first index
sentence = "Tub's World"
print(sentence[:3])
```

Output

```
Tub
```

Input

```
# We can also omit the second index
sentence = "Tub's World"
print(sentence[3:])
```

Output

```
's World
```

Input

```
# We can also omit both index
sentence = "Tub's World"
print(sentence[:])
```

Output

```
Tub's World
```

Reverse Slicing

If we want to print out the `World`, we can use reverse indexing to get the last 5 characters:

Input

```
sentence = "Tub's World"
# Reverse index: 9876543210
print(sentence[-6:-1])
```

Output

```
World
```

Step Size

We can also specify a step size to skip characters in the string. The following code demonstrates how to use a step size of 2 to print out every other character in the string:

Input

```
# Step size
sentence = "Tub's World"
print(sentence[::2])
```

Output

```
Tb sWrd
```

We can also specify a step size of 2 to print out every other character in the string, starting from the second character to the :

Input

```
# Step size
sentence = "Tub's World"
print(sentence[1:6:2])
```

Output

```
u'
```

Negative Step Size

We can also use a negative step size to reverse the string:

Input

```
# Negative step size
sentence = "Tub's World"
print(sentence[::-1])
```

Output

```
dlroW s'buT
```

We can also print out the same result by using the following code:

Input

```
sentence = "Tub's World"
print(sentence[-1:2:-1])
```

Output

```
dlroW s'buT
```

4.2.4 2.2.4. `count()`, `find()`, and `replace()`

The `count()`, `find()`, and `replace()` methods are used to count, find, and replace substrings within a string, respectively.

`count()` returns the number of occurrences of a specified substring in the given string.

Input

```
# Count (return the number of occurrences)
sentence = "Tub's World"
print(sentence.count('o'))
```

Output

```
1
```

`find()` returns the index of the first occurrence of a specified substring in the given string.

Input

```
# Find (return the index of the first occurrence)
sentence = "Tub's World"
print(sentence.find('o'))
```

Output

```
8
```

`replace()` replaces all occurrences of a specified substring (old) with a new substring in the given string.

Input

```
# Replace (replace old with new)
sentence = "Tub's World"
print(sentence.replace('Tub', 'Tom'))
```

Output

```
Tom's World
```

We can also assign the result of the replace() method back to the same variable if we want to update the original string:

Input

```
# We can also assign the result to the same variable
sentence = "Tub's World"
sentence = sentence.replace('Tub', 'Tom')
print(sentence)
```

Output

```
Tom's World
```

4.2.5 2.2.5. String Concatenation

String concatenation is a technique used to join two or more strings together. In Python, string concatenation is done using the `+` operator.

Input

```
# String concatenation
name = 'Tub'
age = 5
sentence = name + 'is' + str(age) + 'years old'
print(sentence)
```

Output

```
Tub is 5 years old
```

String concatenation is essential when working with dynamic content, such as user input or data from external sources, as it enables you to construct meaningful and contextually relevant strings. When concatenating strings, it's essential to pay attention to spaces and punctuation to ensure the resulting string is formatted correctly. For instance, in this example, we've added a space character between the punctuation and noun variables to separate the words in the final string.

4.2.6 2.2.6. String Formatting

String formatting is a technique used to embed values within a string. In Python, there are several ways to format strings, including using the `format()` method and using f-strings. The `format()` method allows you to embed values within a string using placeholders, which are represented by curly braces `{}`. You can also use named placeholders to improve the readability of your code. F-strings are a more recent addition to Python and provide a more concise and intuitive way to embed values within a string.

Here, we have three string variables: `name`, and an integer variable `age`. We introduce three methods for including non-string variables in a string using string formatting:

Input: Setting up variables

```
# If we want to concatenate a lot of strings,
# it is better to use string formatting
name = 'Tub'
age = 5
```

Method 1: Using the `format()` method

Input

```
# Method 1
# Use format() method
sentence = '{} is {} years old'.format(name, age)
print(sentence)
```

Output

```
Tub is 5 years old
```

Method 2: Using the `format()` method with keyword arguments

Input

```
# Method 2
# use format() method with keyword arguments
sentence = '{n} is {a} years old'.format(n=name, a=age)
print(sentence)
```

Output

```
Tub is 5 years old
```

Method 3: Using `f` -strings (Python 3.6+)

Input

```
# Method 3
# Use f-string (3.6+)
sentence = f'{name} is {age} years old'
print(sentence)
```

Output

```
Tub is 5 years old
```

In addition to the methods above, we can also modify the string content within the string formatting. In this case, we convert the name variable to uppercase using the `.upper()` method:

Input

```
# With upper case
sentence = f'{name.upper()} is {age} years old'
print(sentence)
```

Output

```
TUB is 5 years old
```

All three methods allow you to easily include non-string variables in your string, without the need for explicit type conversion, making them more efficient and readable than traditional string concatenation.

4.2.7 2.2.7. `dir()` and `help()`

The `dir()` function returns a list of all attributes and methods of the specified object, while the `help()` function provides documentation on a specific attribute or method.

We pass the name variable, which is a string, to the dir() function. This will return a list of all available attributes and methods for the string object.

Input

```
# dir() function returns a list of all
# attributes and methods of the specified object
name = 'Tub'
print(dir(name))
```

Output

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getstate__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

We can also use `help()` function to display information about the string variable, but instead of passing the variable name, we pass the string object `str` itself.

Input

```
# help() function
name = 'Tub'
print(help(str))
```

Output (partial)

```
Help on class str in module builtins:...
...
```

4.3 2.3. Lists

Lists in Python are ordered, mutable collections of items. They can store elements of different types, such as strings, integers, or other objects.

Creating a list with pet names:

Input

```
# A list of pets names
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(pet)
```

Output

```
['Tub', 'Furrytail', 'Cat', 'Barkalot']
```

Create an empty list using the `[]` or `list()` function:

Input

```
# Create empty list
empty_list = []
# or
empty_list = list()
print(empty_list)
```

Output

```
[]
```

4.3.1 2.3.1. List Indexing

Check the length of the list using the `len()` function:

Input

```
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(len(pet))
```

Output

```
4
```

In Python the first element of a list has index `0`, which is different from other programming languages, e.g. R, where the first element has index `1`.

Access the first element of the list using the index `0`:

Input

```
# Indexing starts from 0
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(pet[0])
```

Output

```
Tub
```

Access the last element of the list using the index `-1` :

Input

```python
# If index is negative, it starts from the end
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(pet[-1])
```

Output

```
Barkalot
```

Access the second element of the list using the index `1` :

Input

```python
```python title="Input"
# If we input 4, it will throw an error
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(pet[4])
```

Output

```
IndexError: list index out of range
```

4.3.2 2.3.2. List Slicing

Slicing is a way to access a subset of a list. We can use the colon `:` to specify the start and end index of the slice. The slice will include the start index, but not the end index.

Basic Indexing for List Slicing

Basic Indexing

```python
# Index: 0  1  2  3
# Reverse index : -4 -3 -2 -1
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
```

Basic Syntax

```python
# list[start:end:step]
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
# pet[index:index:step]
```

Slice the first two elements of the list:

Input

```python
# Slicing starts from 0
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(pet[0:2])
```

Output

```
['Tub', 'Furrytail']
```

If we omit the start index, the slice will start from the beginning of the list:

Input

```python
# Omit the first index
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(pet[:2])
```

Output

```
['Tub', 'Furrytail']
```
```

If we omit the end index, the slice will end at the end of the list:

Input

```
# Omit the second index
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(pet[2:])
```

Output

```
['Cat', 'Barkalot']
```

We can also omit both indices to return the entire list:

Input

```
# Omit both index
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(pet[:])
```

Output

```
['Tub', 'Furrytail', 'Cat', 'Barkalot']
```

Reverse Slicing Recall that the index `-1` refers to the last element of the list. We can use this to reverse the list:

Input

```
# Reverse Index: -5, -4, -3, -2, -1
numbers_list = [1, 2, 3, 4, 5]
print(numbers_list[-3:-1])
```

Output

```
[3, 4]
```

If we want to all the way from the last element to the `-3` index (not including the `-3` index), we can omit the last index:

Input

```
print(numbers_list[-3:])
```

Output

```
[3, 4, 5]
```

Or we can omit the first index to all the way from the first element to the `-3` index (not including the `-3` index):

Input

```
print(numbers_list[:-3])
```

Output

```
[1, 2]
```

We can also use positive step size to reverse the list:

Input

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

Output

```
[2, 3]
```

Step Size

We can also specify the step size of the slice. For example, we can slice every other element of the list by specifying the step size as `2`:

Input

```
```python title="Input"
numbers_list = [1, 2, 3, 4, 5]
# list[start:end:step]
print(numbers_list[0:3:2])
```

Output

```
[1, 3]
```

The default step size is `1`. We can omit the step size if we want to slice every element of the list:

Omit the step size:

Input

```
print(numbers_list[0:3])
```

Output

```
[1, 2, 3]
```

With step size `1`:

Input

```
print(numbers_list[0:3:1])
```

Output

```
[1, 2, 3]
```

We can also use negative step size to reverse the list:

Input

```
print(numbers_list[0:3:-1])
```

Output

```
[]
```

However, this will return an empty list. Since the step size is negative number `-1`, the slice will start from the `0` index and then move backward to the `-1` index, which is on the opposite direction of index `3`. Therefore, it returns an empty list.

To fix this, we can reverse the start and end index:

Input

```
print(numbers_list[-3::-1])
```

Output

```
[3, 2, 1]
```

We can also omit the start and end index to include the entire list, by a step of `-2`:

Input

```
print(numbers_list[::-2])
```

Output

```
[5, 3, 1]
```
```

4.3.3 2.3.2. List Methods

There are many methods that can be used with lists. In the following examples, we will introduce couple of common methods to manipulate the list.

Add an item to the end of the list using the `append()` method:

Input

```
# Add an item to the end of the list
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
pet.append('Hootsworth ')
print(pet)
```

Output

```
['Tub', 'Furrytail', 'Cat', 'Barkalot', 'Hootsworth ']
```

Add an item to a specific index, e.g. `2`, using the `insert()` method:

Input

```
# Add an item to a specific index
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
pet.insert(2, 'Fish')
print(pet)
```

Output

```
['Tub', 'Furrytail', 'Fish', 'Cat', 'Barkalot']
```

Now, we want to insert a list into a list after a specific location, e.g. 0, using the `insert()` method:

Input

```
# Insert a list into a list
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
pet_2 = ['Bumblefluff ', 'Whiskerfloof']
pet.insert(0, pet_2)
print(pet)
```

Output

```
[['Bumblefluff ', 'Whiskerfloof'], 'Tub', 'Furrytail', 'Cat', 'Barkalot']
```

> **`insert()` method inserts the list as a single element**
>
> However, this is not what we want because we want to insert the elements of the list `pet_2` into the list `pet`.

Then we can use the `extend()` instead to do this:

Input

```
# Extend a list
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
pet_2 = ['Bumblefluff ', 'Whiskerfloof']
pet.extend(pet_2)
print(pet)
```

Output

```
['Tub', 'Furrytail', 'Cat', 'Barkalot', 'Bumblefluff ', 'Whiskerfloof']
```

Now, we have successfully inserted the elements of the list `pet_2` into the list `pet`.

We can remove an item from the list using the `remove()` method:

Input

```
# Remove an item from the list
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
pet.remove('Tub')
print(pet)
```

Output

```
['Furrytail', 'Cat', 'Barkalot']
```

We can also remove an item from the list using the `pop()` method. If we do not specify the index, it will remove the last item from the list:

Input

```
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
pet.pop()
print(pet)
```

Output

```
['Tub', 'Furrytail', 'Cat']
```

Or we can specify the index to remove the item at that index:

Input

```
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
pet.pop(1)
print(pet)
```

Output

```
['Tub', 'Cat', 'Barkalot']
```

We can also use the `pop()` method to get the item that we removed from the list, and assign it to the `popped_sp` variable:

Input

```
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
popped_pet = pet.pop()
print(popped_pet)
```

Output

```
Barkalot
```

This is very helpful when we have a queue to keep popping until the queue is empty and we want to keep track of the items that we have popped.

If we want to search for an index of an item in the list, we can use the `index()` method:

Input

```
# Search for an index of an item in the list
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(pet.index('Tub'))
```

Output

```
0
```

We can check if the `'Tub'` is in the pet list using the `in` keyword:

Input

```
# If an item is in the list
print('Tub' in pet)
```

Output

```
True
```

Sometimes, we want to join a list of strings into a single string. We can use the `join()` method to do this:

Input

```
# Join a list of strings
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
pet_str = ','.join(pet)
print(pet_str)
```

Output

```
Tub,Furrytail,Cat,Barkalot
```

Or we can split the single string into a list by a specific character, e.g. `,` :

Input

```
# Split a string into a list by ','
pet_str = 'Tub,Furrytail,Cat,Barkalot'
pet_list = pet_str.split(',')
print(pet_list)
```

Output

```
['Tub', 'Furrytail', 'Cat', 'Barkalot']
```

When dealing with a list of numbers, we can use the `min()`, `max()`, and `sum()` functions to get the minimum, maximum, and sum of the numbers in the list:

`min()` **`max()`** **`sum()`**

Input

```
nums = [5, 3, 2, 4, 1]
print(min(nums))
```

Output

```
1
```

Input

```
nums = [5, 3, 2, 4, 1]
print(max(nums))
```

Output

```
5
```

Input

```
nums = [5, 3, 2, 4, 1]
print(sum(nums))
```

Output

```
15
```

4.3.4 2.3.3. Sorting and Reversing

We can reverse a list of strings using the `reverse()` method in reverse alphabetical order:

Input

```
# Reverse a list
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
```

```
pet.reverse()
print(pet)
```

Output

```
['Barkalot', 'Cat', 'Furrytail', 'Tub']
```

We can sort a list of strings using the `sort()` method in alphabetical order:

Input

```
# Sort a list
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
pet.sort()
print(pet)
```

Output

```
['Barkalot', 'Cat', 'Furrytail', 'Tub']
```

We can sort a list of numbers using the `sort()` method in alphabetical order:

Input

```
nums = [5, 3, 2, 4, 1]
nums.sort()
print(nums)
```

Output

```
[1, 2, 3, 4, 5]
```

Of course, we can also sort a list of numbers in reverse order by using `sort(reverse = True)` :

Input

```
# Instead of using .reverse(), we can use reverse = True
nums.sort(reverse = True)
print(nums)
```

Output

```
[5, 4, 3, 2, 1]
```

However, the above methods `sort()` and `reverse()` are changing our original variables. What if we want to keep the original variables? We can use the `sorted()` function to sort a list of strings in alphabetical order:

Input

```
pet = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
sorted_pet = sorted(pet)
print(sorted_pet)
print(pet)
```

Output

```
['Barkalot', 'Cat', 'Furrytail', 'Tub'] # sorted_pet
['Tub', 'Furrytail', 'Cat', 'Barkalot'] # original pet
```

Here, we don't change the original variable `pet` but create a new variable `sorted_pet` to store the sorted list. This is useful when we want to keep the original list unchanged.

4.4 2.4. Tuples

Tuples are similar to lists, but they are immutable. This means that we cannot change the contents of a tuple once it is created. Tuples are useful when we want to store a list of items that cannot be changed.

4.4.1 2.4.1. Creating a Tuple

Create an empty tuple is similar to creating an empty list, the only difference is that we use `()` instead of `[]`, or you can use the `tuple()` function:

Input

```
# Create empty tuple
empty_tuple = ()
# or
empty_tuple = tuple()
print(empty_tuple)
```

Output

```
()
```

Create a tuple based on the same strings as the list `pet`:

Input

```
pet_tup_1 = ('Tub', 'Furrytail', 'Cat', 'Barkalot')
print(pet_tup_1)
```

Output

```
('Tub', 'Furrytail', 'Cat', 'Barkalot')
```

4.5 2.5. Immutable vs. Mutable

Immutable means that we cannot change the contents of the object. Mutable means that we can change the contents of the object.

Pros and Cons of Immutable

Immutable Data Types	Mutable Data Types

- Numbers
- Strings
- Tuples
- Frozen sets

- Lists
- Dictionaries
- Sets

Here's a general overview of the advantages and disadvantages of mutable and immutable objects:

Pros and Cons of Immutable

Pros of Immutable Cons of Immutable

Note

Simplicity: Immutability makes the code easier to reason about, as you don't have to worry about unintentional changes to the object.

Hashable: Immutable objects can be used as keys in dictionaries, as their content remains constant and their hash values do not change over time.

Optimization: Immutable objects allow languages and compilers to perform certain optimizations that can improve the performance of a program.

Thread-safety: Immutable objects are inherently thread-safe, as multiple threads cannot modify them simultaneously. This property eliminates the need for locking mechanisms when working with immutable objects in multi-threaded environments.

Predictability: When you pass an immutable object to a function, you can be sure that the function will not modify the object, which ensures that the behavior of the program remains predictable.

Note

Memory overhead: Since each operation on an immutable object creates a new object, it can lead to increased memory usage, especially when manipulating large objects or performing many operations.

Performance: Creating new objects for each operation can be slower than modifying objects in-place, particularly in cases where the program performs many operations on objects. In such situations, using mutable data structures may be more efficient.

Pros and Cons of Mutable

Pros of Mutable Cons of Mutable

Note

In-place modification: Mutable objects can be modified in-place, which can lead to better performance and lower memory usage, especially when working with large objects or performing many operations on objects.

Flexibility: Mutable objects offer more flexibility in how you can manipulate and change data, which can be helpful in certain scenarios.

Note

Complexity: Mutable objects can make code harder to reason about, as you need to consider the possibility of unintentional changes to the object.

Thread-safety: Mutable objects are not inherently thread-safe, and using them in multi-threaded environments can lead to race conditions and other concurrency-related issues if proper locking mechanisms are not in place.

Predictability: When you pass a mutable object to a function, you cannot be sure whether the function will modify the object or not, which can make the behavior of the program less predictable.

Not hashable: Mutable objects cannot be used as keys in dictionaries, as their content can change, potentially causing issues with hashing.

4.5.1 2.5.1. String is Immutable

Strings in Python are immutable, meaning you cannot change their content directly. Instead, when you perform operations like concatenation, replacement, changing the case, or slicing, you create new strings rather than modifying the original ones.

Concatenation

Input

```
original_string = "Hello, "
pet = "Tub"

greeting = original_string + pet
print(greeting)
print('Address of original_string is: {}'.format(id(original_string)))
print('Address of greeting is: {}'.format(id(greeting)))
```

Output

```
Hello, Tub
Address of original_string is: 2186723937904
Address of greeting is: 2186728523312
```

As you can see, the memory addresses of the original string and the new string are different, confirming that a new string is created during concatenation.

Replace

The replace() method in Python is used to replace a specified value with another value in a string. When you use the `replace()` method, a new string is created, and the original string remains unchanged.

Input

```
original_string = "Hello, World!"

new_string = original_string.replace("World", "Tub")
print(new_string)
print('Address of original_string is:{}'.format(id(original_string)))
print('Address of new_string is:{}'.format(id(new_string)))
```

Output

```
Hello, Tub
Address of original_string is:1436882337200
Address of new_string is:1436885468848
```

Again, the memory addresses of the original string and the new string are different, confirming that a new string is created during the replacement.

Change the Case

Input

```
original_string = "Tub is awesome!"

uppercase_string = original_string.upper()
print(uppercase_string)
print('Address of original_string is:{}'.format(id(original_string)))
print('Address of uppercase_string is:{}'.format((id(uppercase_string))))
```

Output

```
TUB IS AWESOME!
Address of original_string is:1923265193776
Address of uppercase_string is:1923268233008
```

Again, the memory addresses of the original string and the new string are different.

Then you may ask what is the advantage of immutable:

Slicing When you slice a string, a new string is created, and the original string remains unchanged.

Input

```
original_string = "Tub is cute!"

substring = original_string[0:3]
print(substring)
```

```
print('Address of original_string is:{}'.format(id(original_string)))
print('Address of substring is:{}'.format(id(substring)))
```

Output

```
Tub
Address of original_string is:1923265193776
Address of substring is:1923268233008
```

4.5.2 2.5.1. List is Mutable

Input

