lu

**Python Object Oriented Programming**

Student Workbook

# Python Object Oriented Programming

### Software Development

|  |  |
| --- | --- |
| image001.png | Python Object Oriented Programming |

### Field of Learning

Information Technology, Software Development, Computer Studies

### Aim

To enable students to gain the skills to develop software applications using Object Oriented Programming techniques. Students will enrich their programming and problem-solving skills by applying classes, objects, constructors, methods and properties, inheritance and polymorphism to develop programming applications.

### Learning Outcomes

1. Write and demonstrate programs that solve problems using arrays, input, output, methods, classes, objects, abstract classes, static classes, derived classes and polymorphism
2. Demonstrate effective use of object oriented concepts
3. Demonstrate effective use of integrated development environment features

### Content

* Introduction to high level programming languages and programming environment
* Basic input and output operations
* The use of methods and parameters in solving problems
* Coding standards, debugging and testing
* Solving problems using recursion
* Arrays and algorithms
* Design and Implementation of classes and objects
* Public and private variables
* Constructors, public and private methods
* Modular programming
* Examples of problem solving using classes and objects
* Static classes: examples and its advantages
* Debugging and testing: advanced features of the integrated development environment
* Derived classes and protected variables and methods: examples and problem solving
* Polymorphism and its advantages in software development: examples and problem solving
* Abstract and interface classes
* Using object oriented programming to develop applications
* Object oriented programming and code reuse

### Teaching Learning Methods

Teaching methods will involve theoretical and practical classes which may include but not limited to lectures, class discussions, tutorials, case studies, simulations, computer laboratory work, group activities, face-to-face and online activities.

### Module Outline

**Module text:** Text provided: Python OOP Student Workbook

This module will provide the resource materials, and additional reading and manuals for those who need more resources.

**Software**: *PyCharm* *2017* with *Python 3.5* – an Integrated Development Environment (IDE) used in computer programming, specifically for the Python language. It is developed by the Czech company JetBrains. It provides code analysis, a graphical debugger, an integrated unit tester, integration with version control systems (VCSes), and supports web development with Django.

### Assessments

**Assignment 1:**  20%

**Assignment 2:**  20%

**Assignment 3:** 20%

**Project:** 30%

**Classroom observation and homework:** 10%

### Module Schedule

This is an estimated plan of what we will learn and when we will learn it:

|  |  |
| --- | --- |
| **Lecture** | **Topic** |
| 1 | Introduction and Output |
| 2 | Data Types, Operators and Inputs |
| 3 | Decision Making – IF and Loops |
| 4 | Lists and Dictionaries |
| 5 | Graphical User Interface |
| 6 | Functions and Classes |
| 7 | Classes and Objects |
| 8 | Classes, Objects and Encapsulation |
| 9 | Inheritance |
| 10 | Polymorphism and Abstract Classes |
| 11 | Interfaces and Lists of Objects |
| 12 | Recursion |
| 13 | File Input / Output and Exceptions |
| 14 | Review and Project |
| 15 | Project |
| 16 | Project |

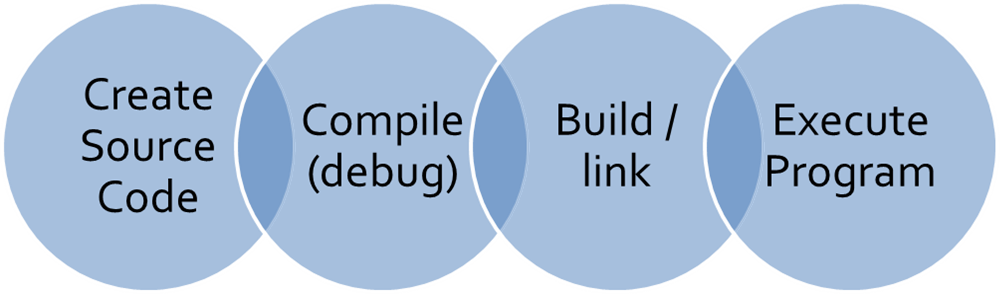
# Section 01: Introduction

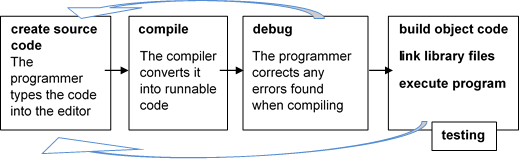
### Introduction to Python OOP Programming

The Python Programming Language

* Python was made in 1991 to make programming easier.
* Python is about the 5th most popular programming language after Java, C, C++ and C#.
* It can be run on all operating systems.
* It is used for web development, data analysis and desktop applications.