```
pet_1 = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
pet_2 = pet_1
pet_1[1] = 'Furrytail 2'
print(pet_1)
print(pet_2)
```

Output

```
['Tub', 'Furrytail 2', 'Cat', 'Barkalot']
['Tub', 'Furrytail 2', 'Cat', 'Barkalot']
```

We can see that when we change the contents of `pet_1`, the contents of `pet_2` also change. This is because `pet_1` and `pet_2` are both references to the same list in memory.

Input

```
print('Address of pet_1 is: {}'.format(id(pet_1)))
print('Address of pet_2 is: {}'.format(id(pet_2)))
```

Output

```
Address of pet_1 is: 2250825683136
Address of pet_2 is: 2250825683136
```

The address of `pet_1` and `pet_2` are the same, which means they are both references to the same list in memory.

Python vs. R in Variable Assignment

Python and R handle variable assignment differently, particularly when it comes to mutable objects like lists in Python.

In R, when you assign one variable to another, it creates a copy of the original variable's data. This means that if you change one variable's contents, the other variable's contents remain unchanged. This behavior is known as "copy-on-write" and allows R to save memory by not duplicating data until it is necessary.

In Python, when you assign one variable to another, you are actually creating a reference to the original variable's data, rather than copying the data itself. This means that if you change the contents of one variable, the other variable's contents will also change because they both point to the same data in memory.

If you want to create a new list that is a copy of `pet_1` like that in R and not a reference to `pet_1`, you can use the `copy()` method:

Input

```
pet_1 = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
pet_2 = pet_1.copy()
pet_1[1] = 'Furrytail 2'
print(pet_1)
print(pet_2)
```

Output

```
['Tub', 'Furrytail 2', 'Cat', 'Barkalot']
['Tub', 'Furrytail', 'Cat', 'Barkalot']
```

We can see that when we change the contents of `pet_1`, the `pet_2` remain unchanged. This is because `pet_2` refers to a new list that is a copy of `pet_1`.

Input

```
print('Address of pet_1 is: {}'.format(id(pet_1)))
print('Address of pet_2 is: {}'.format(id(pet_2)))
```

Output

```
Address of pet_1 is: 1704413046016
Address of pet_2 is: 1704413120128
```

The address of `pet_1` and `pet_2` are different.

4.5.3 2.5.2. Tuple is Immutable

In the other hand, tuple is immutable, so we cannot change the contents of a tuple once it is created. For example, we cannot change the second item `Furrytail` in the tuple `pet_tup_1`:

Input

```
# Immutable
pet_tup_1 = ('Tub', 'Furrytail', 'Cat', 'Barkalot')
pet_tup_1[1] = 'Furrytail 2'
print(pet_tup_1)
```

Output

```
TypeError: 'tuple' object does not support item assignment
```

Here we get an error message `TypeError: 'tuple' object does not support item assignment` on changing the second item in the `pet_tup_1` tuple.

4.6 2.6. Dictionaries

Dictionaries are similar to lists, but instead of using an index to access an item, we use a key. Dictionaries are unordered, and we cannot sort them. Dictionaries are mutable, so we can change the contents of a dictionary once it is created.

4.6.1 2.6.1. Creating a Dictionary

Let's create a dictionary that stores one of our old friend `'Tub'`, its `age`, and favoriate `habitat`:

Input

```
pet = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
print(pet)
```

Output

```
{'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
```

4.6.2 2.6.2. Accessing Items in a Dictionary

We can access the items in a dictionary by using the key `'species'`, `'age'`, and `'habitat'`:

Input

```
print(pet['species'])
print(pet['age'])
print(pet['habitat'])
```

Output

```
Tub
5
['bathroom', 'kitchen']
```

Of course, we can use number as the key, but it is not recommended:

Input

```
pet = {1: 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
print(pet[1])
```

Output

```
Tub
```

What if we accidently acces a key that not exist in the dictionary?

Input

```
# Access a key that not exist
pet = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
print(pet['weight'])
```

Output

```
KeyError: 'weight'
```

We will get an error message `KeyError: 'weight'`.

But practically this is not ideal, sometimes we just want to check if the key is in the dictionary or not without showing error, but return a `flag`.

We can use `get()` method to do this:

Input

```
# Get method
pet = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
print(pet.get('age'))
print(pet.get('weight'))
```

Output

```
5
None
```

We can see that when we use `get()` method, if the key is in the dictionary, it will return the value of the key, e.g. `5`, but if the key is not in the dictionary, it will return `None`.

We can also specify a default value to return if the key is not in the dictionary instead of `None`:

Input

```
pet = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
print(pet.get('weight', 'Not there'))
```

Output

```
Not there
```

4.6.3 2.6.3. Changing Items in a Dictionary

We can update the value of a key, e.g. `'weight'`:

Input

```
pet = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
pet['weight'] = 2000
print(pet)
```

Output

```
{'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen'], 'weight': 2000}
```

We can also use `update()` to update the values from keys in a dictionary:

Input

```
pet = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
pet.update({'weight': 1000, 'name': 'Fluffy', 'age': 6})
print(pet)
```

Output

```
{'species': 'Tub', 'age': 6, 'habitat': ['bathroom', 'kitchen'], 'weight': 1000, 'name': 'Fluffy'}
```

While if we want to delete the key `'age'` and its value from the dictionary, we can use `del`:

Input

```
pet = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
del pet['age']
print(pet)
```

Output

```
{'species': 'Tub', 'habitat': ['bathroom', 'kitchen']}
```

Or use `pop()`:

Input

```
pet = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
pet.pop('age')
print(pet)
```

Output

```
{'species': 'Tub', 'habitat': ['bathroom', 'kitchen']}
```

We can also assign the popped value to a variable:

Input

```
popped_age = pet.pop('age')
print(pet)
print(popped_age)
```

Output

```
{'species': 'Tub', 'habitat': ['bathroom', 'kitchen']}
5
```

This can be useful if we want to use the popped value later.

As we mentioned in the previous sections, `len()` can also be used to check how many keys in a dictionary:

Input

```
pet = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
print(len(pet))
```

Output

```
3
```

4.6.4 2.6.4. Accessing Keys and Values

We can access all the keys from a dictionary using `keys()`:

Input

```
pet = {'species': 'Tub', 'age': 5, 'habitat': ['bathroom', 'kitchen']}
print(pet.keys())
```

Output

```
dict_keys(['species', 'age', 'habitat'])
```

We can also access all the values from a dictionary using `values()` :

Input

```
print(pet.values())
```

Output

```
dict_values(['Tub', 5, ['bathroom', 'kitchen']])
```

If we want to access the key-value pairs, we can use `items()` :

Input

```
print(pet.items())
```

Output

```
dict_items([('species', 'Tub'), ('age', 5), ('habitat', ['bathroom', 'kitchen'])])
```

This is convenient if we want to loop through the key-value pairs.

4.7 2.7. Sets

Sets are unordered collections of unique elements and are useful when we want to store a collection of items that are not in any particular order and we don't want to store duplicate items.

4.7.1 2.7.1. Creating a Set

Create an empty set is similar to creating an empty list, you can use `set()` . Although the brackets `{}` are also used to create a set, it is actually creating an empty dictionary. To create an empty set, you need to use `set()` :

Input

```
empty_dict = {} # This is to create an empty dictionary
empty_set = set() # This is to create an empty set
print(type(empty_dict))
print(empty_set)
print(type(empty_set))
```

Output

```
<class 'dict'>
set()
<class 'set'>
```

Create a set based on the same strings as the list `pet` :

Input

```
pet = {'Tub', 'Furrytail', 'Cat', 'Barkalot'}
print(type(pet))
print(pet)
```

Output

```
<class 'set'>
{'Tub', 'Cat', 'Barkalot', 'Furrytail'}
```

`set` Doesn't Care About Order

Notice that the order of the elements in the set is different from the order of the elements in the list `pet`. This is because sets are unordered collections of unique elements. If you run it multiple times, the order will be different as sets don't care about the order of the elements.

4.7.2 2.7.2. Set Methods

If we create a set with duplicate elements, the set will only keep one copy of the element:

Input

```
# Sets throw away the duplicate elements.
pet = {'Tub', 'Furrytail', 'Cat', 'Barkalot', 'Tub'}
print(pet)
```

Output

```
{'Tub', 'Cat', 'Barkalot', 'Furrytail'}
```

Sets are optimized for checking whether an element is contained in the set.

Input

```
pet = {'Tub', 'Furrytail', 'Cat', 'Barkalot'}
print('Tub' in pet)
print('Tub' not in pet)
```

Output

```
True
False
```

We can check if two sets of pet in common using `intersection()`:

Input

```
# What these pet have in common
pet_1 = {'Tub', 'Furrytail', 'Cat', 'Barkalot'}
pet_2 = {'Tub', 'Furrytail', 'Bumblefluff ', 'Whiskerfloof'}
print(pet_1.intersection(pet_2))
```

Output

```
{'Tub', 'Furrytail'}
```

We can also check if two sets of pet not in common using `difference()`:

Input

```
# What these pet don't have in common
pet_1 = {'Tub', 'Furrytail', 'Cat', 'Barkalot'}
pet_2 = {'Tub', 'Furrytail', 'Bumblefluff ', 'Whiskerfloof'}
print(pet_1.difference(pet_2))
```

Output

```
{'Cat', 'Barkalot'}
```

Finally, we can also union both of the sets using `union()`:

Input

```
# What these pet have in common and what they don't have in common
print(pet_1.union(pet_2))
```

Output

```
{'Tub', 'Cat', 'Barkalot', 'Furrytail', 'Bumblefluff ', 'Whiskerfloof'}
```

5. 3. Control and Functions

5.1 3.1. Conditionals and Booleans

`if` statements are used to control the flow of the program. It allows us to execute a block of code if a certain condition is met. `else` statements are used to execute a block of code if the condition is not met. `elif` is to add more conditions to the `if` statement, which stands for `else if`.

5.1.1 3.1.1. `if` and Boolean Values

Input

```
if True:
    print("It's true!")
```

Output

```
It's true!
```

What if we change the condition to `False`?

Input

```
if False:
    print("It's true!")
```

Output

```
# Nothing will be printed
```

This is because that the condition is `False`, so the block of code is not executed.

In real practice, we don't hardcode the condition to be `True` or `False`, we basically assess the condition to be `True` or `False`

For example, we can use comparison operators to compare two values:

Input

```
pet = 'Tub'
if pet == 'Tub':
    print("It's true!")
```

Output

```
It's true!
```

Comparison Operators

Recall the operator we used in the previous chapter. This time, we use them in the condition, plus the identity operator `is`.

- Equal: `==`
- Not Equal: `!=`
- Greater Than: `>`
- Less Than: `<`
- Greater or Equal: `>=`
- Less or Equal: `<=`
- Identity: `is`

5.1.2 3.1.2. `if` , `else` and `elif`

Let's continue on the previous example. We campare `pet` and `Tub` to see if the pet is `Tub`

Input

```
pet = 'Tub'
if pet == 'Tub':
    print("Pet is Tub!")
else:
    print("Pet is not Tub!")
```

Output

```
Pet is Tub!
```

If we change the value of `pet` to `Barkalot` , the condition is not met, so the `else` statement is executed.

Input

```
pet = 'Barkalot'
if pet == 'Tub':
    print("Pet is Tub!")
else:
    print("Pet is not Tub!")
```

Output

```
Pet is not Tub!
```

We can also use `elif` to add more conditions to the `if` statement. Here we have two conditions, `pet == 'Tub'` and `pet == 'Barkalot'` . If the first condition is not met, we move on to the next condition. If the second condition is not met, we move on to the `else` statement:

Input

```
pet = 'Barkalot'
if pet == 'Tub':
    print("Pet is Tub!")
elif pet == 'Barkalot':
    print("Pet is Barkalot!")
else:
    print("Pet is not Tub!")
```

Output

```
Pet is Barkalot!
```

5.1.3 3.1.3. `is` vs. `==`

Now, we investigate the difference between `is` and `==` :

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

Here, we have two list with the same values.

Input

```
pet_1 = ['Tub', 'Barkalot', 'Furrytail']
pet_2 = ['Tub', 'Barkalot', 'Furrytail']
```

We use `==` to compare them, and the result is `True` .

Input

```
print(pet_1 == pet_2)
```

Output

```
True
```

This is because that the values of the two lists are the same.

If we use `is` to compare them, the result is `False`.

Input

```
print(pet_1 is pet_2)
```

Output

```
False
```

The reason is that `pet_1` and `pet_2` point to different objects in memory.

We can check out the memory address of the two objects using `id()`:

Input

```
print(id(pet_1))
print(id(pet_2))
```

Output

```
2398480322752
2398482691520
```

You can see that the memory addresses are different.

But if we assign `pet_2` to `pet_1`, they will point to the same object in memory, and of course have the same values.

Input

```
pet_2 = pet_1
print(pet_1 == pet_2)
print(pet_1 is pet_2)
```

Output

```
True
True
```

Now, the memory addresses are the same:

5.1.4 3.1.4. `and`, `or` and `not`

We can use `and` and `or` to combine conditions.

- `and` means both conditions must be met
- `or` means at least one condition must be met

For example, we want to check if **both** the account name is `Tub` and the passcode is correct by using `and`:

Input

```
account_name = 'Tub'
account_passcode = True

if account_name == 'Tub' and account_passcode:
    print("Login successful!")
else:
    print("Login failed!")
```

Output

```
Login successful!
```

If we want to know if at least one of the account name or account passcode is correct, we use `or` :

Input

```
account_name = 'Tub'
account_passcode = True

if account_name == 'Tub' or account_passcode:
    print("Name or passcode is correct!")
else:
    print("Name and passcode are incorrect!")
```

Output

```
Name or passcode is correct!
```

If we want to negate a condition, we use `not` .

- `not` means the condition must not be met

  Input

  ```
  account_passcode = True
  if not account_passcode:
      print("Please enter your passcode!")
  else:
      print("Login successful!")
  ```

  Output

  ```
  Login successful!
  ```

  Here, we use `not` to negate the condition `account_passcode == True` to `account_passcode == False` . Therefore, the condition must not be met, and the `else` statement is not executed.

If we remove `not` , the condition must be met, which is `account_passcode == True` , and the `if` statement is executed.

Input

```
account_passcode = True
if account_passcode:
    print("Please enter your passcode!")
else:
    print("Login successful!")
```

Output

```
Please enter your passcode!
```

5.1.5 3.1.5. `in` and `not in`

We can use `in` to check if a value is in a list.

- `in` means the value must be in the list
- `not in` means the value must not be in the list

Here is the example of `in` :

Input

```
pets = ['Tub', 'Barkalot', 'Furrytail']
if 'Tub' in pets:
    print("Tub is in the list!")
else:
    print("Tub is not in the list!")
```

Output

```
Tub is in the list!
```

If we use `not in`, the condition is negated, and the `else` statement is executed:

Input

```
pets = ['Tub', 'Barkalot', 'Furrytail']
if 'Tub' not in pets:
    print("Tub is not in the list!")
else:
    print("Tub is in the list!")
```

Output

```
Tub is in the list!
```

5.1.6 3.1.6. False Values

False Values

In Python, the following values are considered as `False`:

- `False`
- `None`
- `0` (any zero numeric types)
- Empty sequence. e.g., `''`, `()`, `[]`.
- Empty mapping. e.g., `{}`.

`False` is considered as False:

Input

```
account_name = False
if account_name:
    print("Login successful!")
else:
    print("Please enter your account name!")
```

Output

```
Please enter your account name!
```

`None` is considered as False:

Input

```
account_name = None
if account_name:
    print("Login successful!")
else:
    print("Please enter your account name!")
```

Output

```
Please enter your account name!
```

Only number `0` is considered as False:

Input

```
account_name = 0
if account_name:
    print("Login successful!")
else:
    print("Please enter your account name!")
```

Output

```
Please enter your account name!
```

Empty sequences, e.g. `''`, `()`, `[]`, are considered as False:

Input

```
account_name = ''
if account_name:
    print("Login successful!")
else:
    print("Please enter your account name!")
```

Output

```
Please enter your account name!
```

Empty dictionary is considered as False

Input

```
account_name = {}
if account_name:
    print("Login successful!")
else:
    print("Please enter your account name!")
```

Output

```
Please enter your account name!
```

5.2 3.2. Functions

In this section, we will walk you through various examples related to functions in Python, exploring different concepts such as defining and calling functions, arguments, default values, and more.

5.2.1 3.2.1. Functions Basics

First, let's define a simple function called hello_tub:

Input

```
def hello_tub():
    pass
print(hello_tub)
```

Output

```
<function hello_world at 0x000001E5F1F9B790>
```

This function does nothing, as it contains a pass statement. When you print the function, you will get the memory address of the function object:

Input

```
print(hello_tub())
```

Output

```
None
```

Calling the function with `hello_tub()` returns None, as the function has no return statement.

Now, let's modify the hello_tub function to print a greeting:

Input

```
def hello_tub():
    print('Hello Tub')
hello_tub()
```

Output

```
Hello Tub
```

Using functions is advantageous when you want to reuse code. For instance, if you want to change the greeting from `Tub` to `Barkalot`, you only need to modify the function's implementation, and all the calls to the function will use the updated greeting.

For example, if we want to call `Hello Tub` twice, we can do the following:

Input

```
print('Hello Tub')
print('Hello Tub')
```

Output

```
Hello Tub
Hello Tub
```

But this is not convenient, as we have to repeat the same code twice. Instead, we can define a function and call it twice, and even if we want to change both `Tub`:

Input

```
def hello_tub():
    print('Hello Barkalot')
hello_tub()
hello_tub()
```

Output

```
Hello Barkalot
Hello Barkalot
```

Functions can also return values, take parameters, and have default values for parameters. Here are some examples:

Input

```
def hello_tub():
    return 'Hello Tub'
print(hello_tub())
```

Output

```
Hello Tub
```

We can also call the function with a method, e.g. `lower()`, to convert the returned value to lowercase:

Input

```
print(hello_tub().lower())
```

Output

```
hello tub
```

Functions can also return values, take parameters, and have default values for parameters. Here are some examples:

Input

```
def hello_tub(name):
    return 'Hello ' + name
print(hello_tub('Tub'))
```

Output

```
Hello Tub
```

Input

```
def hello_tub(name):
    return 'Hello {}'.format(name)
print(hello_tub('Tub'))
```

Output

```
Hello Tub
```

We set up a default value for the name parameter, so that if we don't pass a value for name, the function will use the default value:

Input

```
def hello_tub(greeting, name = 'Tub'):
    return '{}, {}'.format(greeting, name)
print(hello_tub('Hello'))
```

Output

```
Hello, Tub
```

When we pass a value for name, the default value is ignored:

Input

```
print(hello_tub('Hello', 'Barkalot'))
```

Output

```
Hello, Barkalot
```

5.2.2 3.2.2. Positional Arguments

In Python, non-default arguments (those without default values) must be defined before default arguments (those with default values). In the given code, the greeting parameter has a default value, while name does not. This causes a SyntaxError.

Input

```
def hello_tub(greeting = 'Hello', name):
    return '{}, {}'.format(greeting, name)
```

Output

```
SyntaxError: non-default argument follows default argument
```

To fix this issue, you should move the non-default argument before the default argument:

Input

```
def hello_tub(name, greeting='Hello'):
    return '{}, {}'.format(greeting, name)
```

Now, the function works as expected, and you can call it with or without providing a greeting argument:

Input

```
print(hello_tub('Tub'))
print(hello_tub('Tub', 'Hi'))
```

Output

```
Hello, Tub
Hi, Tub
```

Below let's go through a real example how to find the number of days in a month

Example - Find the number of days in a month

Credits: Python Standard Library, and Corey Schafer.

Number of days per month. First value placeholder for indexing purposes.

month_days

```
month_days = [0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

The `is_leap()` function takes a year as input and returns True if it's a leap year, and False otherwise. Leap years are those divisible by 4, but not divisible by 100, unless they are also divisible by 400.

is_leap()

```
def is_leap(year):
    """
    Return True for leap years, False for non-leap years.
    """
    return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)
```

The `days_in_month()` function takes a year and a month as input and returns the number of days in that month for that year. It checks if the input year is a leap year and adjusts the number of days in February accordingly. If an invalid month is provided, it returns `Invalid Month`.

days_in_month()

```
def days_in_month(year, month):
    """
    Return number of days in that month in that year.
    """
    if not 1 <= month <= 12:
        return 'Invalid Month'

    if month == 2 and is_leap(year):
        return 29

    return month_days[month]
```

Finally, the code demonstrates calling the `is_leap()` and `days_in_month()` functions with specific inputs:

Print

```
print(is_leap(2023))
print(days_in_month(2023, 4))
```

Output

```
False
30
```

5.2.3 3.2.3. `*args` and `**kwargs`

In the following code, there are several concepts being illustrated: `*args`, `**kwargs`, and two custom functions `is_leap()` and `days_in_month()`.