The Programming Life Cycle

****



**Bug**

****

PyCharm

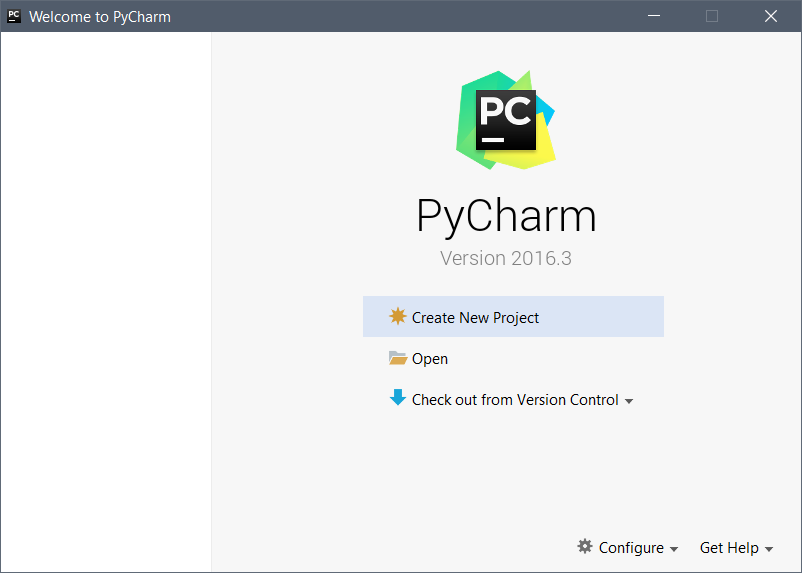
**PyCharm** is an Integrated Development Environment (IDE) used in computer programming, specifically for the Python language. It is developed by the Czech company JetBrains. It provides code analysis, a graphical debugger, an integrated unit tester, integration with version control systems (VCSes), and supports web development with Django.

We will be using *PyCharm* *2016.3* because the lab computers have this version installed.

Opening PyCharm

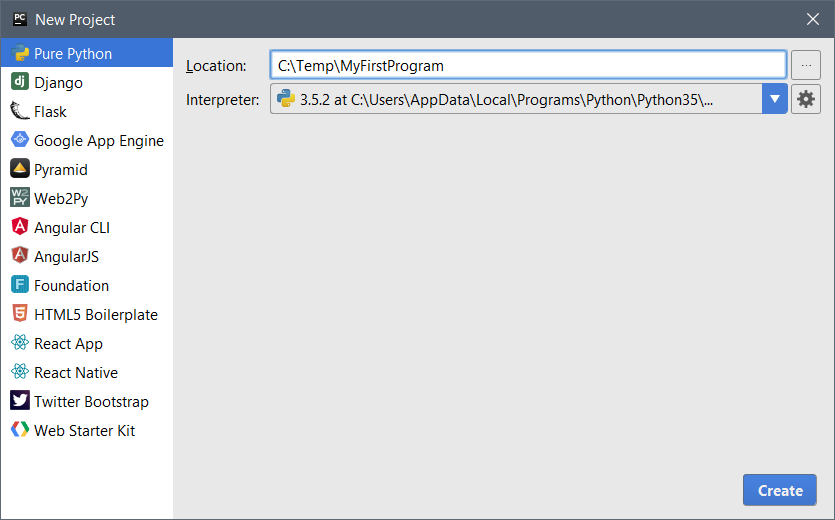
To open the PyCharm:

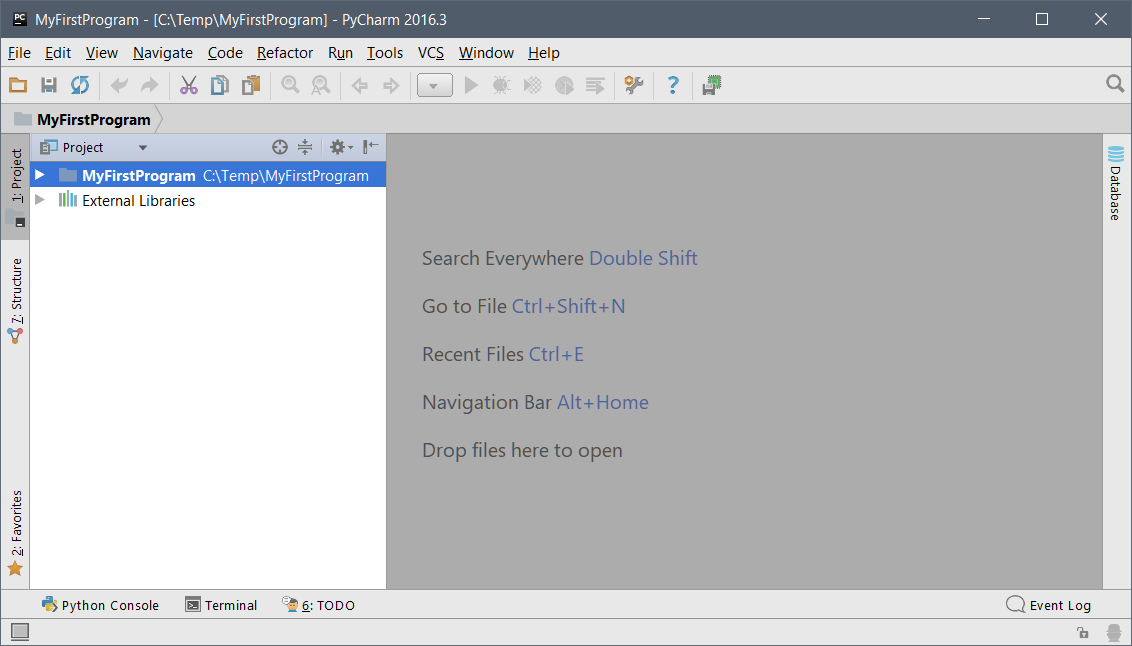
1. Find the shortcut to *JetBrains PyCharm* on the *Desktop* or find it in the *Start* menu.
2. The program will open and show the Start Page:



Creating a new Project

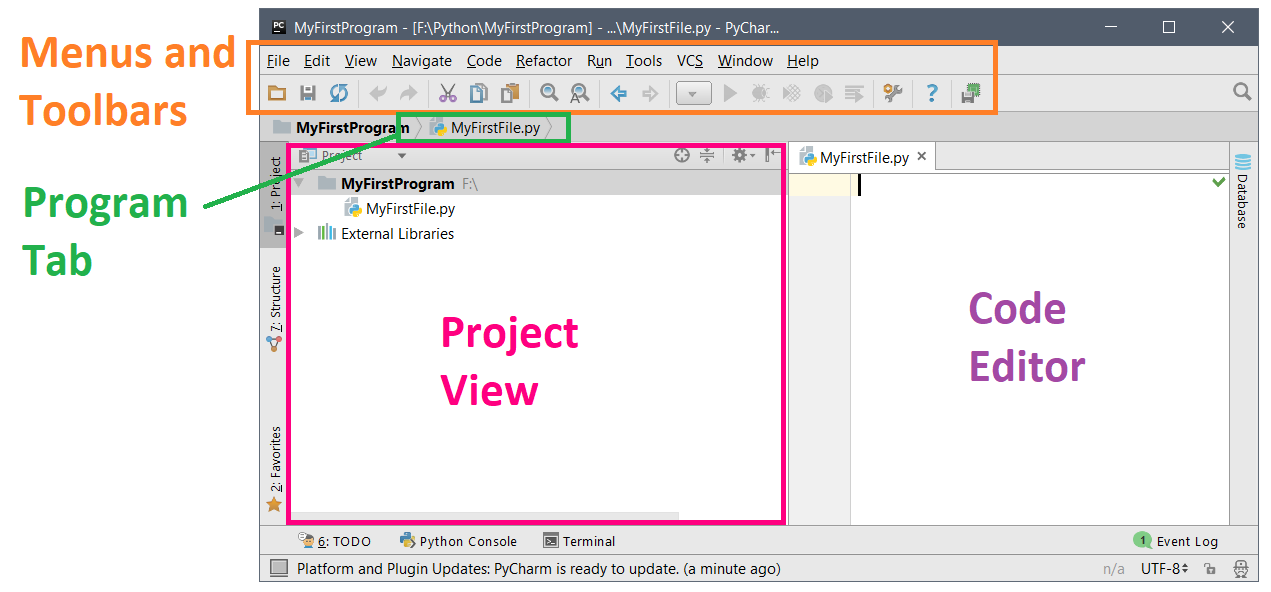
1. Click *Create New Project*
2. Choose a place to save the project.
3. Name your program MyFirstProgram
4. Click *Create*





* Click *File | New | Python File*
* Call it *MyFirstFile*

**





Menu and Toolbars

The most often used buttons are:

* *File | New Project….*
* *File | New…*
* *Run*
* *Debug*
* *File | Close Project*

Program Tab

Every Python file that is opened appears in its own page in the code editor. Clicking on the tab will make the editor display the program code.

Program Code Window / Page

Program code or source code for every Python program.

Project View

The Project View window shows all files in a project. When an application program is created, many files are used to construct, build, and run a program. It also shows the libraries used.

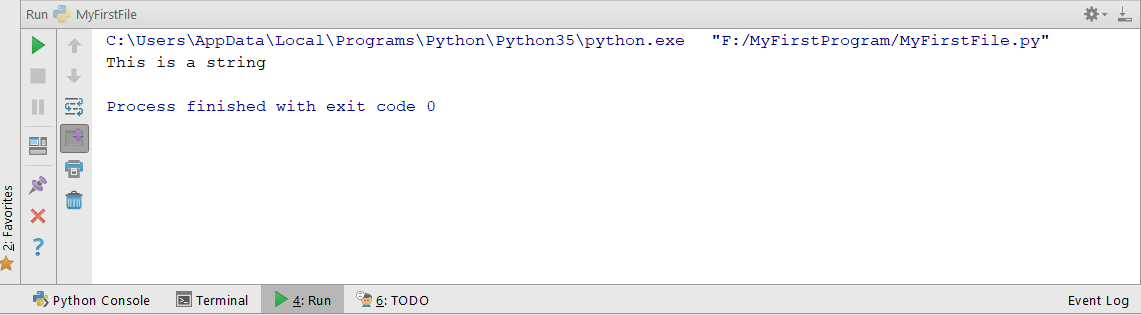
Running a Python Program

When programs are typed into the program editor, the next part is to run the program. Running the program lets the Python editor to do 2 things: check the program for errors and run your program.

* The first time you want to Run the code, you can right click the code and click *Run ‘MyFirstFile’:*



* The second time you want to Run the code, you can click the  button.
* The output will show at the bottom in the *Run Window:*



Closing a Python Program

To close a Python program:

Click *File | Close Project*. Only clicking the close box of the code page does not close the project but instead closes the code page. This means that all other background files linked to the program are still open. If another project is created while the background files from another program remain open, then these files will merge with the new project causing both projects to not run. It is important to close projects correctly to avoid many problems and issues.

Exercise 01-01

* Create a new *Project* called ***MyFirstProgram***and *file* called ***myFirstFile***
* Write this code inside the *Code Window* and *Run* it:

|  |  |
| --- | --- |
| *1*  *2*  *3*  *4*  *5*  *6*  *7*  *8*  *9* | *# This method writes the Happy Birthday song*  **def** happyBirthday():  print(**"Happy Birthday to you!"**)  print(**"Happy Birthday to you!"**)  print(**"Happy Birthday, dear ?"**)  print(**"Happy Birthday to you!"**)  happyBirthday() |

Happy Birthday to you!

Happy Birthday to you!

Happy Birthday, dear ?

Happy Birthday to you!

**Line 1** **Comment** statement

**Line 2** Defines a **function** called happyBirthday()

**Lines 4-7** **Prints** out the words to the Happy Birthday song. The **function** print() makes the words show in the output window.