`*args` and `**kwargs` are used in function definitions to allow passing a variable number of arguments. `*args` is used for passing a variable number of non-keyword (positional) arguments, while `**kwargs` is used for passing a variable number of keyword arguments.

Input

```
def pet_info(*args, **kwargs):
    print(args)
    print(kwargs)
pet_info('Tub', 'Barkalot', 'Furrytail', pet1 = 'Tub', pet2 = 'Barkalot', pet3 = 'Furrytail')
```

Output

```
('Tub', 'Barkalot', 'Furrytail')
{'pet1': 'Tub', 'pet2': 'Barkalot', 'pet3': 'Furrytail'}
```

In the `pet_info()` function, both `*args` and `**kwargs` are used. When you call the function with different types of arguments, you can see how they are grouped and printed:

Input

```
def pet_info(*args, **kwargs):
    print(args)
    print(kwargs)

favorite_food = ['Carrot', 'Brocolli', 'Ice Cream']
info = {'name': 'Tub', 'age': 25}
pet_info(favorite_food, info)
```

In the first call to pet_info, we pass a list `favorite_food` and a dictionary info as arguments without using the `*` or `**` unpacking operators. This means the entire list and dictionary are treated as single positional arguments. The output shows that args contains a tuple with two elements: the list `favorite_food` and the dictionary info. Since we didn't provide any keyword arguments, kwargs is an empty dictionary.

Output

```
(['Carrot', 'Brocolli', 'Ice Cream'], {'name': 'Tub', 'age': 25})
{}
```

In the second call to pet_info, we use the `*` and `**` unpacking operators to pass the list favorite_food and the dictionary info as individual elements. The `*` operator unpacks the list elements as positional arguments, and the `**` operator unpacks the dictionary items as keyword arguments. In this case, the output shows that args contains a tuple with three elements ('`Carrot`', '`Brocolli`', '`Ice Cream`') and kwargs contains a dictionary with the keys and values from the info dictionary.

Input

```
pet_info(*favorite_food, **info)
```

Output

```
('Carrot', 'Brocolli', 'Ice Cream')
{'name': 'Tub', 'age': 25}
```

5.2.4 3.2.3. Variable Scope - LEGB rule

In this section, we are discussing variable scope in Python, which determines where a variable can be accessed or modified. Python searches for a variable following the LEBG rule and order: Local, Enclosing, Global, and Built-in.

Local A variable defined within a function has local scope. It can only be accessed inside that function.

Input

```
def test():
    y = 'local variable y'
    print(y)

test()
```

Output

```
local variable y
```

Global

A variable defined outside any function has global scope. It can be accessed both inside and outside of functions.

In the following example, we define a variable `x` outside of the `test()` function. We can access and modify this variable inside the function.

Input

```
x = 'global variable x'

def test():
    y = 'local variable y'
    print(x)
```

```
test()
print(x)
```

Output

```
global variable x
global variable x
```

Here the results are the same because we are accessing the global variable `x` inside the function. However, if we try to access the local variable `y` outside of the function, we get an error.

Input

```
print(y)
```

Output

```
NameError: name 'y' is not defined
```

What if we have a variable with the same name inside and outside of a function? In this case, the local variable takes precedence over the global variable. The following example demonstrates this:

Input

```
x = 'global variable x'
def test():
    x = 'local variable x'
    print(x)

test()
print(x)
```

Output

```
local variable x
global variable x
```

This example shows that the python searches for a variable in the local scope first. If it doesn't find it, it searches the global scope. This is the reason that we get the local variable `x` inside the function first and the global variable `x` next

If it doesn't find it there, it will throw an error.

In the next example, we demonstrate how to use the global keyword to change the value of a global variable within a function.

Input

```
# What if we want to set a new global x
x = 'global variable x'
def test():
    global x
    x = 'local variable x'
    print(x)

test()
print(x)
```

Output

```
local variable x
local variable x
```

In this example, we use the `global` keyword to change the value of the global variable `x` inside the function, although this is not recommended in the practice because it can lead to unexpected behavior and make the code difficult to debug and review.

You can also do the following:

Input

```
def test():
    global x
    x = 'local variable x'
    print(x)
```

```
test()
print(x)
```

Output

```
local variable x
local variable x
```

However, it is not recommended to use global often.

Input

```
def test(z):
    print(z)

test('local variable z')
print(z)
```

Output

```
local variable z
NameError: name 'z' is not defined
```

Built-in

In this example, we are discussing the built-in scope in Python. Built-in scope refers to the predefined functions and variables available in Python, which are part of the standard library.

Python has a set of built-in functions, like min(), max(), print(), etc., which are readily available for use.

Input

```
# Built-in
import builtins
# print(dir(builtins))

minimum = min([1,2,3])
print(minimum)
```

Output

```
1
```

However, you should avoid overwriting built-in functions with your own functions or variables. Doing so can lead to errors or unintended behavior.

Input

```
# If we overwrite the built-in function min()
def min():
    pass

m = min([1,2,3])
print(m)
```

Output

```
TypeError: min() takes 0 positional arguments but 1 was given
```

To avoid conflicts with built-in functions, it's a good practice to use different names for your own functions.

So the best way to do this is to use a different name instead of the default name `min()`.

Input

```
def find_min():
    pass

minimum = min([1,2,3])
print(minimum)
```

Output

```
1
```

Being mindful of built-in functions and avoiding name conflicts will help you write clean, error-free code.

Enclosing

In this example, we discuss the concept of enclosing scope in Python. Enclosing scope is the scope of variables that are defined in an outer function but not in the global scope. Enclosing scope variables are accessible from the inner function.

Let's look at an example:

Input

```
x = 'global variable x'
def outer():
    x = 'local-outer variable x'

    def inner():
        x = 'local-inner variable x'
        print(x) # 1st print

    inner() # 1st call for 1st print
    print(x) # 2nd print


outer() # 2nd call for 1st print and 2nd print
print(x) # 3rd print
```

Output

```
local-inner variable x
local-outer variable x
global variable x
```

The `outer()` function has its own local variable `x`, and the `inner()` function also has its own local variable x. When we call the functions, the inner function prints its local variable `x`, the outer function prints its local variable `x`, and then the global variable x is printed.

Now let's use the nonlocal keyword to modify the enclosing variable from the inner function:

Input

```
x = 'global variable x'
def outer():
    x = 'local-outer variable x'

    def inner():
        nonlocal x # Make our local-inner variable x to be the enclosing variable x
        x = 'local-inner variable x'
        print(x)

    inner()
    print(x)

outer()
print(x)
```

Output

```
local-inner variable x
local-inner variable x
global variable x
```

In this case, we use the nonlocal keyword inside the `inner()` function to indicate that we want to modify the enclosing variable `x` (the one defined in the `outer()` function) instead of creating a new local variable. When the functions are called, both the inner and outer functions print the modified enclosing variable `x`, and then the global variable `x` is printed.

6. 4. Advanced Formatting

In Chapter 2, we have seen the basic string formatting. In this chapter, we will see some more advanced formatting examples upon on that.

6.1 4.1. Formatting with placeholders

Let's see the following example first:

Input

```
species_1 = {'species': 'Tub', 'age': 5}
sentence = 'My name is ' + species_1['species'] + ' and I am ' + str(species_1['age']) + ' years old.'
print(sentence)
```

Output

```
My name is Tub and I am 5 years old.
```

We can see there are a lot of blank space we have to manually put in the beginning and the end of each string. We also have to cast the integer/number, `species_1['age']`, into strings by using `str()`.

Of course, there is a better way to do this:

Input

```
sentence = 'My name is {} and I am {} years old.'.format(species_1['species'], species_1['age'])
print(sentence)
```

Output

```
My name is Tub and I am 5 years old.
```

We can see that we don't have to cast the integer/number into strings anymore.

We can also use the index of the placeholder to specify the order of the arguments:

Input

```
sentence = 'My name is {0} and I am {1} years old.'.format(species_1['species'], species_1['age'])
print(sentence)
```

Output

```
My name is Tub and I am 5 years old.
```

This is another example using the index of the placeholder with repeated arguments:

Input

```
tag = 'p'
text = 'This is a paragraph'
sentence = '<{0}>{1}</{0}>'.format(tag, text)
print(sentence)
```

Output

```
<p>This is a paragraph</p>
```

This is very useful when we have a placeholder that you want to reuse. Let's see another example:

Input

```
sentence = 'My name is {0} and I am {1} years old. My friend Barkalot is also {1} years old.'.format(
    species_1['species'], species_1['age'])
print(sentence)
```

Output

```
My name is Tub and I am 5 years old. My friend Barkalot is also 5 years old.
```

There is another way to do this by using the key in the placeholder instead of the index:

Input

```
sentence = 'My name is {0[species]} and I am {1[age]} years old.'.format(species_1, species_1)
print(sentence)
```

Output

```
My name is Tub and I am 5 years old.
```

Here, we call the value of the key `species` in the dictionary `species_1` by using `{0[species]}`.

But this is redundant because we have to repeat the same argument twice.

We can do the following instead:

Input

```
sentence = 'My name is {0[species]} and I am {0[age]} years old.'.format(species_1)
print(sentence)
```

Output

```
My name is Tub and I am 5 years old.
```

We can also use the index of a list in the placeholder:

Input

```
species = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
sentence = 'I am {0[1]} and my friend is {0[3]}.'.format(species)
print(sentence)
```

Output

```
I am Furrytail and my friend is Barkalot.
```

We can also access the attributes of an object in the same way. For example, we define a class called `Species` that represents a species with a `species` and an `age`. The class has an `__init__` method that initializes the instance variables species and age. We then create an instance of the `Species` class called `species_new` with the name 'Jerry' and the age `88`.

You then create an instance of the Species class called species_new with the name 'Jerry' and the age 88.

Input

```
class Species:

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

species_new = Species('Jerry', 88)
```

Now we can access the attributes of the object `species_new` by using the index of the placeholder:

Input

```
sentence = 'My name is {0.species} and I am {0.age} years old.'.format(species_new)
print(sentence)
```

Output

```
My name is Jerry and I am 88 years old.
```

The string contains placeholders `{0.species}` and `{0.age}` which will be replaced by the species and age attributes of the `species_new` object. The `0` in the placeholders refers to the first argument passed to the `.format()` method, which is `species_new`.

This time, we are using keyword arguments to pass the values that will replace the placeholders in the string. The string contains placeholders `{species}` and `{age}`, which will be replaced by the values provided as keyword arguments in the `.format()` method.

Input

```
sentence = 'My name is {species} and I am {age} years old.'.format(species='Jerry', age=88)
print(sentence)
```

Output

```
My name is Jerry and I am 88 years old.
```

6.2 4.2. `**` in `.format()

In the format() method, the ** operator can be used to unpack a dictionary that contains the keyword arguments for the placeholders in the string. This can be a convenient way to format a string using a dictionary that has keys matching the placeholders in the string.

Here's an example:

Input

```
pet = {'species': 'Tub', 'age': 5}
sentence = 'My name is {species} and I am {age} years old.'.format(**pet)
print(sentence)
```

Output

```
My name is Tub and I am 5 years old.
```

In this example, we have a dictionary `pet` containing the keys `species` and `age`. We then use the `**` operator to unpack the dictionary when calling the `.format()` method on the string. The values from the dictionary are used to replace the placeholders in the string, resulting in the sentence `"My name is Tub and I am 5 years old."`

This method is useful when we print out the dictionaries, which is more readable.

Why use `**species` instead of `species`?

When you use `**` before a dictionary in a function call, like in the `.format()` method, it is known as dictionary unpacking. It allows you to pass the key-value pairs in the dictionary as named (keyword) arguments to the function.

When a dictionary is passed as a keyword argument in this way, the keys in the dictionary are treated as the parameter names and the corresponding values are passed as the parameter values.

So, in this case, the keys in the `pet` dictionary (`'species'` and `'age'`) are treated as parameter names in the sentence string, and their corresponding values (`'Tub'` and `5`) are passed as parameter values.

If you were to pass the pet dictionary without the double asterisks, like so:

Input

```
sentence = 'My name is {species} and I am {age} years old.'.format(species)
print(sentence)
```

Output

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'species'
```

The `.format()` method would be looking for a single argument that is a dictionary, rather than separate keyword arguments, and would raise a TypeError.

6.3 4.3. Formatting Numbers

In this block, we are using the `.format()` method to insert values into a string. The `{}` serves as a placeholder for the value that will be inserted. The output will be a series of strings with the respective values from the loop.

Input

```
for i in range(1, 6):
    sentence = 'The value is {}'.format(i)
    print(sentence)
```

Output

```
Output:
The value is 1
The value is 2
The value is 3
The value is 4
The value is 5
```

Padding with zeros (width 2): In this block, we're using the :02 format specifier within the placeholder to pad the value with zeros, ensuring a minimum width of 2 characters. This is useful for maintaining consistent formatting when dealing with numbers of varying lengths.

Input

```
for i in range(1, 6):
    sentence = 'The value is {:02}'.format(i)
    print(sentence)
```

Output

```
Output:
The value is 01
The value is 02
The value is 03
The value is 04
The value is 05
```

Padding with zeros (width 3): Similarly, we can use the :03 format specifier to pad the value with zeros, ensuring a minimum width of 3 characters. This results in a more extensive padding for the smaller numbers, keeping the output format consistent.

Input

```
for i in range(1, 6):
    sentence = 'The value is {:03}'.format(i)
    print(sentence)
```

Output

```
Output:
The value is 001
The value is 002
The value is 003
The value is 004
The value is 005
```

By using `:.2f` within the placeholder, we can format a floating-point number to display two decimal places.

Input

```
# We can also use the `:.2f` with two decimal places
e = 2.71828
sentence = 'e is equal to {:.2f}'.format(e)
print(sentence)
```

Output

```
e is equal to 2.72
```

By using `:,` within the placeholder, we can format a number with a thousands separator, `,`.

Input

```
sentence = '1 KM is equal to {:,.2f} meters'.format(1000)
print(sentence)
```

Output

```
1 KM is equal to 1,000.00 meters
```

6.4 4.4. Formatting Date and Time with `datetime`

When formatting date and time, we can refer to the strftime and strptime behavior for the format codes.

In the following code, we are using Python's `datetime` module to create a `datetime` object representing a specific date and time. We create a `datetime` object for `April 1, 2023, 10:10:30 AM`, and print it.

Input

```
import datetime
today_date = datetime.datetime(2023, 4, 1, 10, 10, 30)
print(today_date)
```

Output

```
2023-04-01 10:10:30
```

We can use the `strftime` method to format the date in a more human-readable format. The `strftime` method allows you to create custom date and time formats by using format codes. We can use the `strftime` method to format the date.

But we have to import the `datetime` module first.

Input

```
import datetime
today_date = datetime.datetime(2023, 4, 1, 10, 10, 30)
today_date = '{:%B %d, %Y}'.format(today_date)
print(today_date)
```

Output

```
April 01, 2023
```

Here, we use the format codes %B, %d, and %Y to display the full month name, the day of the month with a leading zero, and the year with the century, respectively. The resulting formatted date is passed to the format function and printed.

We can also use the `strptime` method to parse the string into a date:

Input

```
import datetime
date_str = 'April 01, 2023'
today_date = datetime.datetime.strptime(date_str, '%B %d, %Y')
print(today_date)
```

Output

```
2023-04-01 00:00:00
```

n this example, we have the date string `'April 01, 2023'. The format codes used to parse this string are `%B`, `%d`, and `%Y`, which represent the full month name, the day of the month with a leading zero, and the year with the century, respectively. The strptime method reads the date string and creates a datetime object with the provided date information.

Input

```
import datetime
today_date = datetime.datetime(2023, 4, 1, 10, 10, 30)
```

```
sentence = 'Today is {0:%B %d, %Y} on {0:%A}, and {0:%Y} has passed {0:%j} days'.format(today_date)
print(sentence)
```

Output

```
Today is April 01, 2023 on Thursday, and 2023 has passed 091 days
```

We can also use the `timetuple` method to remove the `0` in the beginning of `091`:

Input

```
sentence = 'Today is {0:%B %d, %Y} on {0:%A}, and {0:%Y} has passed {1:d} days'.format(today_date, today_date.timetuple().tm_yday)
print(sentence)
```

Output

```
Today is April 01, 2023 on Thursday, and 2023 has passed 91 days
```

7. 5. Loops and Comprehensions

7.1 5.1. Loops

This chapter provides an overview of loops and iterations in Python, specifically focusing on the `for`, `while`, and `else` statements.

7.1.1 5.1.1. For Loops

In this section, we will cover the following statements and functions:

- Basic Usage: The for loop iterates over a list of pets and prints each pet's name. This is the simplest usage of a for loop.
- `break` Statement: The for loop is used in combination with a conditional statement and break, which terminates the loop once a specific condition is met.
- `continue` Statement: The continue statement is used to skip the rest of the current loop iteration and immediately start the next one.
- Nested for Loops: This demonstrates the concept of nested loops, where a for loop is contained within another for loop.
- `range()` Function: The `range()` function generates a sequence of numbers over which the for loop iterates. The function can be called with different numbers of arguments to change the start, end, and step size of the sequence.

This chapter provides an excellent foundation for understanding loops in Python.

Basic Usage

Input

```
pets = ['Tub', 'Barkalot', 'Furrytail']
for pet in pets:
    print(pet)
```

Output

```
Tub
Barkalot
Furrytail
```

This code creates a list of pet names, then uses a `for` loop to iterate over each item in the list. Each time through the loop, it prints the current pet name.

`break` Statement

Input

```
pets = ['Tub', 'Barkalot', 'Furrytail']
for pet in pets:
    if pet == 'Barkalot':
        print('We got Barkalot!')
        break
    print(pet)
```

Output

```
Tub
We got Barkalot!
```

This code is similar to the previous example, but it includes an `if` statement that checks whether the current pet name is `Barkalot`. If it is, the code prints a special message and then uses the `break` statement to immediately exit the loop.

`continue` Statement

Input

```
# `continue` statement skip the current iteration and continue with the next.
# Here we skip the `Barkalot` pet and print `We got Barkalot!` instead.
pets = ['Tub', 'Barkalot', 'Furrytail']
for pet in pets:
    if pet == 'Barkalot':
        print('We got Barkalot!')
```

```
        continue
    print(pet)
```

Output

```
Tub
We got Barkalot!
Furrytail
```

Again, this code is similar to the previous examples. The difference is that when the current pet name is `Barkalot`, it uses the `continue` statement to immediately start the next iteration of the loop, skipping the `print(pet)` statement for `Barkalot`.

Nested for Loops

Input

```
for pet in pets:
    for letter in 'ab':
        print(letter, pet)
```

Output

```
a Tub
b Tub
a Barkalot
b Barkalot
a Furrytail
b Furrytail
```

Here, the outer `for` loop iterates over the list of pet names, and the inner `for` loop iterates over the string 'ab'. For each combination of pet name and letter, it prints the letter and the pet name.

`range()` Function

Input

```
for pet in pets:
    for num in range(2):
        print(num, pet)
```

Output

```
0 Tub
1 Tub
0 Barkalot
1 Barkalot
0 Furrytail
1 Furrytail
```

This code is the same as the previous example, but the inner loop iterates over the numbers produced by `range(2)`, which are 0 and 1.

`range()` Function with Start and End

Input

```
for num in range(0, 5):
    print(num)
```

Output

```
0
1
2
3
4
```

This code uses the `range()` function to generate a sequence of numbers from 0 to 4. The `for` loop iterates over these numbers, printing each one.

`range()` Function with Start, End, and Step

Input

```
for num in range(0, 5, 2):
    print(num)
```

Output

```
0
2
4
```

This code is similar to the previous example, but it adds a step size of 2 to the `range()` function. This means it only generates every second number in the range from 0 to 4, so the `for` loop prints the numbers 0, 2, and 4.

7.1.2 5.1.2. While Loops

In Python, a `while` loop repeatedly executes a target statement as long as a given condition is true.

For example:

Input

```
num = 1
while num < 5:
    print(num)
    num += 1
```

Output

```
1
2
3
4
```

The `while` loop above continues executing until the condition `num < 5` is no longer true, which occurs after the fourth iteration.

Using `break` with While Loop

The `break` statement is used to exit a `while` loop prematurely. When a `break` statement is encountered inside the loop, the loop is immediately terminated, and program control resumes at the next statement following the loop.

In the following code, `break` is triggered when `num` equals 3, causing an early exit from the loop.

Input

```
num = 1
while num < 5:
    if num == 3:
        break
    print(num)
    num += 1
```

Output

```
1
2
```

`break` in an Infinite Loop

An infinite `while` loop can be created using `while True:`. This loop will run indefinitely unless it encounters a `break` statement.

`break` is used to stop an otherwise infinite loop when num equals 3:

Input

```
num = 1
while True:
    if num == 3:
        break
    print(num)
    num += 1
```

Output

```
1
2
```

If Statement versus While Loop

An `if` statement checks a condition once, whereas a `while` loop continues to execute the block of code as long as the condition is true.

Input

```
num = 1
if num < 5:
    print(num)
    num += 1
```

Output

```
1
```

In this code, the `if` statement checks the condition `num < 5` once and then executes the block of code if the condition is true. After that, it doesn't check the condition again or repeat the block of code. This is the main difference between a `while` loop and an `if` statement.

7.1.3 5.1.3. Else Clause with Loops

The `else` clause in Python can also be used with loops. Unlike in an `if` statement, where `else` executes when the `if` condition is false, with loops, the else clause executes after the loop completes normally, i.e., when no `break` statement has been encountered.

If-Else

The else clause executes after the loop completes normally.

Here's a simple example of an `if-else` statement:

Input

```
tub_age = 5

if tub_age > 18:
    print('Tub is an adult.')
else:
    print('Tub is a child.')
```

Output

```
Tub is a child.
```

The `else` clause is executed because the `if` condition is false.

Else with For Loop

In the context of a `loop`, an else statement can be thought of as a "no break" statement. It will execute once the loop has finished iterating over the items.

Another example may not be so obvious. And always makes people confused:

Input

```
pets = ['Tub', 'Barkalot', 'Furrytail']

for pet in pets:
    print(pet)
else: # More like a `no break` statement
    print('No more pets.')
```

Output

```
Tub
Barkalot
Furrytail
No more pets.
```

The `else` statement associated with a loop (either a `for` or `while` loop) in Python might initially seem confusing since in many other programming languages `else` is associated only with `if` statements.

You may think why the `else` statement is executed?

In Python, the `else` clause in a loop executes when the loop has finished iterating over all items (in a `for` loop) or when the condition becomes false (in a `while` loop), but not when the loop is prematurely ended by a `break` statement.

Let's break down the above example:

- The `for` loop begins iterating over the `pets` list. For each pet in `pets`, it prints the pet's name.
- When there are no more items left in the `pets` list, the loop's condition becomes false (it has run out of items to process). At this point, instead of just ending, it checks if there is an `else` clause associated with it.
- Since there is an `else` clause, it runs the code inside the `else` block. The print statement inside the `else` block runs, outputting `No more pets.`.

So, the phrase `no break` used in comments beside the `else` statement means that if the loop finishes its iterations normally without encountering a `break` statement (i.e., it was not forced to stop prematurely), the code inside the else block will be executed.

This behavior is particularly useful when you use a loop to search for an item in a list or another data structure. If the item is found, you can use the `break` statement to stop the loop, and the `else` block will be ignored. If the item isn't found and the loop ends normally after checking all items, the `else` block will run, allowing you to handle the case where the item isn't found.

To understand this, let's have a more concrete example:

In `for` loops

`else` terminated by `break` **`else` not terminated by `break` and plays as `no break`**

Input

```
for pet in pets:
    print(pet)
    if pet == 'Barkalot':
        break
else:
    print('No more pets.')
```

Output

```
Tub
Barkalot
```

The `else` statement is not executed because the loop is terminated by the `break` statement.

Input

```
for pet in pets:
    print(pet)
    # Now the loop is not terminated by the `break` statement
    # as the condition is never met  as the above example.
    # You can remove the `if` statement, and it will still work.
    if pet == 'NOT EXIST':
        break
else: # More like a `no break` statement.
    print('No more pets.')
```

Output

```
Tub
Barkalot
Furrytail
No more pets.
```

This is also the same as for `while` loop.

`else` within `while` loop and terminated by `break` `else` within `while` loop and plays as `no break`

Input

```
age = 7
while age >=5:
    print(age)
    age -= 1
else: # More like a `no break` statement
    print('End')
```

Output

```
7
6
5
End
```

The `else` statement is not executed because the loop is terminated by the `break` statement.

Input

```
age = 7
while age >=5:
    print(age)
    age -= 1
    if age == 6:
        break
else: # More like a `no break` statement
    print('End')
```

Output

```
7
6
```

In this exercise, you'll create a function that searches for a specific number in a list using a for `loop` and an `else` clause. The function should print a message indicating whether or not the number was found in the list.

Exercise 1: Search Number in List

Tasks:

1. Write a function `search_number_in_list` that takes two parameters: a list of numbers (`numbers_list`) and a number to search (`search_number`).
2. Inside the function, start a `for` loop that iterates over each number in `numbers_list` .
3. Inside the loop, use an `if` statement to check if the current number equals `search_number` . If it does, print a message like `Number { search_number } found in the list!` , and then use the `break` statement to immediately exit the loop.
4. After the `for` loop, write an `else` clause that prints a message like `Number {search_number} not found in the list.` . This `else` clause should be executed if the `for` loop completes all its iterations without hitting the `break` statement.
5. Test your function with a list of numbers and a search number of your choice.

Here's a skeleton of the function to get you started:

Skeleton to started   Solution

Skeleton

```
def search_number_in_list(numbers_list, search_number)-->str:
# Your code here
pass

# Test the function
numbers = [1, 3, 5, 7, 9, 11]
search_number = 7
search_number_in_list(numbers, search_number)
```

Expected Ouput

```
Number 7 found in the list!
```

Or if you search for a number not in the list:

Expected Ouput

```
Number {search_number} not found in the list.
```

Solution

```
numbers = [1, 3, 5, 7, 9, 11]
search_number = 7

for num in numbers:
    if num == search_number:
        print(f'Number {search_number} found in the list!')
        break
else:
    print(f'Number {search_number} not found in the list.')
```

In this exercise, you'll modify the function `find_first_even` to return the index of the first even number found in the list. If no even number is found, the function should return `None` .

Exercise 2: Find First Even Number Function

Tasks:

1. Modify the function `find_first_even` that takes a list of numbers (`nums`) as parameter.
2. Inside the function, start a `for` loop that iterates over number/s in `nums` .
3. Inside the loop, use an `if` statement to check if the current number is even. If it is, return the current `num` and then use the `break` statement to immediately exit the loop.
4. After the `for` loop, write an `else` clause that returns `None` . This `else` clause should be executed if the `for` loop iterates over all the numbers from the list without hitting the `break` statement. In other words, we didn't find any even number in the list.
5. Test your function with a list of numbers of your choice.

Here's a skeleton of the function to get you started:

Skeleton to started   Solution

Skeleton

```
def find_first_even(nums)-->str:
    # Your code here
    pass

# Test the function
nums = [1, 3, 5, 7, 9, 11]
# It should print `First even number is: 8`
print('First even number is: {}'.format(first_even))
nums = [1, 3, 5, 2, 9, 11]
# It should print `First even number is: None`
print('First even number is: {}'.format(first_even))
```

Expected Ouput

```
First even number is: 8
First even number is: None
```

Solution

```
def find_first_even(nums)->str:
    for num in nums:
        if num % 2 == 0:
            break
    else:
        num = 'None'
    return num
```

7.2 5.2. `enumerate()` function

The `enumerate` function is a built-in function in Python that allows you to loop over something and have an automatic counter, or index tracker. It adds a counter as the key of the enumerate object, alongside the items of the iterable, returning an enumerate object which you can convert to a list, tuple or other data structures. The function signature is as follows:

`enumerate(iterable, start=0)`

- `iterable` : any object that supports iteration
- `start` : the index value from which the counter should start, default is 0

7.2.1 5.2.1. Basic Usage of enumerate()

Input

```
pets = ['Tub', 'Barkalot', 'Furrytail']
for i, pet in enumerate(pets):
    print(i, pet)
```

Output

```
0 Tub
1 Barkalot
2 Furrytail
```

Here, `enumerate(pets)` returns a sequence of tuples, and each tuple consists of two items: the index and the value of the corresponding item in the iterable. `i` and `pet` are tuple unpacking the result returned by enumerate.

7.2.2 5.2.2. Using enumerate with a Different Start Index

In this example, the index starts at `1` (instead of the default 0) because we've set `start=1`.

Input

```
pets = ['Tub', 'Barkalot', 'Furrytail']
for i, pet in enumerate(pets, start=1):
    print(i, pet)
```

Output

```
1 Tub
2 Barkalot
3 Furrytail
```

7.2.3 5.2.3. Practical Usage of enumerate

`enumerate` is particularly useful when you need to track the index of items within a loop. For example, if you want to replace an item in a list:

Input

```
pets = ['Tub', 'Barkalot', 'Furrytail']
for i, pet in enumerate(pets):
    if pet == 'Tub':
        pets[i] = 'Cat'
print(pets)
```

Output

```
['Cat', 'Barkalot', 'Furrytail']
```

In the above code, we use `enumerate` to get the index of each pet. When we find the pet `Tub`, we use the index to replace `Tub` with `Cat` in the original list.

Now it's time to try using `enumerate` in your own code! Try to think of situations where you need both the item and its index within a loop.

Exercise: Find First Even Number and Its Index

Tasks:

1. Modify the function `find_first_even` that takes a list of numbers (`nums`) as parameter.
2. Inside the function, start a `for` loop that iterates over each number in `nums` . You'll need to use the `enumerate` function so that you have access to the index of each number.
3. Inside the loop, use an `if` statement to check if the current number is even. If it is, return the current index and number as a `tuple(index, number)` , and then use the `break` statement to immediately exit the loop.
4. After the `for` loop, write an `else` clause that returns `(None, None)` . This `else` clause should be executed if the `for` loop completes all its iterations without hitting the `break` statement.
5. Test your function with a list of numbers of your choice.

Here's a skeleton of the function to get you started:

Skeleton to started   Solution

Skeleton

```
def find_first_even(nums)->tuple(int, int):
    # Your code here
    return (i, num)

nums = [1, 3, 8, 7, 3, 2, 3]
first_even = find_first_even(nums)
print('The index and value of the first even number are: {}'.format(first_even))
# Output: The index and value of the first even number are: (2, 8)

nums = [1, 3, 1, 7, 3, 9, 3]
first_even = find_first_even(nums)
print('The index and value of the first even number are: {}'.format(first_even))
# Output: The index and value of the first even number are: ('None', 'None')
```

Expected Ouput

```
The index and value of the first even number are: (2, 8)
The index and value of the first even number are: ('None', 'None')
```

Solution

```
def find_first_even(nums)->tuple(int, int):
    for i, num in enumerate(nums):
        if num % 2 == 0:
            break
    else:
        return ('None', 'None')
    return (i, num)
```

7.3 5.3. List Comprehensions

List comprehensions are a powerful feature in Python, allowing you to create lists from existing lists or other iterable objects. They provide a concise way to apply operations to the values in a sequence.

7.3.1 5.3.1. Basic List Comprehensions

Consider the following example where we have a list of ages, and we want to create a new list with the same ages:

Using For loop

Input

```
ages = [5, 12, 3, 56, 24, 78, 1, 15, 44]
age_list = []
for age in ages:
```

```
    age_list.append(age)
print(age_list)
```

Output

```
[5, 12, 3, 56, 24, 78, 1, 15, 44]
```

Now, we make the above for loop into a list comprehension

Input

```
# [item for item in iterable]
age_list = [age for age in ages]
print(age_list)
```

Output

```
[5, 12, 3, 56, 24, 78, 1, 15, 44]
```

Another example, we also apply operations to the values in the sequence. For example, we can add 1 to each age:

Using For loop

Input

```
ages = [5, 12, 3, 56, 24, 78, 1, 15, 44]
age_list = []
for age in ages:
    age_list.append(age + 1)
print(age_list)
```

Output

```
[6, 13, 4, 57, 25, 79, 2, 16, 45]
```

List comprehension

Input

```
# [expression for item in iterable]
age_list = [age + 1 for age in ages]
print(age_list)
```

Output

```
[6, 13, 4, 57, 25, 79, 2, 16, 45]
```

7.3.2 5.3.2. Comparing List Comprehensions and `map`

`map()` is a built-in function that applies a function to each item in an iterable object. It returns a map object, which can be converted into a list or tuple.

Let's see how we can use `map` to add 1 to each age as the previous example:

Input

```
age_list = list(map(lambda age: age + 1, ages))
print(age_list)
```

Output

```
[6, 13, 4, 57, 25, 79, 2, 16, 45]
```

`map` is faster than list comprehension, but list comprehension is more readable.

7.3.3 5.3.3. List Comprehensions with Conditionals

You can also include conditions in your list comprehension. For example, we can create a list that only contains even ages:

Using For loop

Input

```
ages = [5, 12, 3, 56, 24, 78, 1, 15, 44]
age_list = []
for age in ages:
    if age % 2 == 0:
        age_list.append(age)
print(age_list)
```

Output

```
[12, 56, 24, 78, 44]
```

List comprehension

Input

```
# [expression for item in iterable if condition]
age_list = [age for age in ages if age % 2 == 0]
print(age_list)
```

Output

```
[12, 56, 24, 78, 44]
```

Using `lambda` with `filter`

Input

```
# filter(lambda)
age_list = list(filter(lambda age: age % 2 == 0, ages))
print(age_list)
```

Output

```
[12, 56, 24, 78, 44]
```

7.3.4 5.3.4. Real World Example

List comprehensions can be used to solve real-world problems more concisely. For example, let's say you want to create an unordered HTML list from a list of pet names:

Using For loop

Input

```
pets = ['Tub', 'Furrytail', 'Cat', 'Barkalot', 'Bumblefluff ', 'Whiskerfloof']
output = '<ul>\n'
for pet in pets:
    output += '\t<li>{}</li>\n'.format(pet)
    # print('Address of output is {}'.format(id(output)))
output += '</ul>'
print(output)
```

Output

```
<ul>
<li>Tub</li>
<li>Furrytail</li>
<li>Cat</li>
<li>Barkalot</li>
<li>Bumblefluff </li>
<li>Whiskerfloof</li>
</ul>
```

- Tub
- Furrytail
- Cat
- Barkalot
- Bumblefluff
- Whiskerfloof

List comprehension

Input

```
pets = ['Tub', 'Furrytail', 'Cat', 'Barkalot', 'Bumblefluff ', 'Whiskerfloof']
output = '<ul>\n'
# List comprehension to create a list of formatted list items
formatted_pet_names = ['\t<li>{}</li>'.format(pet) for pet in pets]
# Join the list items with newline characters and add the closing </ul> tag
output += '\n'.join(formatted_pet_names) + '\n</ul>'
print(output)
```

Output

```
<ul>
<li>Tub</li>
<li>Furrytail</li>
<li>Cat</li>
<li>Barkalot</li>
<li>Bumblefluff </li>
<li>Whiskerfloof</li>
</ul>
```

7.3.5 5.3.5. Performance Comparison: For Loop vs List Comprehension

Let's compare the performance of using a `for` loop vs a list comprehension to generate a list of formatted numbers:

Input

```
import timeit
import random

# Create a list of 10,000 random numbers between 0 and 100
nums = [random.randint(0, 100) for i in range(10000)]

# For loop implementation
def for_loop():
    output = '<ul>\n'
    for num in nums:
        output += '\t<li>{}</li>\n'.format(num)
    output += '</ul>'
    return output

# List comprehension implementation
def list_comprehension():
    output = '<ul>\n' + ''.join(['\t<li>{}</li>\n'.format(num) for num in nums]) + '</ul>'
    return output

# Measure the execution time of both implementations
for_loop_time = timeit.timeit(for_loop, number=1000)
list_comprehension_time = timeit.timeit(list_comprehension, number=1000)

print("For loop execution time: ", for_loop_time)
print("List comprehension execution time: ", list_comprehension_time)
```

Output

```
For loop execution time:  0.016999999999999999
List comprehension execution time:  0.012999999999999956
```

When running this code, you'll find that the list comprehension implementation is typically faster than the `for` loop implementation. This is because list comprehensions are optimized for performance in Python, and they can often perform the same task more quickly than the equivalent `for` loop. However, the difference in speed may not be significant unless you're dealing with very large data sets.

It's also worth noting that while list comprehensions can be faster and more concise, they can also be harder to read if they become too complex. Therefore, it's important to strike a balance between performance and readability when writing your code.

Pros of List Comprehensions vs For Loops **Cons of List Comprehensions Vs For Loops**

- **Succinctness**: List comprehensions provide a concise way to create lists. They can often achieve the same result as a for-loop in a single, short line of code.
- **Speed**: List comprehensions are generally faster than for-loops because they are specifically optimized for creating new lists.
- **Functionality**: List comprehensions can incorporate conditionals and multiple for-loops, enabling quite complex list creation in a single line.

- **Readability**: List comprehensions can be harder to read than for-loops, especially if they are complex.
- **MemoryUsage**: List comprehensions create new lists in memory, which can cause problems if you're working with very large data sets.
- **Debugging**: List comprehensions can be harder to debug than for-loops, especially if they are complex.

7.4 5.4. zip() Function

In Python, the `zip()` function is used to combine corresponding elements from multiple iterables (like lists or tuples) into tuples. Let's first understand it with an example:

Input

```
ages = [5, 12, 3, 56, 24, 78, 1, 15, 44]
names = ['Tub', 'Barkalot', 'Furrytail']

print(zip(ages, names))
print(list(zip(ages, names)))
```

Output

```
<zip object at 0x0000020F6F6F0A48>
[(5, 'Tub'), (12, 'Barkalot'), (3, 'Furrytail')]
```

What zip does is it takes the first item from each iterable and puts them together in a tuple then it takes the second item from each iterable and puts them together in a tuple and so on. The result is a zip object that we can convert into a list of tuples using the `list()` function.

7.4.1 5.4.1. zip() with for Loop

In the following example, the `zip()` function pairs up the elements from ages and names lists by their indices.

Input

```
ages = [5, 12, 3, 56, 24, 78, 1, 15, 44]
names = ['Tub', 'Barkalot', 'Furrytail']

my_list = []
for age in range(3):
    for name in names:
        my_list.append((age, name))
print(my_list)
```

Output

```
[(0, 'Tub'), (0, 'Barkalot'), (0, 'Furrytail'), (1, 'Tub'), (1, 'Barkalot'), (1, 'Furrytail'), (2, 'Tub'), (2, 'Barkalot'), (2, 'Furrytail')]
```

We can utilize this function along with comprehensions for more complex operations.

7.4.2 5.4.2. zip() with List Comprehension

Similarly, we can use `zip()` with list comprehensions. Let's create a list of tuples where each tuple consists of a number and a pet name:

Input

```
my_list = [(age, name) for age in range(3) for name in names]
print(my_list)
```

Output