**Line 9** Calls the **function** happyBirthday()

Extra:

* Make a **variable** called myName and make it equal to your name
* Put the **variable** into line 6, so that it prints “Happy Birthday, dear ” and your name.

Happy Birthday to you!

Happy Birthday to you!

Happy Birthday, dear Sam

Happy Birthday to you!

Indentation

A **block** is a group of code. **Indentation** is making white space in the blocks using TABs in the code. **Indentation** is very important in Python. It tells the program which block a line of code belongs to.

password = **"apple"  
  
if** password == **"apple"**:  
 print(**"Logging on ..."**)  
**else**:  
 print(**"Incorrect password."**)  
  
print(**"All done!"**)

This shows that the print(**"Logging on ..."**) block belongs to the **if** statement, and the print(**"Incorrect password."**) block belongs to the **else** statement.

The print(**"All done!"**) block is **NOT** indented so it does not belong to the **else** statement.

Comments

Comments are very important when writing a program. They help you and other programmers understand what the code does. Comments should be clear, formatted and detailed.

Types of comment statements:

* One-line comments: *# This is a one line comment*
* Multiple line (many line) comments: *""" This is a*

*multiple*

*line comment"""*

Common Errors

* Wrong spelling – capital letters are important
* Wrong indentation – tabs are very important in Python
* Quotes **"** are not correct
* Using = in **if** statements: you must use ==

Exercise 01-02

* Make a new Python project and file
* Make these variables with the values:
  + *my\_int* has the value 7
  + *my\_float* has the value 1.23
  + *my\_bool* has the value True
* Print out *my\_int*
* Copy this code and fix it:

**def** food():  
eggs = 12  
 name = **"chicken'  
return** eggs  
  
 print(Food())

12

* Make a **one line comment** at the top of the program. Write anything you want.
* Below that comment, make a **multiple** **line** **comment**. Write anything you want.

Escape Sequence Tags

|  |  |  |
| --- | --- | --- |
| Description | Syntax | Examples |
| Make a new line | **\n** | print(**"\nHello\nWorld"**)  Displays each word on a new line |
| Horizontal tab | **\t** | print(**"\tHello\tWorld"**)  Puts a tab, then “Hello”, a second tab and then “World”. |
| Carriage return (ENTER key) to put the cursor at the beginning of the current line | **\r** | print(**"Hello\rWorld"**)  Not used often. Output that appears after the carriage return is overwritten. |
| Print a back slash \ | **\\** | print(**"Odd\\Even"**) |
| Print single quotation symbol ' | **\'** | print(**"\'Hello World\'"**)  Shows ‘Hello World’ with single quotes. |
| Print double quotation symbols " | **\"** | print(**"\"Hello World\""**)  Shows “Hello World” with double quotes. |

Output Statements

New Line

* You can writeprint() with nothing inside the round brackets **()**
* You can write **\n** at the end of the previous print statement, for example,   
  print (**"Hello\n"**)

**Horizontal Spaces**

* You can use print(**" "**)
* You can use the tab character: print(**"\t"**)

Formatting Output

First you make a **template** with **placeholders**:

template = **"{0} {1}"**

Then you give the template **arguments** inside the format method:

result = template.format(**"Hello"**, **"friend"**)

Now you can print out the formatted text:

print(result) *# output: Hello friend*

Here is the full code:

template = **"{0} {1}"**

result = template.format(**"Hello"**, **"friend"**)  
print(result) *# output: Hello friend*

Or here is a short way:

print(**"{0} {1}"**.format(**"Hello"**, **"friend"**)) *# output: Hello friend*

You can use < to make the output go to the left, or > to make the output go to the right. The number after the < or > is for the width of the argument.

template = **"{0:<10},{1:>15}"**result = template.format(**"Hello"**, **"friend"**)  
print(result) *# output: Hello , friend*

* **'Hello'** has 10 characters (5 characters for “Hello” and then 5 characters of spaces).
* **'friend'** has 15 characters (first 9 characters of spaces, then 6 characters for “friend”).