```
[(0, 'Tub'), (0, 'Barkalot'), (0, 'Furrytail'), (1, 'Tub'),
(1, 'Barkalot'), (1, 'Furrytail'), (2, 'Tub'), (2, 'Barkalot'), (2, 'Furrytail')]
```

This code creates a tuple for each combination of age and name, and adds it to the list. Here, `age` ranges from `0` to `2`, and `name` is taken from the `names` list.

7.5 5.5. Dictionary Comprehensions

A dictionary comprehension is similar to a list comprehension, but it constructs a dictionary instead of a list.

If we want to create a dictionary that maps each pet's name to its age, we can use a for-loop like this:

Regular For Loop with zip()

Input

```
my_dict = {}
for age, name in zip(ages, names):
    my_dict[name] = age
print(my_dict)
```

Output

```
{'Tub': 5, 'Barkalot': 12, 'Furrytail': 3}
```

Dictionary comprehension with zip()

Input

```
# {key: value for item in iterable}
my_dict = {name: age for name, age in zip(names, ages)}
print(my_dict)
```

Output

```
{'Tub': 5, 'Barkalot': 12, 'Furrytail': 3}
```

Here, `name` : `age` is the key-value pair for each item in the dictionary.

We can also add a condition in the dictionary comprehension to filter the items:

Input

```
# {key: value for item in iterable if condition}
my_dict = {name: age for name, age in zip(names, ages) if age > 10}
print(my_dict)
```

Output

```
{'Barkalot': 12}
```

This will include only the pets that are older than 10 in the dictionary.

7.6 5.6. Set Comprehensions

Set comprehensions work just like list and dictionary comprehensions, but they produce a set, which is an unordered collection of unique elements.

Let's start with an example where we want to create a set from the ages list:

Input

```
ages = [5, 12, 3, 56, 24, 78, 1, 15, 44]
names = ['Tub', 'Barkalot', 'Furrytail']
```

```
my_set = set()
for age in ages:
    my_set.add(age)
print(my_set)
```

Output

```
{1, 3, 5, 12, 15, 44, 56, 78}
```

We're creating a set and adding each age to it. The resulting set includes each age once, even if it appeared multiple times in the list.

We can achieve the same result more succinctly with a set comprehension:

Input

```
# {expression for item in iterable}
my_set = {age for age in ages}
print(my_set)
```

Output

```
{1, 3, 5, 12, 15, 44, 56, 78}
```

Here, `age for age in ages` is the expression for each item in the set.

Similar to list and dictionary comprehensions, we can also include a condition in the set comprehension:

Input

```
# {expression for item in iterable if condition}
my_set = {age for age in ages if age > 10}
print(my_set)
```

Output

```
{12, 15, 44, 56, 78}
```

This will include only the ages that are greater than `10` in the set.

In summary, set comprehensions provide a concise way to create sets in Python. They can be especially useful when you need to remove duplicates from a list or other iterable, because sets automatically discard duplicate values.

7.7 5.6. Generator Compreshensions

Generator comprehensions are an elegant way to create generators using a syntax that is similar to list comprehensions. In fact, you can convert a list comprehension into a generator comprehension just by replacing the square brackets `[]` with parentheses `()`.

Generators are a powerful feature in Python for creating iterable objects. They are a key component of Python's approach to handling large data streams or sequences of data because they enable you to create an iterable object without needing to store all of the values in memory at once. This can provide substantial performance benefits when dealing with large data sets.

We will cover more details in the next Chapter.

7.7.1 5.6.1. Traditional Generator Function

The gen_func function is a generator function that takes a list of ages as an argument. Inside this function, we iterate over the ages list using a for loop. For each age in ages, we yield the value of age incremented by 1. The yield keyword is used in Python generator functions as a sort of "return" that does not end the function, but instead provides a value and pauses the function's execution until the next value is requested.

Input

```
ages = [5, 12, 3, 56, 24, 78, 1, 15, 44]

def gen_func(ages):
    for age in ages:
        yield age+1
```

```
my_gen = gen_func(ages)
for item in my_gen:
    print(item)
```

Output

```
6
13
4
57
25
79
2
16
45
```

The `gen_func` function is a generator function that takes a list of `ages` as an argument. Inside this function, we iterate over the `ages` list using a `for` loop. For each `age` in `ages`, we `yield` the value of `age` incremented by 1. The `yield` keyword is used in Python generator functions as a sort of "return" that does not end the function, but instead provides a value and pauses the function's execution until the next value is requested.

We call our generator function `gen_func` with the `ages` list as the argument, and the result (a generator object) is assigned to `my_gen`. Then we use a `for` loop to iterate over the generator object, which causes the generator function to execute and `yield` its values one by one. Each yielded value is printed out.

7.7.2 5.6.2. Generator Comprehension

The second part of the code shows a generator comprehension, which is a more compact way of creating a generator. The syntax is very similar to list comprehensions, but uses parentheses `()` instead of square brackets `[]`.

Input

```
# (expression for item in iterable)
my_gen = (age for age in ages)
print(my_gen)

for item in my_gen:
    print(item)
```

Output

```
<generator object <genexpr> at 0x0000020F6F6F0A48>
5
12
3
...
```

This line creates a generator object that will generate the same values as the `ages` list, but on-demand, not storing the entire list in memory.

Generators, both through traditional functions and comprehensions, are a powerful tool in Python. They provide an efficient way to work with large data sets or streams of data that would be inefficient or impractical to store in memory all at once.

8. 6. Iterable, Iterator, Generator

In Python, Iterables and Iterators might seem similar, but they serve distinct purposes and have unique characteristics.

An **Iterable** is an object that can be looped over. This means you can go through its elements one by one, typically using a for loop. Lists, tuples, strings, and dictionaries are all examples of Iterables. They need to implement an `__iter__()` method that returns an Iterator or a `__getitem__()` method for indexed access.

An **Iterator**, on the other hand, is an object that keeps track of its current state during iteration. It must have a `__next__()` method, which returns the next item in the sequence and moves forward, and an `__iter__()` method that returns self. One key aspect of Iterators is that they can only move forward; you can't get an item that has already been iterated over unless you create a new Iterator.

So, the primary difference is that while both can be iterated over, an Iterator also keeps track of its current position in the Iterable. While all **Iterators** are **Iterables** (because they implement an `__iter__()` method), not all **Iterables** are **Iterators** (because they do not provide a `__next__()` method). The power of Iterators comes from their ability to provide items one at a time rather than storing all items in memory at once, which is especially useful when working with large data sets.

A **generator** is a specific type of iterator, which allows us to implement an iterator in a clear and concise way. It's a special kind of function that returns an iterator which we can iterate over to yield sequence of values.

The main difference between a function and a generator in Python is the presence of the yield keyword. While a return statement completely finishes a function execution, yield produces a value and suspends the function's execution. The function can then be resumed from where it left off, allowing the function to produce a series of values over time, rather than computing them all at once and sending them back.

8.1 6.1. Iterable

Data types are **Iterables** in Python:

- Lists
- Tuples
- Strings
- Dictionaries
- Sets
- Generators

In Python, an **Iterable** is an object that can be iterated (looped) over. Essentially, if an object has an `__iter__()` method, it is an Iterable.

You can use `dir()` to check if an object is iterable

Here's an example:

Input

```
pets = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
for pet in pets:
    print(pet)
```

Output

```
Tub
Furrytail
Cat
Barkalot
```

In this case, `pets` is a list, and lists in Python are **Iterables**. We can loop over the list using a `for` loop, and it will print each pet's name one at a time.

8.1.1 6.1.1. Checking if an Object is Iterable

You can check whether an object is iterable by using Python's built-in `dir()` function. If `__iter__` appears in the output, the object is iterable.

Input

```
print('__iter__' in dir(pets))  # Will output: True
```

Output

```
True
```

8.2 6.2. Iterator

An Iterator, on the other hand, is an object with a state that remembers where it is during iteration. While all Iterator objects are Iterable, not all Iterables are Iterators.

An Iterator object is initialized using the `iter()` method, and the `next()` method is used for iteration. Important to note, an Iterator can only move forward; it cannot go backward.

Let's turn our list of pets into an Iterator:

Input

```
pets = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
print(next(pets))
```

Output

```
Output: TypeError: 'list' object is not an iterator
```

Here we get an error because `pets` is a list, and lists are not Iterators. We can, however, turn it into an Iterator using the `iter()` method:

Input

```
iterator_obj = pets.__iter__()
print(iterator_obj)
```

Output

```
<list_iterator object at 0x000001E0F9F4B4C0>
```

Or a better way to do it is:

Input

```
iterator_obj = iter(pets)
print(iterator_obj)
```

Output

```
<list_iterator object at 0x000001E0F9F4B4C0>
```

Now, `iterator_obj` is an **Iterator**. You can't use `next()` directly on the list pets (you would get a `TypeError`), but you can on `iterator_obj`. Let's print the names one by one:

Input

```
print(next(iterator_obj))
print(next(iterator_obj))
print(next(iterator_obj))
print(next(iterator_obj))
print(next(iterator_obj)) # This will raise StopIteration error
```

Output

```
Tub
Furrytail
Cat
Barkalot
StopIteration
```

Calling `next()` again would result in a StopIteration error, as there are no more items to iterate over. This is how Python signals the end of an Iterator.

8.3 6.2.1. Handling StopIteration

We can handle the `StopIteration` error elegantly using a try/except block. Here's how to loop over all the elements of an **Iterator**, stopping cleanly when there are no more items:

Input

```
pets = ['Tub', 'Furrytail', 'Cat', 'Barkalot']
iterator_obj = iter(pets)
while True:
    try:
        next_obj = next(iterator_obj)
        print(next_obj)
    except StopIteration:
        break
```

Output

```
Tub
Furrytail
Cat
Barkalot
```

This will print out each pet's name, just like the `for` loop did earlier, but this time we're using the Iterator's `next()` method.

When we reach the end of the Iterator and a `StopIteration` exception is thrown, our `except` clause catches it and we `break` out of the loop, preventing any errors.

NOTE: Here is one example of how to create a Iterator class, we will cover more details in Chapter 7

Creating a Custom Iterator

Let's look at an example: the `NumIter` class, which is a simple iterator that counts numbers within a given range.

Input

```
class NumIter:
def __init__(self, start, end):
    self.start = start
    self.end = end
    self.current = start

def __iter__(self):
    return self

def __next__(self):
    if self.current >= self.end:
        raise StopIteration
    else:
        return_val = self.current
        self.current += 1
        return return_val
```

Here's what's happening in this code:

1. The `__init__` method initializes the iterator. It takes a `start` and an `end` as arguments, which determine the range of numbers that the iterator will cover. self.current is used to keep track of the current number in the sequence.
2. The `__iter__` method is what makes this class an Iterable. It returns `self`, indicating that an instance of the class is its own iterator.
3. The `__next__` method is the heart of an Iterator. It returns the next value in the sequence each time it's called, and increments `self.current` to prepare for the next call. When there are no more numbers left in the sequence, it raises the `StopIteration` exception, signalling that all values have been returned.

Now, let's see how we can use this NumIter class:

Input

```
nums = NumIter(1, 5)
for num in nums:
    print(num)
```

Output

```
1
2
3
4
```

In this code, we create an instance of `1` that starts at `1` and ends at `5`. We then use a `1` loop to iterate over `1`. This will output the numbers `1` to `4`, one per line.

Note that after you've iterated over an **Iterator**, it's "`exhausted`" and you can't iterate over it again. So if you try to use the `1` loop with 1 again, it won't print anything.

However, if we create a new instance, we can manually iterate over it using the `next()` function:

Input

```
nums = NumIter(1, 5)
print(next(nums))
print(next(nums))
print(next(nums))
print(next(nums))
print(next(nums))
```

Output

```
1
2
3
4
StopIteration
```

This allows us to manually control when we want the next value, but remember that if you try to get the next value after the iterator is exhausted, it will raise a StopIteration exception.

8.4 6.3. Generator

A **generator** is a special type of iterator in Python. Generators don't store all their values in memory, but generate them on the fly. This makes them more memory efficient, especially when dealing with large data sets. It doesn't need **iter**() and **next**() methods

8.4.1 6.3.1. Creating a Generator

A generator function looks very much like a regular function, but instead of returning a value, it yields it. Here's an example:

Input

```
def iter_nums(start, end):
    current = start
    while current < end:
        yield current
        current +=1
```

In this example, `iter_nums` is a generator function. It takes a `start` and an `end` argument and yields numbers from `start` up to, but not including, `end`. We can iterate over this generator using the `next()` function:

Input

```
nums = iter_nums(1, 5)
print(next(nums))
print(next(nums))
print(next(nums))
```

Output

```
1
2
3
```

Or we can use a `for` loop:

Input

```
nums = iter_nums(1, 5)
for num in nums:
    print(num)
```

output

```
1
2
3
4
```

This will output numbers 1 through 4.

Generator function is more readable than iterator class we created previously.

Another example

Input

```
numbers = [1, 2, 3, 4, 5]

# Now we have a list of numbers, and we want to double each number in the list
def double_nums(nums):
    output = []
    for num in nums:
        output.append(num*2)
    return output

output_nums = double_nums(numbers)
print(output_nums)
```

Output

```
[2, 4, 6, 8, 10]
```

We can easily turn this function into a generator by replacing the append and return statements with a yield statement:

Input

```
numbers = [1, 2, 3, 4, 5]
def double_nums(nums):
    for num in nums:
        yield num*2

output_nums = double_nums(numbers)
print(output_nums)
```

As the generator doesn't store the values in memory You will get the generator object:

Output

```
<generator object double_nums at 0x000001E0F9F4B040>
```

To get the values, you can use `next()` :

Input

```
print(next(output_nums))
```

Output

```
2
```

Or output as a list:

Input

```
print(list(output_nums))
```

Output

```
[2, 4, 6, 8, 10]
```

Of course you can also convert it to the list comprehension as we mentioned in the Chapter 5:

Input

```
numbers = [1, 2, 3, 4, 5]
output_nums_list = [num*2 for num in numbers]
print(output_nums_list)
```

Output

```
[2, 4, 6, 8, 10]
```

8.4.2 6.3.2. Generator Expressions

If you replace the square brackets with the parentheses, you will get the generator object:

Input

```
output_nums_generator = (num*2 for num in numbers)
print(list(output_nums_generator))
```

Output

```
[2, 4, 6, 8, 10]
```

In this case, `(num * 2 for num in numbers)` is a generator expression that generates doubled numbers.

8.4.3 6.3.3. Generator vs Iterator

In Python, both iterator and generator can be used to iterate over a sentence, word by word. They provide a convenient way to process each word individually. Let's see how we can create an iterator and a generator to do this.

Creating a Sentence Iterator

First, let's create an iterator. The iterator class, SentIter, will split the sentence into words and yield each word one by one:

Input

```
class SentIter:
    def __init__(self, sentence):
        self.sentence = sentence
        self.index = 0
        self.words = sentence.split()

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= len(self.words):
            raise StopIteration
        else:
            return_val = self.words[self.index]
            self.index += 1
            return return_val
```

Here's how to use this iterator:

Input

```
corpus = "Tub likes Blue Cheese"
words = SentIter(corpus)
for word in words:
    print(word)
```

Output

```
Tub
likes
Blue
Cheese
```

This code will output each word of the sentence on a new line.

Creating a Sentence Generator

Now, let's see how we can do the same thing with a generator. We'll write a function, `iter_sent`, that takes a sentence, splits it into words, and yields each word:

Input

```
def iter_sent(sentence):
    for word in sentence.split():
        yield word
```

And here's how to use this generator:

Input

```
words = iter_sent(corpus)
for word in words:
    print(word)
```

Output

```
Tub
likes
Blue
Cheese
```

As you can see, the generator function is shorter and simpler than the iterator class. This is one of the reasons why generators are often preferred over iterators in Python.

8.4.4 6.3.4. Generator Performance

In Python, a generator is a simpler and more memory-efficient alternative to a list, especially when the list is large. To demonstrate the difference, let's compare the memory usage and execution time of a list and a generator.

Firstly, let's import the necessary modules. We'll use `random` for generating random data, `time` for measuring execution time, `psutil` and `os` for measuring memory usage:

Installation of psutil via `conda` :

```
conda install -c conda-forge psutil
```

Performance Comparison: List vs Generator in Python

I adapte the code with minor updates from Corey to compare the performance of list and generator

Dependencies

```
import random
import time
import psutil
import os
import sys
```

Then, let's define a function to measure the current memory usage:

Input

```
def memory_usage_psutil():
    process = psutil.Process(os.getpid())
    mem = process.memory_info().rss / float(2 ** 20)
    return mem
```

Next, let's prepare some dummy data for our test:

Dummy Data

```
names = ['John', 'Corey', 'Adam', 'Steve', 'Rick', 'Thomas']
majors = ['Math', 'Engineering', 'CompSci', 'Arts', 'Business']
```

Before generating the data, we'll check and print the memory usage:

Memory Usage (Before)

```
print('Memory (Before): {}Mb'.format(memory_usage_psutil()))
```

Now let's define a function to generate a list of people:

Functions for List

```
def people_list(num_people):
    result = []
    for i in range(num_people):
        person = {
                    'id': i,
                    'name': random.choice(names),
                    'major': random.choice(majors)
                }
        result.append(person)
    return result
```

And a function to generate the same data as a generator:

Function for Generator

```
def people_generator(num_people):
    for i in range(num_people):
        person = {
                    'id': i,
                    'name': random.choice(names),
                    'major': random.choice(majors)
                }
        yield person
```

Now let's measure the memory usage and execution time of generating the data using the list and the generator:

List - Memory Usage

```
# Use people_list for comparison
t1 = time.process_time()
people = people_list(1000000)
t2 = time.process_time()

print('Memory (After) : {}Mb'.format(memory_usage_psutil()))
print('Took {} Seconds'.format(t2-t1))
```

Output

```
Memory (Before): 41.6484375Mb
Memory (After) : 265.08984375Mb
Took 0.59375 Seconds
```

You can see the memory usage increased by `223.44140625Mb` after generating the list of people. Now let's see how the generator performs:

Generator - Memory Usage

```
# Use people_generator for comparison
t1 = time.process_time()
people = people_generator(1000000)
t2 = time.process_time()

print('Memory (After) : {}Mb'.format(memory_usage_psutil()))
print('Took {} Seconds'.format(t2-t1))
```

Output

```
Memory (Before): 41.60546875Mb
Memory (After) : 41.61328125Mb
Took 0.0 Seconds
```

As you can see, the memory usage increased by only `0.0078125Mb` after generating the generator object. This is because the generator doesn't store the data in memory. Instead, it yields each person one by one. This is why the memory usage of the generator is much lower than that of the list.

9. 7. Object-oriented programming

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes), and code, in the form of procedures (often known as methods).

In Python, classes are used to create objects (instances), and each object can have attributes and behaviors. Let's dive into it with a simple example.

9.1 7.1. Classes and Instances

A Class is like an object constructor, or a "blueprint" for creating objects. In Python, we define a class using the class keyword.

Here when we say data and functions, we mean attributes and methods. The method here is associated with one class.

Let's create a simple class called `PetEmployee`.

Input

```
class PetEmployee:
    pass
```

Python has a keyword pass that is used as a placeholder. It is syntactically needed for the code to be valid Python, but doesn't actually do anything. In this case, we're using it because we're declaring a class but don't want to put anything inside it yet.

The `PetEmployee` class doesn't currently have any attributes or methods. But it's still a valid class, and we can still create instances of it:

Input

```
barkalot = PetEmployee()
furrytail = PetEmployee()
print(barkalot)
print(furrytail)
```

Here, barkalot and furrytail are instances of the PetEmployee class. When we print them, we'll see the memory address where these objects are stored in wer machine's memory:

Output

```
<__main__.Pet object at 0x0000020E4F6F4E80>
<__main__.Pet object at 0x0000020E4F6F4F10>
```

The output `<__main__.Pet object at 0x0000020E4F6F4E80>` is telling we that `barkalot` is an object of type `PetEmployee`, and it's at the memory location `0x0000020E4F6F4E80`. The specific memory address we see will be different every time we run the program.

These objects don't have any attributes or methods yet, but since they're different instances, they're not identical – they exist independently of each other in different parts of memory.

9.1.1 7.1.1. Attributes and **init** method

The initial snippet shows us how we can add attributes to instances of a class. Here, we're manually adding attributes such as `name`, `age`, `species`, `email`, and `level` to the instances barkalot and furrytail of the PetEmployee class. These attributes are just variables that are associated with each instance of the class.

Input

```
barkalot.name = "Barkalot"
barkalot.age = 3
barkalot.species = "Dog"
barkalot.email = 'barkalot.dog@gmail.com'
barkalot.level = 5

furrytail.name = "Furrytail"
furrytail.age = 2
furrytail.species = "Cat"
furrytail.email = 'furrytail.cat@gmail.com'
furrytail.level = 11
```

```
print(barkalot.name)
print(furrytail.name)
```

Output

```
Barkalot
Furrytail
```

However, this is not the most efficient way to set up our class. It's manual, repetitive, and prone to error (we might forget to initialize an attribute, or make a typo in the attribute name).

`__init__` method

The `__init__` method in Python is similar to constructors in other programming languages. It gets called when we create a new instance of a class. we can use it to set up attributes that every instance of the class should have when it gets created.

Input

```
class PetEmployee:
    # Of course, we can use other names instead of self. But it is a convention to use self.
    def __init__(self, name, age, species, level):
        self.name = name
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level
```

Here, `self` represents the instance of the class. By using the `self` keyword we can access the attributes and methods of the class in python.

When we create new `PetEmployee` instances, we now pass in the initial values for `name`, `age`, `species`, and `level`:

Input

```
barkalot = PetEmployee('Barkalot', 3, 'Dog', 5)
furrytail = PetEmployee('Furrytail', 2, 'Cat', 11)
```

We can then access these attributes using dot notation:

Input

```
print(barkalot.name)
print(furrytail.name)
```

Output

```
Barkalot
Furrytail
```

This will print the names of `barkalot` and `furrytail`. The key thing to note here is that the name attribute for `barkalot` and `furrytail` are separate - changing the name attribute for `barkalot` won't affect `furrytail`'s name attribute, and vice versa.

9.1.2 7.1.2. Methods

A method is simply a function that is associated with an object. In the context of classes, methods often operate on data attributes of the class instances.

The first few lines show how to manually concatenate the `name` and `species` of a `PetEmployee` instance:

Input

```
print('{} {}'.format(barkalot.name, barkalot.species))
```

Output

```
Barkalot Dog
```

This works, but it would be more elegant and maintainable to define a method within the `PetEmployee` class to do this for us:

Class

```python
class PetEmployee:
    # Of course, we can use other names instead of self. But it is a convention to use self.
    def __init__(self, name, age, species, level):
        self.name = name
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level

    def fullname(self):
        return '{} {}'.format(self.name, self.species)
```

This `fullname` method returns a string that is a concatenation of the `name` and `species` attributes of a `PetEmployee` instance. To call this method, we would use the following syntax:

Input

```python
barkalot = PetEmployee('Barkalot', 3, 'Dog', 5)
# we need the parenthesis here because fullname is a method not a attributes as the above
print(barkalot.fullname())
```

Output

```
Barkalot Dog
```

Note the parentheses after `fullname`. This is because `fullname` is a method, not an attribute. If we forget the parentheses, Python will return the method itself, not the result of the method.

we can also call the method on the class, passing the instance as an argument:

Input

```python
print(PetEmployee.fullname(barkalot))
```

Output

```
Barkalot Dog
```

In Python, instance methods need to have `self` as their first parameter so that they can access instance attributes and other instance methods. This is a Python convention. When we call a method on an object, Python automatically passes the object as the first argument. That's why we need to include `self` in the method definition.

Why we need to put self in the method?

If we don't put self in the method, we will get an error.

Input

```python
class PetEmployee:
    # Of course, we can use other names instead of self. But it is a convention to use self.
    def __init__(self, name, age, species, level):
        self.name = name
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level

    def fullname():
        return '{} {}'.format(self.name, self.species)

barkalot = PetEmployee('Barkalot', 3, 'Dog', 5)
print(barkalot.fullname())
```

Output

```
TypeError: fullname() takes 0 positional arguments but 1 was given
```

This is because when we call the method, the instance `barkalot` is passed as the first argument to the method.

9.2 7.2. Class and Instance Variables

Class variables and instance variables are the two types of variables that we can define in a Python class. Both are useful in different scenarios, and understanding them is crucial for effective object-oriented programming in Python.

Instance Variables: Instance variables are associated with instances of the class. This means that for each object or instance of a class, the instance variables are different. Instance variables are defined within methods and are prefixed with the self keyword. They are useful when the value of a variable may differ from one instance of a class to another. For example, in a PetEmployee class, each pet will have a unique name, age, and species, so these would be instance variables.

Class Variables: Class variables are variables that are shared among all instances of a class. They are not defined inside any methods, and they don't have the self prefix. Class variables are useful when we want a variable to be the same for every instance of a class. For example, if we wanted to apply a uniform promotion increment to all PetEmployee instances, we might define a class variable like promotion_increment = 1.

In summary, class variables are shared by all instances of a class, while instance variables can have different values for each class instance. Knowing when to use class variables versus instance variables is essential for creating efficient and organized code in Python.

9.2.1 7.2.1. Instance Variables

Let's add some instance variables to our `PetEmployee` class. We'll add `level` instance variable to the `PetEmployee` class, and we'll set them to the values passed in when the instance is created:

Input

```
class PetEmployee:
    def __init__(self, name, age, species, level):
        self.name = name
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level

    def fullname(self):
        return '{} {}'.format(self.name, self.species)

    def apply_promotion(self):
        # Apply promotion to the level of the pet employee
        self.level = self.level + 1
```

In this code, a class named `PetEmployee` is created, which has data attributes such as `name`, `age`, `species`, `email`, and `level`. It also includes two methods: `fullname` and `apply_promotion`.

The `fullname` method returns a formatted string that concatenates the `name` and `species` attributes of the `PetEmployee` instance, as previously explained.

The `apply_promotion` method increases the `level` attribute of the `PetEmployee` instance by 1. In this context, we might consider `level` as an indication of the employee's rank or position - as the `apply_promotion` method is called, the `level` attribute increases, signifying a promotion.

Here's a walkthrough of what happens when the script is run:

1. Two instances of `PetEmployee` are created, `barkalot` and `furrytail`, with the given attributes.
2. The `level` attribute of the `barkalot` instance is printed out, which shows 3 as per the initial data given at instance creation.
3. The `apply_promotion` method is then called on the `barkalot` instance, which increments `barkalot`'s `level` attribute by 1.
4. Printing `barkalot.level` now shows 4, confirming that the `apply_promotion` method has successfully incremented the `level`.

> **`self`**
>
> Of course, we can use other names instead of `self`. But it is a convention to use `self`.

Input

```
barkalot = PetEmployee('Barkalot', 3, 'Dog', 3)
furrytail = PetEmployee('Furrytail', 2, 'Cat', 5)

print(barkalot.level)
```

```python
barkalot.apply_promotion()
print(barkalot.level)
```

Output

```
3
4
```

9.2.2 7.2.2. Class Variables

What if we want to change the promotion rate? We don't want to change the promotion rate for each instance mannually.

We can use class variable to do this.

Input

```python
class PetEmployee:
    # Class variable
    promotion_rate = 1
    def __init__(self, name, age, species, level):
        self.name = name
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level

    def fullname(self):
        return '{} {}'.format(self.name, self.species)

    def apply_promotion(self):
        # We need to use the class name to access the class variable
        # This can be either self
        # or PetEmployee.promotion_rate
        self.level = self.level + self.promotion_rate
```

In the revised `PetEmployee` class, a class variable `promotion_rate` is introduced. Class variables are variables that are shared across all instances of the class - unlike instance variables, which can have different values for each instance.

In this case, `promotion_rate` determines how much an employee's `level` will increase each time the `apply_promotion` method is called. Since it's a class variable, changing `promotion_rate` will affect all instances of `PetEmployee`, not just one.

The `apply_promotion` method is adjusted to use `self.promotion_rate` when increasing the `level` attribute. The `self` keyword ensures that the instance refers to the class variable, not a potential instance variable of the same name. This way, if `promotion_rate` is changed for the `PetEmployee` class, all instances will use the new rate when `apply_promotion` is called.

In this current setting, calling `apply_promotion` on either `barkalot` or `furrytail` will increment their `level` attribute by 1, as the `promotion_rate` is set to 1.

Input

```python
barkalot = PetEmployee('Barkalot', 3, 'Dog', 3)
furrytail = PetEmployee('Furrytail', 2, 'Cat', 5)
# How to understand this?
# Here we print out the promotion rate of the class and the instance.
# The promotion rate of the instance is the same as the class.
print(PetEmployee.promotion_rate)
print(barkalot.promotion_rate)
print(furrytail.promotion_rate)
```

Output

```
1
1
1
```

The Sequence of Attribute Lookup

When we access the attribute of an instance, it will first check if the instance has the attribute.

If not, it will check if the class has the attribute.

If not, it will check if the parent class has the attribute.

Here, the instance doesn't have the promotion_rate attribute, so it will check the class.

The class has the promotion_rate attribute, so it will use the class attribute.

When we first create the `barkalot` and `furrytail` instances, they don't have an instance variable called `promotion_rate`. So when we try to access `barkalot.promotion_rate` or `furrytail.promotion_rate`, Python doesn't find the attribute in the instance's `__dict__`. In this case, Python falls back to the class (`PetEmployee`) and checks if `PetEmployee` has an attribute `promotion_rate`, which it does.

Now let's lookup the `__dict__` attribute of the class and the instance.

Input

```
print(barkalot.__dict__)
print(PetEmployee.__dict__)
```

Output

```
{'name': 'Barkalot', 'age': 3, 'species': 'Dog', 'email': 'Barkalot.Dog@gmail.com', 'level': 3}
{'__module__': '__main__', 'promotion_rate': 1, '__init__': <function PetEmployee.__init__ at 0x000001B8AD52D940>, ...}
```

The output contains the attribute `promotion_rate` for the class PetEmployee.

When we modify the `promotion_rate` attribute of the `PetEmployee` class, it affects both `barkalot` and `furrytail` because they fall back to the class attribute when their own `promotion_rate` attribute is not found.

Input

```
PetEmployee.promotion_rate = 2
print(PetEmployee.promotion_rate)
print(barkalot.promotion_rate)
print(furrytail.promotion_rate)
```

Output

```
2
2
2
```

However, when we set `barkalot.promotion_rate = 3`, we're creating an instance attribute `promotion_rate` specific to `barkalot`. Now when we try to access `barkalot.promotion_rate`, Python finds it in the `barkalot` instance's `__dict__` and doesn't need to fall back to the class attribute. Therefore, `barkalot.promotion_rate` shows `3`, while `furrytail.promotion_rate` and `PetEmployee.promotion_rate` still show `2`.

Input

```
barkalot.promotion_rate = 3
print(PetEmployee.promotion_rate)
print(barkalot.promotion_rate)
print(furrytail.promotion_rate)
```

Output

```
2
3
2
```

Also check `__dict__` again.

Input

```python
print(barkalot.__dict__)
```

Output

```
{'name': 'Barkalot', 'age': 3, 'species': 'Dog', 'email': 'Barkalot.Dog@gmail.com', 'level': 3, 'promotion_rate': 3}
```

This demonstrates how class variables and instance variables in Python work and the difference between them. Class variables are shared among all instances of a class unless specifically overridden within an instance, as we did with `barkalot`.

Here we introduce an extra usage of class variables: counting the number of instances (objects) created for a class. This is handy if we want to keep track of how many pet employees we've hired so far. I can imagine the HR department being pretty grateful for this feature. (I mean, it would be quite embarrassing if they lost track of how many pets they've hired, right?)

9.2.3 7.2.3. Example

Counting the Number of Instances

Here we want to count the number of employees when we create a new emplyee instance.

Input

```
# Here we want to count the number of employees when we create a new emplyee instance.
class PetEmployee:
    # Class variable
    num_of_pet_employees = 0
    promotion_rate = 1

    def __init__(self, name, age, species, level):
        self.name = name
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level

        PetEmployee.num_of_pet_employees += 1

    def fullname(self):
        return '{} {}'.format(self.name, self.species)

    def apply_promotion(self):
        # We need to use the class name to access the class variable
        # This can be either self or PetEmployee
        self.level = self.level + self.promotion_rate
```

In this iteration of `PetEmployee`, we introduce a new class variable: `num_of_pet_employees`. This variable is incremented each time a new `PetEmployee` instance is created, thanks to the magic line `PetEmployee.num_of_pet_employees += 1` in the `__init__` method. Remember, `__init__` is called each time we instantiate a new object, making it the perfect place to keep count of our newly employed pets.

Here's how it works:

Input

```
print(PetEmployee.num_of_pet_employees)
```

Output

```
0
```

We haven't created any instances of `PetEmployee` yet, so our pet employee count is a big, fat zero. HR is twiddling their thumbs, waiting for some action.

Input

```
barkalot = PetEmployee('Barkalot', 3, 'Dog', 3)
furrytail = PetEmployee('Furrytail', 2, 'Cat', 5)
```

Hold onto your hats, HR, we just employed two new pets! `barkalot` and `furrytail` join the team, and `PetEmployee.num_of_pet_employees` is incremented each time, thanks to our handy `__init__` method.

Input

```
print(PetEmployee.num_of_pet_employees)
```

Output

```
2
```

Voila! HR breathes a sigh of relief. They didn't have to count on their paws - our class variable did the job for them.

This example showcases how a class variable can be used as a handy counter for all instances of a class. It's one of those Python tricks that makes your life as a developer easier and helps you keep track of the state of your program.

9.2.4 7.2.4. Exercise

Let's go one step further and imagine a scenario where HR wants to know which species they've employed most. Here's a little exercise for you: Can you modify our `PetEmployee` class to keep track of how many Dogs and Cats they've hired? (Hint: You might want to use a dictionary as a class variable!)

Which Pet Employee Species Do We Have the Most Of?

Skeleton Solution

Input

```
class PetEmployee:
    num_of_pet_employees = 0
    species_count = {}
    promotion_rate = 1

    def __init__(self, name, age, species, level):
        self.name = name
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level

        PetEmployee.num_of_pet_employees += 1

        # Updating species count
        # Your code here

    def fullname(self):
        return '{} {}'.format(self.name, self.species)

    def apply_promotion(self):
        self.level = self.level + self.promotion_rate
```

Input

```
class PetEmployee:
    num_of_pet_employees = 0
    species_count = {}
    promotion_rate = 1

    def __init__(self, name, age, species, level):
        self.name = name
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level

        PetEmployee.num_of_pet_employees += 1

        # Updating species count
        if species in PetEmployee.species_count:
            PetEmployee.species_count[species] += 1
        else:
            PetEmployee.species_count[species] = 1

    def fullname(self):
        return '{} {}'.format(self.name, self.species)

    def apply_promotion(self):
        self.level = self.level + self.promotion_rate
```

Now, each time a `PetEmployee` is created, `species_count` is updated. Let's create some instances and see how it works:

Input

```
barkalot = PetEmployee('Barkalot', 3, 'Dog', 3)
furrytail = PetEmployee('Furrytail', 2, 'Cat', 5)
mewton = PetEmployee('Mewton', 4, 'Cat', 7)
```

Input

```
print(PetEmployee.species_count)
```

Output

```
{'Dog': 1, 'Cat': 2}
```

9.3 7.3. Classmethods and Staticmethods

Alright, let's dive into the magical world of Python's `classmethods` and `staticmethods`!

In addition to instance methods, which operate on individual objects (or "instances"), Python classes can also have `classmethods` and `staticmethods`.

We'll kick things off by looking at `classmethods`.

9.3.1 7.3.1. Classmethods

To create a class method in Python, we use the `@classmethod` decorator and the special `cls` parameter, which points to the class, not the instance of the object.

In our example code, we have a class `PetEmployee` with a class variable `promotion_rate`. Let's dive into the details:

Input

```
class PetEmployee:
    # Class variable
    promotion_rate = 1

    def __init__(self, name, age, species, level):
        self.name = name
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level

    def fullname(self):
        return '{} {}'.format(self.name, self.species)

    def apply_promotion(self):
        # We need to use the class name to access the class variable
        # This can be either self or PetEmployee
        self.level = self.level + self.promotion_rate

    @classmethod
    def set_promotion_rate(cls, rate):
        cls.promotion_rate = rate
```

Here, we have the `set_promotion_rate` class method. This method changes the `promotion_rate` for all instances of the class, not just for one instance.

So, if we have two pet employees, Barkalot and Furrytail:

Input

```
barkalot = PetEmployee('Barkalot', 3, 'Dog', 3)
furrytail = PetEmployee('Furrytail', 2, 'Cat', 5)
```

And we print their `promotion_rate`, we get `1` for both as the class variable `promotion_rate` is set to `1`:

Input

```
print(PetEmployee.promotion_rate)
print(barkalot.promotion_rate)
print(furrytail.promotion_rate)
```

Output

```
1
1
1
```

But what happens if we change the `promotion_rate` using the `set_promotion_rate` class method?

Input

```
PetEmployee.set_promotion_rate(2)
```

Boom! The promotion rate changes for both Barkalot and Furrytail. That's the power of class methods:

Input

```
print(PetEmployee.promotion_rate)
print(barkalot.promotion_rate)
print(furrytail.promotion_rate)
```

Output

```
2
2
2
```

Classmethods as Alternative Constructors

Classmethods are also commonly used as alternative constructors. This means they can provide additional ways to create objects.

For instance, suppose we have pet employee data as a hyphen-separated string. We can use a class method to parse this string and create a new `PetEmployee` object.

Input

```
@classmethod
def from_string(cls, emp_str):
    name, age, species, level = emp_str.split('-')
    return cls(name, age, species, level)
```

And we can easily create a new `PetEmployee` using this new class method:

Input

```
barkalot_str = 'Barkalot-3-Dog-3'
furrytail_str = 'Furry

barkalot = PetEmployee.from_string(barkalot_str)
furrytail = PetEmployee.from_string(furrytail_str)

print(barkalot.fullname())
```

Output

```
Barkalot Dog
```

With one line of code, we've turned a string into a full-fledged `PetEmployee` object! Who's a good boy? `classmethod`, you're a good boy!

Now, we've tackled class methods like pros. Let's tease apart static methods, shall we?

9.3.2 7.3.2. Staticmethods

Static methods don't access or modify any instance or class data. They're more like handy utility functions we bundle with the class. They're defined using the `@staticmethod` decorator.

Input

```
class PetEmployee:
    # Class variable
    promotion_rate = 1

    def __init__(self, name, age, species, level):
        self.name = name
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level

    def fullname(self):
        return '{} {}'.format(self.name, self.species)

    def apply_promotion(self):
        # We need to use the class name to access the class variable
        # This can be either self or PetEmployee
        self.level = self.level + self.promotion_rate

    @classmethod
    def set_promotion_rate(cls, rate):
        cls.promotion_rate = rate

    @classmethod
    def from_string(cls, emp_str):
        name, age, species, level = emp_str.split('-')
        return cls(name, age, species, level)

    @staticmethod
    def is_walking_pet_today(day):
        if day.weekday() == 6:
            return 'Yaay! It\'s time to walk the pets!'
        return 'Sorry, you have to get back to work!'
```

Let's update our `PetEmployee` class with a static method that checks if `is_walking_pet_today`:

Input

```
@staticmethod
def is_walking_pet_today(day):
    if day.weekday() == 6:
        return 'Yaay! It\'s time to walk the pets!'
    return 'Sorry, you have to get back to work!'
```

Input

```
barkalot = PetEmployee('Barkalot', 3, 'Dog', 3)
furrytail = PetEmployee('Furrytail', 2, 'Cat', 5)
```

This method doesn't rely on any specific instance or class variable, making it a perfect candidate for a static method. It takes in a date and checks if it's a Sunday (weekday `6`). If so, it returns a cheerful message encouraging pet walks. Otherwise, it sadly informs you to get back to work. No fun!

We can call this static method without any instance, just by using the class name:

Input

```
import datetime
today_date = datetime.date.today()
print(PetEmployee.is_walking_pet_today(today_date))
```

Output

```
Yaay! It's time to walk the pets!
```

This block of code imports the `datetime` module, gets today's date, and then checks if it's a pet walking day according to our `PetEmployee` guidelines.

One minor correction I'd like to point out is that the output wouldn't be `False`, but rather one of the two strings our method returns: `'Yaay! It's time to walk the pets!'` or `'Sorry, you have to get back to work!'`, depending on the day of the week.

9.4 7.4. Inheritance

Inheritance allows us to create a new class using details of an existing class without modifying it. This is like saying, "Hey, I like what you've done here. I'll take it, and add a little sprinkle of my own magic."

9.4.1 7.4.1. Creating Subclasses

Initial Class Setup

```
class PetEmployee:
    # Class variable
    promotion_rate = 1

    def __init__(self, name, age, species, level):
        self.name = name
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level

    def fullname(self):
        return '{} {}'.format(self.name, self.species)

    def apply_promotion(self):
        # We need to use the class name to access the class variable
        # This can be either self or PetEmployee
        self.level = self.level + self.promotion_rate
```

In our example, we're creating a new class `PetDataScientist` that inherits from our existing `PetEmployee` class:

Input

```
# Subclass of PetEmployee
class PetDataScientist(PetEmployee):
    pass
```

Here, `pass` is a placeholder because Python expects an indented block for classes. It says, "I don't want to add anything new in this class, just use everything from `PetEmployee`."

So, when we create `PetDataScientist` instances, they have access to the same attributes and methods as `PetEmployee` instances:

Input

```
barkalot = PetDataScientist('Barkalot', 3, 'Dog', 3)
furrytail = PetDataScientist('Furrytail', 2, 'Cat', 5)

print(barkalot.email)
print(furrytail.email)
```

Output

```
Barkalot.Dog@gmail.com
Furrytail.Cat@gmail.com
```

You can see that `barkalot` and `furrytail`, even though they're data scientist pets (probably discussing the latest in machine learning algorithms), have emails formatted the same way as any `PetEmployee`. That's inheritance in action!

One cool feature Python provides is the `help` function. This function displays important details about a class, including its Method Resolution Order (MRO). The MRO is the order in which Python looks for a method in a hierarchy of classes. Here, it tells us that when looking for a method, Python first checks `PetDataScientist`, then `PetEmployee`, and finally the built-in `object` class that every class implicitly inherits from:

Input

```
print(help(PetDataScientist))
```

Output

```
Method resolution order:
 |      PetDataScientist
 |      PetEmployee
 |      builtins.object
```

This is just the tip of the inheritance iceberg, and there's so much more to explore. If you're up for it, why don't we add some unique methods to our `PetDataScientist` class? Maybe a method to analyze data (just pretend data for now) or to present findings? Let your imagination run wild!

Now let's create a functional subclass of `PetEmployee` named `PetDataScientist`:

Child Class - PetDataScientist **Parent Class - PetEmployee**

Input

```
class PetDataScientist(PetEmployee):
    promotion_rate = 2
```

Initial Class Setup

```
class PetEmployee:
    # Class variable
    promotion_rate = 1

    def __init__(self, name, age, species, level):
        self.name = name
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level

    def fullname(self):
        return '{} {}'.format(self.name, self.species)

    def apply_promotion(self):
        # We need to use the class name to access the class variable
        # This can be either self or PetEmployee
        self.level = self.level + self.promotion_rate
```

In this piece of code, we redefine `promotion_rate` for our `PetDataScientist` class, effectively overriding the `promotion_rate` of `PetEmployee` class. You can think of it as saying, "PetEmployee, you did a good job with the promotion rate, but we data scientist pets need it to be a bit faster. So we'll take it from here."

Now when a `PetDataScientist` applies for a promotion:

Input

```
barkalot = PetDataScientist('Barkalot', 3, 'Dog', 3)
print(barkalot.level)
```

```
barkalot.apply_promotion()
print(barkalot.level)
```

Output

```
3
5
```

Barkalot, the data scientist dog (which is honestly the cutest mental image), receives a promotion of 2 levels, unlike regular `PetEmployee` s who only advance by 1. The reason is that when `apply_promotion()` is called, it uses `PetDataScientist` 's `promotion_rate` , not `PetEmployee` 's.

This little example shows the power of inheritance. By changing just one line in the subclass, we've changed the behavior of a method inherited from the superclass without having to rewrite the entire method!

9.4.2 7.4.2. Overriding Methods

Barkalot and Furrytail are stepping up their game! Not only are they data scientists, but they also have their favorite programming languages now. Let's see how you've accomplished this:

Child Class - PetDataScientist **Parent Class - PetEmployee**

Input

```
class PetDataScientist(PetEmployee):
    promotion_rate = 2

    def __init__(self, name, age, species, level, language):
        super().__init__(name, age, species, level)
        self.language = language
```

Initial Class Setup

```
class PetEmployee:
    # Class variable
    promotion_rate = 1

    def __init__(self, name, age, species, level):
        self.name = name
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level

    def fullname(self):
        return '{} {}'.format(self.name, self.species)

    def apply_promotion(self):
        # We need to use the class name to access the class variable
        # This can be either self or PetEmployee
        self.level = self.level + self.promotion_rate
```

In this code, you have overridden the `__init__` method in the `PetDataScientist` subclass. You've added a new parameter `language` to keep track of the favorite programming language of our data scientist pets.

The magic happens in this line: `super().__init__(name, age, species, level)` . The `super()` function is like a time machine that brings us to the parent class, `PetEmployee` in this case. When we call `super().__init__(name, age, species, level)` , it executes the `__init__` method from `PetEmployee` , initializing the common attributes.

Then we come back to the future (or the `PetDataScientist` class) and add the new attribute `language` .

Input

```
ds_barkalot = PetDataScientist('Barkalot', 3, 'Dog', 3, 'Python')
ds_furrytail = PetDataScientist('Furrytail', 2, 'Cat', 5, 'Mojo')
print(ds_barkalot.language)
```

Output

```
'Python'
```

When we create a `PetDataScientist` instance like `ds_barkalot` , we can now provide a programming language. Barkalot prefers Python, just like us!

Inheritance and overriding allow us to extend and modify behavior without disturbing the existing class. Quite neat, isn't it?

One more child class - PetLeader

Let's dive into the marvelous world of team management.

Child Class - PetLeader **Child Class - PetDataScientist** **Parent Class - PetEmployee**

Input

```
class PetLeader(PetEmployee):
    promotion_rate = 1

    def __init__(self, name, age, species, level, team=None):
        super().__init__(name, age, species, level)
        if team is None:
            self.team = []
        else:
            self.team = team

    def add_team_member(self, employee):
        if employee not in self.team:
            self.team.append(employee)

    def remove_team_member(self, employee):
        if employee in self.team:
            self.team.remove(employee)

    def print_team(self):
        for employee in self.team:
            print(' ', employee.fullname())
```

Input

```
class PetDataScientist(PetEmployee):
    promotion_rate = 2

    def __init__(self, name, age, species, level, language):
        super().__init__(name, age, species, level)
        self.language = language
```

Initial Class Setup

```
class PetEmployee:
    # Class variable
    promotion_rate = 1

    def __init__(self, name, age, species, level):
        self.name = name
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level

    def fullname(self):
        return '{} {}'.format(self.name, self.species)

    def apply_promotion(self):
        # We need to use the class name to access the class variable
        # This can be either self or PetEmployee
        self.level = self.level + self.promotion_rate
```

Here, you've introduced a new subclass `PetLeader` that inherits from `PetEmployee`. It includes a new instance variable `team`, which is a list of `PetEmployee` objects. A `PetLeader` has the ability to manage a team, adding and removing team members with the `add_team_member()` and `remove_team_member()` methods, respectively. They can also print their team with the `print_team()` method, showing us the full names of their team members.

Then you've created some instances and made Barkalot a leader:

Input

```
ds_barkalot = PetDataScientist('Barkalot', 3, 'Dog', 3, 'Python')
ds_furrytail = PetDataScientist('Furrytail', 2, 'Cat', 5, 'Mojo')

manager_whiskers = PetLeader('Whiskers', 5, 'Cat', 5, [ds_barkalot])
manager_whiskers.print_team()
```

Output

```
Barkalot Dog
```

Add furrytail to Barkalot's team:

Input

```
manager_barkalot.add_team_member(ds_furrytail)
manager_barkalot.print_team()
```

Output

```
 Barkalot Dog
 Furrytail Cat
```

You've shown us how `isinstance()` and `issubclass()` functions work. `isinstance()` checks if an object is an instance of a class or its subclasses, while `issubclass()` checks if a class is a subclass of another. It's like an identity card for our classes and objects.

Input

```
print(isinstance(manager_whiskers, PetLeader))
print(isinstance(manager_whiskers, PetDataScientist))
```

Output

```
True
False
```

Input

```
print(issubclass(PetLeader, PetEmployee))
print(issubclass(PetDataScientist, PetLeader))
```

Output

```
True
False
```

These tools can be handy when we want to verify the relationships between objects and classes.

Now that we've got a manager, perhaps we could consider a task or project class for the team to work on. What do you think?

9.5 7.5. Polymorphism

Ah, polymorphism! The magical concept in object-oriented programming that allows objects to take on many forms. It's like our pets morphing into different roles in the company at runtime!

First, we need to understand what polymorphism is. It refers to the ability of an object to behave in multiple ways. This comes from Greek, where 'poly' means 'many', and 'morph' means 'form'. In programming, it's the ability of a function or a method to behave differently based on the object that calls it.

Let's use our `PetEmployee`, `PetDataScientist`, and `PetLeader` classes to illustrate the concept.

Input

```
class PetEmployee:
    promotion_rate = 1

    def __init__(self, name, age, species, level):
        self.name = name
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level

    def fullname(self):
        return '{} {}'.format(self.name, self.species)

    def apply_promotion(self):
        self.level = self.level + self.promotion_rate

    def daily_duty(self):
        return "Work! Work! Work!"

class PetDataScientist(PetEmployee):
    promotion_rate = 2

    def __init__(self, name, age, species, level, language):
        super().__init__(name, age, species, level)
```

```
        self.language = language

    def daily_duty(self):
        return "Importing data, analyzing data, and drinking coffee"

class PetLeader(PetEmployee):
    promotion_rate = 1

    def __init__(self, name, age, species, level, team=None):
        super().__init__(name, age, species, level)
        if team is None:
            self.team = []
        else:
            self.team = team

    def daily_duty(self):
        return "Managing team and setting goals"

def pet_daily_duty(pet):
    print(pet.daily_duty())
```

Input

```
emp_barkalot = PetEmployee('Barkalot', 3, 'Dog', 3)
ds_furrytail = PetDataScientist('Furrytail', 2, 'Cat', 5, 'Python')
manager_whiskers = PetLeader('Whiskers', 5, 'Cat', 5)

pets = [emp_barkalot, ds_furrytail, manager_whiskers]

for pet in pets:
    pet_daily_duty(pet)
```

Output

```
Work! Work! Work!
Importing data, analyzing data, and drinking coffee
Lead team and setting goals
```

The `daily_duty()` method has different implementations in the `PetEmployee`, `PetDataScientist`, and `PetLeader` classes. When we call `daily_duty()` on an object, the appropriate method is selected based on the object's class, not the type of the variable that is used to call the method. This is a classic example of polymorphism.

`raise` keyword

In Python, `raise` is a keyword that's used to generate exceptions. By invoking `raise`, you're signaling to Python that an error has occurred, and you're asking Python to stop the normal execution of your program and instead, to "throw" an error that needs to be caught and handled.

Now, let's talk about `NotImplementedError`. This is a special type of exception that we raise when we have a method or function that is supposed to be implemented by a subclass. It's effectively a way of saying, "Hey, if you're seeing this error, it means you've forgotten to implement this method in your subclass."

So when we define a method as follows in the PetEmployee class:

PetEmployee class

```
def daily_duty(self):
    raise NotImplementedError("Implement this abstract method in a subclass")
```

It's like we're putting up a big neon sign saying "Hey, this method needs to be implemented in any subclass that uses it".

The difference between `NotImplementedError` and other types of exceptions is really just about semantics and when they're used. We raise a `NotImplementedError` when we're creating a method that is supposed to be overridden by a subclass.

9.6 7.6. Magic Methods

9.6.1 7.6.1. `__repr__` and `__str__`

We are about to plunge into the wacky world of Magic (or Dunder) Methods in Python. These methods are special functions with double underscores at the start and end of their names (e.g., `__init__`, `__repr__`, `__str__`), hence the nickname "Dunder" (from Double UNDERscore).

Now, let's dissect our code here, which depicts a class `PetEmployee` we created in the previous sections:

Input

```
class PetEmployee:
    promotion_rate = 1

    def __init__(self, name, age, species, level):
        self.name = name
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level

    def fullname(self):
        return '{} {}'.format(self.name, self.species)

    def apply_promotion(self):
        self.level = self.level + self.promotion_rate

    def __repr__(self):
        return "PetEmployee('{}', {}, '{}', {})".format(self.name, self.age, self.species, self.level)

    def __str__(self):
        return '{}, {}'.format(self.fullname(), self.species)
```

In this class, we've implemented two magic methods, `__repr__` and `__str__`. They are used to represent our objects in different ways.

The `__repr__` method returns a string that represents the exact state of the object. This is super useful for debugging and logging, as it provides a complete representation of the object, which we could use to recreate it.

The `__str__` method, on the other hand, is more user-friendly. It returns a string that represents the object in a way that is easy to read. This is what is displayed to the end user.

Let's say we create two `PetEmployee` instances:

```
barkalot = PetEmployee('Barkalot', 3, 'Dog', 3)
furrytail = PetEmployee('Furrytail', 2, 'Cat', 5)
```

Before defining `__repr__` and `__str__`, printing `barkalot` would give something like `<__main__.PetEmployee object at 0x0000020E0F6F6F98>`. Not so informative, right? It's just telling us that `barkalot` is an object of `PetEmployee` class at a specific memory address.

However, after defining these methods:

Input

```
print(repr(barkalot))
print(str(barkalot))
```

The output now becomes much more informative:

Output

```
"PetEmployee('Barkalot', 3, 'Dog', 3)"
"Barkalot Dog"
```

The first one is the `__repr__` output, which provides a complete representation of the `barkalot` object. The second one is the `__str__` output, which is more human-readable and pleasant to the eye. Now we're talking!

Remember, folks, the magic of Dunder methods lies in their ability to let us customize Python class behavior in powerful ways. These methods open the door to a whole new world of possibilities! So go ahead and try using them in your own classes. You'll be amazed at what you can achieve!

9.6.2 7.6.2. `__add__` and `__len__`

Let's go over the code snippet provided:

Input

```
class PetEmployee:
    promotion_rate = 1

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

```
        self.age = age
        self.species = species
        self.email = name + '.' + species + '@gmail.com'
        self.level = level

    def fullname(self):
        return '{} {}'.format(self.name, self.species)

    def apply_promotion(self):
        self.level = self.level + self.promotion_rate

    def __repr__(self):
        return "PetEmployee('{}', {}, '{}', {})".format(self.name, self.age, self.species, self.level)

    def __str__(self):
        return '{}, {}'.format(self.fullname(), self.species)

    def __add__(self, other):
        return self.level + other.level

    def __len__(self):
        return len(self.species)
```

Input

```
barkalot = PetEmployee('Barkalot', 3, 'Dog', 3)
furrytail = PetEmployee('Furrytail', 2, 'Cat', 5)

print(barkalot + furrytail)
print(len(barkalot))
```

Output

```
8
3
```

Now, let's untangle this. We've got two new magic methods on our hands: `__add__` and `__len__`.

The `__add__` method allows us to define the behavior for the addition operator `+`. Here, we've chosen to add the levels of two `PetEmployee` instances together. It's like saying, "Hey, Python! When I add two pet employees together, what I really want is to add their levels."

So, if we were to add `barkalot` and `furrytail`:

```
print(barkalot + furrytail)
# Or print(barkalot.__add__(furrytail))
```

We'd get `8`, because `barkalot`'s level is `3` and `furrytail`'s level is `5`. Quick math, folks!

Similarly, the `__len__` method allows us to define behavior for the `len()` function applied to an instance of our class. Here, it's been defined to return the length of the species name.

So, printing `len(barkalot)`:

```
print(len(barkalot))
# Or print(barkalot.__len__())
```

Would yield `3`, because the species name 'Dog' has three characters.

9.7 7.7. Getters, Setters, and Deleters

We're about to delve into the land of Getters, Setters, and Deleters in Python. Picture this, your pet has attributes, like its name, species, and level. These attributes are like the pet's toys. Your pet can fetch these toys, place them somewhere else, or even destroy them (hopefully, they don't do this often). In the coding world, these actions translate to getting, setting, and deleting attributes!

Let's take a peek at the magic Python has tucked up its sleeve:

Getters **Setters** **Deleters**

Getters are like a fetching command for your pet. They fetch the value of a private attribute. Python, being the friendly language that it is, makes getters easy to use with the `@property` decorator. This allows us to access a method as if it were a simple attribute. Here's how it looks:

```
class Pet:
    def __init__(self, name=None):
        self._name = name

    @property
    def name(self):
        return self._name
```

In this example, `name` is a getter for the private attribute `_name`.

Setters are like telling your pet to place its toy somewhere else. They allow us to set the value of private attributes. We use the `@<attribute>.setter` decorator to create a setter in Python:

```
class Pet:
    def __init__(self, name=None):
        self._name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        self._name = name
```

Here, `@name.setter` allows us to set the value of `_name`.

Deleters are like your pet destroying its toy. They allow us to delete attributes. We use the `@<attribute>.deleter` decorator to create a deleter in Python:

```
class Pet:
    def __init__(self, name=None):
        self._name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        self._name = name

    @name.deleter
    def name(self):
        del self._name
```

The `@name.deleter` allows us to delete `_name` from our instance.

With these magic methods, we can have full control over our class attributes, just like training your pets to handle their toys responsibly. Don't forget to treat your pets, and your code, with care!

Next up, we'll see how these getters, setters, and deleters play together in a single class. Keep your coding boots on; it's going to be a thrilling ride!

9.7.1 7.7.1. Motivation

It's time to introduce getters, setters, and deleters - Python's very own magic carpet ride for navigating the world of object attributes.

First, let's revisit our initial code:

Input

```
class PetEmployee:

    def __init__(self, name, species, level):
```

```
        self.name = name
        self.species = species
        self.email = name + '.' + species + '@gmail.com'

    def fullname(self):
        return '{} {}'.format(self.name, self.species)


barkalot = PetEmployee('Barkalot', 'Dog', 3)

barkalot.name = 'Furrytail'
print(barkalot.name)
print(barkalot.fullname())
print(barkalot.email)
```

Output

```
Furrytail
Furrytail Dog
Barkalot.Dog@gmail.com
```

Now, upon looking at the output, we can see a big woof-woof. We changed `barkalot` 's name to 'Furrytail', and the full name changes as expected. But the email stays the same! It's like calling a cat a dog and expecting it to bark. Now, we could manually update the email every time we change the name, but who wants to do all that extra work? Certainly not us!

9.7.2 7.7.2. Getter

Now we have two ways to fix the above issue:

Define a new instance method **Use a getter @property**

New instance method

```
class PetEmployee:

    def __init__(self, name, species, level):
        self.name = name
        self.species = species

    def email(self):
        return "{}.{}@gmail.com".format(self.name, self.species)

    def fullname(self):
        return '{} {}'.format(self.name, self.species)
```

Input

```
barkalot = PetEmployee('Barkalot', 'Dog', 3)
barkalot.name = 'Furrytail'

print(barkalot.name)
print(barkalot.fullname())
print(barkalot.email()) # We have to change all instances of email to email()
```

Output

```
Furrytail
Furrytail Dog
Furrytail.Dog@gmail.com
```

The issue we faced was that every time we changed the name of our `PetEmployee`, we had to manually update the `email`. So, we turned our email attribute into a method that dynamically generates the email based on the current `name` and `species`. Problem solved, right? Well, not exactly. **Our solution created a new problem: we have to change every instance of `email` to `email()`.** Let's check our the other solution.

Getter @property

```
class PetEmployee:

    def __init__(self, name, species, level):
        self.name = name
        self.species = species

    @property
    def email(self):
        return "{}.{}@gmail.com".format(self.name, self.species)

    def fullname(self):
        return '{} {}'.format(self.name, self.species)
```

Input

```
barkalot = PetEmployee('Barkalot', 'Dog', 3)
barkalot.name = 'Furrytail'

print(barkalot.name)
print(barkalot.fullname())
print(barkalot.email) # You don't have to change anything!
```

Output

```
Furrytail
Furrytail Dog
Furrytail.Dog@gmail.com
```

In this code, we introduced the `@property` decorator before our `email` method. Now we can access it as if it were a simple attribute, no need to write those pesky parentheses. It's just like a self-walking pet; no extra effort required!

We've not only kept the functionality of our first solution (dynamically updating the email), but also made it much more user-friendly. This is what we call a win-win situation in the coding world!

9.7.3 7.7.3. Setter

We're about to dive into the realm of setters. Setters are kind of like giving your pet a new name. You're setting a new value to an attribute.

In Python, we can disguise methods as attributes using the `@property` decorator. But when we want to set a new value to this "attribute", we need a `setter`. A `setter` allows us to define custom behavior for setting values. You might think of it as a strict pet owner who insists on a specific way to feed their pet.

Here's the code for our `PetEmployee` class with a `setter`:

Setter

```
class PetEmployee:
    def __init__(self, name, species, level):
        self.name = name
        self.species = species

    @property
    def email(self):
        return "{}.{}@gmail.com".format(self.name, self.species)

    @property
    def fullname(self):
        return '{} {}'.format(self.name, self.species)

    @fullname.setter
    def fullname(self, name):
        first, last = name.split(' ')
        self.name = first
        self.species = last
```

Let's dissect this piece of beauty:

1. We start by initializing our `PetEmployee` with a `name`, `species`, and `level`.
2. We then create a `@property` for `email`, which takes the `name` and `species` and creates an email-like string. With this, when we call `barkalot.email`, Python calls the `email` method behind the scenes.
3. We do the same for `fullname`, which gives us a concatenated string of `name` and `species`.
4. Then comes the star of our show, the `@fullname.setter` decorator. This turns our `fullname` method into a setter, allowing us to assign a new value to `fullname`. It splits the assigned value into two parts - `first` and `last` - and sets `name` and `species` respectively.

Finally, we test our code:

Input

```
barkalot = PetEmployee('Barkalot', 'Dog', 3)
barkalot.fullname = 'Furrytail Cat'

print(barkalot.name)
print(barkalot.fullname)
print(barkalot.email)
```

Output

```
Furrytail
Furrytail Cat
Furrytail.Cat@gmail.com
```

Our pet Barkalot has now been successfully renamed to Furrytail, a cat, and his email has changed too.

9.7.4 7.7.4. Deleter

The following code is the class `PetEmployee` with a deleter:

Deleter

```
class PetEmployee:
    def __init__(self, name, species, level):
        self.name = name
        self.species = species

    @property
    def email(self):
        return "{}.{}@gmail.com".format(self.name, self.species)
```

```
    @property
    def fullname(self):
        return '{} {}'.format(self.name, self.species)

    @fullname.setter
    def fullname(self, name):
        first, last = name.split(' ')
        self.name = first
        self.species = last

    @fullname.deleter
    def fullname(self):
        print('Delete Pet Name!')
        self.name = None
        self.species = None
```

So, what's going on in here?

1. As before, we initialize our `PetEmployee` with a `name`, `species`, and `level`.
2. Then, we define the `@property` for `email` and `fullname` which return a string representation of email and the full name of the pet respectively.
3. We also have our `@fullname.setter` from earlier which allows us to set a new `name` and `species` for our pet.
4. But here comes the new kid on the block, the `@fullname.deleter`. This piece of magic deletes the `name` and `species` of our pet, effectively sending them into oblivion, and prints a message saying, "Delete Pet Name!".

Let's test it:

Input

```
barkalot = PetEmployee('Barkalot', 'Dog', 3)
del barkalot.fullname

print(barkalot.name)
print(barkalot.fullname)
print(barkalot.email)
```

Output

```
Delete Pet Name!
None
None None
None.None@gmail.com
```

With a wave of our wand (well, the `del` command), we've gone ahead and removed our pet's name. Now, that's a power you'd want to handle carefully!

And just like that, we've completed our trilogy of Python's getters, setters, and deleters! It's like we've just stepped out of a rollercoaster ride of Python object-oriented programming. But worry not, there are plenty more exciting rides in this amusement park.

In our next adventure, how about we look at Python's built-in `property` function and how it can be used instead of the `@property` decorator? Or perhaps, we could delve into how Python's `getattr`, `setattr`, and `delattr` functions work. They provide another way to get, set, or delete attributes of an object.

9.7.5 7.7.5. Built-in property function

You've seen the `@property` decorator in action, now let's see how its sibling `property()` works its magic. Ready? Let's get coding!

In Python, `property()` is a built-in function that creates and returns a property object. A property object has three methods, getter(), setter(), and deleter() that we can use instead of @property and its associated decorators.

Let's put this into context with our beloved `PetEmployee` class. Instead of using `@property`, `@fullname.setter`, and `@fullname.deleter` decorators, we'll use the `property()` function:

Built-in property function

```
class PetEmployee:

    def __init__(self, name, species, level):
        self._name = name
        self._species = species

    def get_fullname(self):
        return '{} {}'.format(self._name, self._species)

    def set_fullname(self, name):
        first, last = name.split(' ')
        self._name = first
```

```
        self._species = last

    def del_fullname(self):
        print('Delete Pet Name!')
        self._name = None
        self._species = None

    fullname = property(get_fullname, set_fullname, del_fullname,
                        "I'm the 'fullname' property.")
```

What just happened? Let's dissect this piece by piece:

1. We're defining our `get_fullname`, `set_fullname`, and `del_fullname` methods as usual. But notice that we're now working with `_name` and `_species`. These are called 'private' attributes, and it's a convention in Python to indicate that these attributes should not be accessed directly. They're meant to be manipulated through methods instead.
2. Finally, the line `fullname = property(get_fullname, set_fullname, del_fullname, "I'm the 'fullname' property.")` creates the `fullname` property. The `property()` function takes four arguments: fget (getter function), fset (setter function), fdel (deleter function), and doc (docstring). We've set all of these for our `fullname` property.

Now let's test our new and shiny `PetEmployee`:

Input

```
barkalot = PetEmployee('Barkalot', 'Dog', 3)

print(barkalot.fullname)  # Output: Barkalot Dog

barkalot.fullname = 'Furrytail Cat'
print(barkalot.fullname)  # Output: Furrytail Cat

del barkalot.fullname  # Output: Delete Pet Name!
```

Output

```
Barkalot Dog
Furrytail Cat
Delete Pet Name!
```

With `property()`, we've gained another tool to effectively encapsulate data in our Python classes.

In our next thrilling episode, we'll be exploring Python's `getattr()`, `setattr()`, and `delattr()` functions. These handy functions allow us to interact with an object's attributes using their string names!

9.7.6 7.7.6. getattr(), setattr(), and delattr()

The `getattr()` function is used to retrieve the value of a named attribute of an object. If not found, it returns the default value provided to the function.

Input

```
class PetEmployee:
    def __init__(self, name, species, level):
        self.name = name
        self.species = species
        self.level = level

barkalot = PetEmployee('Barkalot', 'Dog', 3)

# Using getattr()
print(getattr(barkalot, 'name'))  # Output: Barkalot
```

Output

```
Barkalot
```

The `setattr()` function is used to set the value of a named attribute of an object. If the attribute does not exist, this function creates a new attribute by the given name.

Input

```python
# Using setattr()
setattr(barkalot, 'name', 'Furrytail')
print(barkalot.name)  # Output: Furrytail
```

Output

```
Furrytail
```

The `delattr()` function is used to delete an attribute. If the attribute does not exist, this raises an `AttributeError`.

Input

```python
# Using delattr()
delattr(barkalot, 'name')

# Now trying to access the name attribute will raise an AttributeError
print(barkalot.name)
```

Output

```
AttributeError: 'PetEmployee' object has no attribute 'name'
```

Error Handling

Instead of raising an error, we can also use a `try` / `except` block to handle the error gracefully:

Input

```python
try:
    print(barkalot.name)
except AttributeError:
    print("'PetEmployee' object has no attribute 'name'")
```

Output

```
'PetEmployee' object has no attribute 'name'
```

Seeing "object has no attribute 'name'" is Python's way of telling you that you've crossed a boundary and attempted to access something that just doesn't exist. It's like trying to walk through a door that isn't there. You're just going to run into a wall (or in our case, an error).

These methods can be particularly useful in situations where you want to manipulate attributes dynamically, like in large projects or when working with user-defined inputs.

Exercise

Objective:

Your task is to further enhance the Circle class in Python, making it aware of the unit system used (Metric or Imperial).

Requirements:

The Circle class currently supports a radius in centimeters (cm). However, we also want to accommodate input in inches for our friends who use the Imperial system. Enhance the Circle class to support initializing the radius in either cm or inches.

Extend the radius setter method to convert an input radius in inches to cm before storing it in the _radius attribute. The unit attribute should control whether conversion takes place. If unit is 'inch', convert the input to cm (remember that 1 inch equals 2.54 cm). If unit is 'cm', store the input as is.

Add a new property method, radius_inch, that returns the current radius converted to inches as a sanity check.

Ensure that the area and circumference properties continue to work as expected, returning the area and circumference of the circle in cm² and cm, respectively.

Code Skeleton **Solution**

Input

```
import math

class Circle:
    def __init__(self, radius, unit='cm'):
        self.unit = unit
        self.radius = radius

    @property
    def area(self):
        pass

    @property
    def circumference(self):
        pass

    @property
    def radius(self):
        pass

    @radius.setter
    def radius(self, radius):
        pass

    @property
    def radius_inch(self):
        pass

circle_1 = Circle(3)
print(circle_1.radius)
print(circle_1.radius_inch)

circle_2 = Circle(4, 'inch')
print(circle_2.radius)
print(circle_2.radius_inch)
```

Input

```
import math

class Circle:
    def __init__(self, radius, unit='cm'):
        self.unit = unit
        self.radius = radius  # Radius in specified unit

    @property
    def area(self):
        return math.pi * self._radius**2

    @property
    def circumference(self):
        return 2 * math.pi * self._radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, radius):
        if self.unit == 'inch':
            self._radius = radius * 2.54  # Convert from inch to cm
        else:
            self._radius = radius

    @property
    def radius_inch(self):
        return self._radius / 2.54  # Convert from cm to inch

circle_1 = Circle(3)  # Radius in cm
print(circle_1.radius)  # Output: 3
print(circle_1.radius_inch)  # Output: 1.1811 (3 cm in inches)

circle_2 = Circle(4, 'inch')  # Radius in inches
print(circle_2.radius)  # Output: 10.16 (4 inches in cm)
print(circle_2.radius_inch)  # Output: 4
```

In Python, we use the `@property` decorator to define `getter` methods. A `getter` method lets us access the value of a private attribute. Here, `radius` is a property of the class `Circle`, and the `radius` method gets the value of `_radius`.

The `@radius.setter` decorator defines the setter method for the `radius` property. A setter method allows us to set or modify the value of a private attribute. In our case, the `radius` setter converts the given `radius` to centimeters if the provided unit is in inches.

The `radius_inch` property allows us to convert and get the `radius` from centimeters to inches.

In the code snippet above, we create two instances of the class `Circle`. For `circle_1`, we define the `radius` in centimeters, and for `circle_2`, we define the `radius` in inches. The code then demonstrates how these concepts can be applied to convert and print the `radius` in different units.

10. 8. Error Handling and Exceptions

10.1 8.1. Motivation

Exception handling is a process of responding to the occurrence, during computation, of exceptions – anomalous or exceptional conditions requiring special processing – often changing the normal flow of program execution. It is provided by the software to handle an exception.

Imagine this: you're feeling great, confidently coding away, when suddenly, you hit an error. You see a terrifying red message on your console: `[Errno 2] No such file or directory: 'test_.txt'`. Your code has crashed, and your app came to a screeching halt. This is the nightmare we all strive to avoid.

Here's the offending line of code that brought upon this digital disaster:

Input

```
f = open('test_.txt')
```

Output

```
[Errno 2] No such file or directory: 'test_.txt'
```

Seems innocent enough, right? You're just trying to open a file named `test_.txt`. But there lies the problem: What if the file doesn't exist?

Well, that's exactly what's happened here. The file `test_.txt` doesn't exist in the directory, and Python, ever so literal, freaks out. It raises an error and stops the entire program.

This situation can be quite distressing, especially if your code is running a mission-critical application. What if you're running an app that controls a hospital's life support systems, or an app that provides real-time navigation for drivers? An unexpected crash could have serious consequences.

That's why error handling is so vital in programming. In Python, we have a powerful tool to catch and handle these errors: the `try/except` block.

10.2 8.2. Try/Except Blocks

Our seatbelt in this situation is the `try/except` block:

Input

```
try:
    f = open('test_.txt')  # We are trying to open the file here
    var = not_exist_file  # Oops, this variable doesn't exist!
except Exception:
    print('Sorry. Wrong File Name')  # If an error happens in the try block, we catch it here and print a friendly message.
```

Output

```
Sorry. Wrong File Name
```

When our code hits the `try` block, it attempts to execute the code inside. If everything goes smoothly, it continues to run the rest of the code. But if it hits an error (like trying to access a non-existent file or variable), it jumps straight to the `except` block. The `Exception` keyword catches all types of errors, so no matter what went wrong in the `try` block, the `except` block will handle it gracefully.

In our case, since the file 'test_.txt' doesn't exist and the variable `not_exist_file` is not defined, an error occurs. But rather than crashing our program, it triggers the `except` block, and we see the output: `'Sorry. Wrong File Name'`. The magic of try/except saves our day (or code)!

Keep in mind that handling errors gracefully is an essential part of writing robust, production-ready code. With Python's try/except blocks, you're well-equipped to handle any bumps along the way.

10.3 8.3. Catching Specific Errors

However, our `except` block was a bit too general. It catches all kinds of errors, not just the one we're anticipating (`FileNotFoundError`).

We can specify which error our `except` block should catch. To do this, we simply follow the `except` keyword with the name of the error we're expecting. In our case, `FileNotFoundError`. If this specific error occurs, our `except` block will be executed:

Input

```
try:
    f = open('test.txt')
    var = not_exist_file # this will throw an Nameerror.
except FileNotFoundError:
    print('Sorry. Wrong File Name')
```

Output

```
NameError: name 'not_exist_file' is not defined
```

But wait! There's more! If we run the code now, we're greeted with another error: `NameError: name 'not_exist_file' is not defined`. Our `try` block contains another error that we didn't catch. Remember, Python stops executing the `try` block as soon as it encounters an error.

The plot thickens. Let's add another `except` block to catch this second error. Each `except` block will catch its specified error:

Input

```
try:
    f = open('test.txt')
    var = not_exist_file # this will throw an error.
except FileNotFoundError:
    print('Sorry. Wrong File Name')
except NameError:
    print('Name not found in current scope.')
```

Output

```
Name not found in current scope.
```

Now our program runs without crashing, handling both errors gracefully, and our world is back in balance. So remember folks, always keep your exceptions specific. It's like inviting guests to a party.

So far, we've seen how to catch specific errors using multiple `except` blocks. Now let's learn how to catch the error message itself.

When an error occurs, Python creates an `exception object` that contains specific information about the error. We can grab this object and print its content to get a detailed error message.

To do this, we'll add an `as e` after our exception in the `except` block. The `e` is just a variable name; you could call it anything, but `e` is common. This variable now holds the exception object. When we print `e`, we get the specific error message:

Input

```
try:
    f = open('test.txt')
    var = not_exist_file # this will throw an error.
except FileNotFoundError as e:
    print(e)
except NameError as e:
    print(e)
```

Output

```
name 'not_exist_file' is not defined
```

This block of code will print: `name 'not_exist_file' is not defined` if the `NameError` is encountered. So instead of a vague "Oops, something went wrong", you get a detailed report of the error that occurred.

This technique helps us debug our code by providing more specific information about what went wrong.

10.4 8.4. `else` and `finally` Blocks

Now that we've gotten comfortable with using `try` and `except` to catch and handle exceptions, let's introduce two new blocks that can be used within the `try/except` structure: `else` and `finally`.

The `else` block:

The `else` block in `try/except` is used to specify a block of code to be executed if no exceptions were raised in the `try` block. In other words, if everything in the `try` block goes smoothly, the `else` block runs. It's kind of like saying, "If there are no issues, then let's do this too!"

Here's how we used it in the code you provided:

Input

```
try:
    f = open('test.txt')
except FileNotFoundError as e:
    print(e)
except Exception as e:
    print(e)
else:
    print(f.read())
    f.close()
```

Output

```
test!
```

In this code snippet, we attempt to open a file named 'test.txt'. If the file does not exist, a `FileNotFoundError` is raised and handled. If any other type of exception is raised, it's also caught and handled. If no exceptions are raised (meaning the file opens successfully), the `else` block is executed, and the file is read and then closed.

The `finally` block:

The `finally` block in `try/except` is a place to put any code that MUST execute, whether an exception was raised or not. It's like a safety net that catches any code you absolutely want to run, no matter what happens.

Take a look at how we used it in the code:

Input

```
try:
    f = open('test.txt')
except FileNotFoundError as e:
    print(e)
except Exception as e:
    print(e)
else:
    print(f.read())
    f.close()
finally:
    print('Executing Finally...')
```

Output

```
test!
Executing Finally...
```

After trying to open the file, whether it succeeds or an exception occurs, 'Executing Finally...' is printed. This is because the `finally` block always executes, regardless of whether an exception occurred in the `try` block.

`finally` can be useful in many scenarios. For example, you might use it for cleanup tasks, such as closing files or network connections, regardless of the success or failure of the earlier operations.

With the addition of `else` and `finally`, we now have a lot of control over how our program handles exceptions and ensures certain code always runs.

10.5 8.5. Raising Exceptions

In the above code, you used the `raise` keyword. The `raise` keyword is used to trigger an exception explicitly. We can also pass a custom message or another exception class with the `raise` keyword. Let's take a closer look at how you've used it:

Input

```
try:
    f = open('corrupt_.txt')
    if f.name == 'corrupt.txt':
```

```
        raise Exception
except FileNotFoundError as e:
    print(e)
except Exception as e:
    print('Error!')
else:
    print(f.read())
    f.close()
finally:
    print('Executing Finally...')
```

Output

```
[Errno 2] No such file or directory: 'corrupt_.txt'
Executing Finally...
```

In this case, you're attempting to open a file named `corrupt_.txt`. You then check if the file's name is `corrupt.txt`. If it is, you raise an Exception. If a `FileNotFoundError` occurs (i.e., if the file doesn't exist), it gets caught and you print out the error. If the raised Exception occurs, it also gets caught and you print out 'Error!'.

In your specific case, the file `corrupt_.txt` does not exist. So, the `FileNotFoundError` is raised first and caught by the first `except` block. The error message is printed, and then the `finally` block is executed, printing 'Executing Finally...'.

If the file had existed and its name was `corrupt.txt`, the `Exception` would be raised, 'Error!' would be printed, and the `finally` block would be executed.

Let's say we want to raise an exception with a custom error message when the file name is 'corrupt.txt'. We can do this by passing a string to the `Exception` class, like so:

Input

```
try:
    f = open('corrupt.txt')
    if f.name == 'corrupt.txt':
        raise Exception('This is a corrupt file!')
except FileNotFoundError as e:
    print(e)
except Exception as e:
    print('Error!')
else:
    print(f.read())
    f.close()
finally:
    print('Executing Finally...')
```

Output

```
Error!
Executing Finally...
```

In this scenario, if the file `corrupt.txt` is found and opened successfully, an `Exception` is raised with the message 'Error!'. This exception is then caught by the second `except` block and the custom error message is printed. After that, the `finally` block is executed.

Raising exceptions can be very useful when you want to enforce certain conditions in your code, and halt the execution if these conditions are not met.

That wraps up this guide on Python's error handling and exceptions. By using `try/except` blocks, raising exceptions, and leveraging `else` and `finally` blocks, you can make your code more robust and able to handle unexpected errors more gracefully.