Formatting codes

You can format numbers with decimal points like **floats**. A decimal point is written, then how many decimal places to show, then the format of the number:

template = **'{0:.2f}'** *# Format the number as a float with 2 decimal places*

template = **'{0:<10.2f}'***# Format the number as a float with 2 decimal*

*# places and make it on the left with a width of 10*

template = **'{0:<10.2f},{1:<10.3f},{2:>10.3f}'**result = template.format(11.2345, 123.76587, 54.3)  
print(result) *# output: 11.23 ,123.766 , 54.300*

You can use variables with formatting templates. Here we format the output using 2 variable arguments:

name = **"Milk"**price = 2.7843256  
template = **"Name: {0}, Price: {1:5.2f}"**output\_string = template.format(name, price)  
print(output\_string) *# output: Name: Milk, Price: 2.78*

Formatting numbers

You can format numbers with a letter after the colon **:**

|  |  |  |
| --- | --- | --- |
| Letter | How it is printed | Example for 1000 |
| f | As a floating point number | 1000.000000 |
| d | As a decimal integer | 1000 |
| e | As a floating point number with exponential format | 1.000000e+03 |
| , | As a decimal integer with commas | 1,000 |

Example

num1 = 16305.3524  
num2 = 18589.164793  
  
template = **"{0:<28}{1:>14}"**result = template.format(**"Number 1"**, **"Number 2"**)  
print(result)  
  
template2 = **"{0:<28.2f}{1:>14.3f}"**result2 = template2.format(num1, num2)  
print(result2)

Output:

Number 1 Number 2

16305.35 18589.165

Exercise 01-03

Make this output:

My name's "Sam" \ and I am 20 years old

Welcome to Python

Apple Orange

1.69 2.1613

**Welcome** is formatted to the left with a width of 15

**to** is formatted to the right with a width of 5

**Python** is formatted to the right with a width of 10

apple = 1.69234  
orange = 2.1612561

**Apple** is formatted to the left with a width of 12

**Orange** is formatted to the right with a width of 12

**1.69** is formatted to the left with a width of 12

**2.1613** is formatted to the right with a width of 12

Exercise 01-04

Make this output:

"Name" "Score"

Max 77.2

Alice 89.3

Bob 68.0

Using this information:

|  |  |
| --- | --- |
| Name | Score |
| Max | 77.234 |
| Alice | 89.325235 |
| Bob | 68 |

* Use variables like name1, score1, name2, score2 …
* The names are formatted to the left with a width of 10
* The scores are formatted to the left with a width of 5

# Section 02: Data Types, Inputs and Operators

### Data Types

Dynamically Typed Languages

**Dynamically typed** programming languages do type checking at **run time** and not at compile time. You do not need to write the code to tell the program what data type a variable is. Examples of **dynamically typed languages** are Python, JavaScript and PHP.

Statically Typed Languages

**Statically typed** programming languages do type checking at **compile time** and not at run time. You need to write the code to tell the program what data type a variable is. C# is a **statically typed** language.

Python Data Types

There are built in data types in Python.

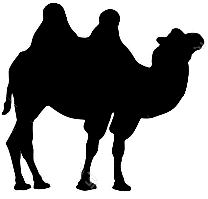
|  |  |  |
| --- | --- | --- |
| Data Type | Syntax | Examples |
| Integer | **int** | 1, 44, -3468 |
| String | **str** | Characters inside ***double*** or ***single*** quotes  Examples: **"Hello"**, **"Tom"**, **'Hamilton'** |
| Boolean | **boolean** | **True or False** |
| Float | **float** | 15.20, 0.0, -21.9, 70.2-E12 |
| List | **list** | [**'a'**, **'b'**, **'John'**, **'z'**, **'example'**] |
| Dictionary | **dictionary** | {**'Name'**: **'Sam'**, **'Age'**: 21, **'Class'**: **'English'**} |

Naming Conventions

* A variable’s name can only contain:
  + Letters a-z, A-Z
  + Underscore \_
  + Digits 0-9
* Cannot start with digits
* Cannot use Python **keywords** like class or int

This is how we should name variables in Python:

**Classes**

* Start with capital (big) letter. For example: *Animal*
* Camel case. Each word has a capital letter. For example, *MyBigAnimal*

**Variables**

* Should be lowercase.
* Words should be separated with underscores \_ . For example: *my\_dog*
* Camel case is sometimes allowed. Each word has a capital letter. For example, *myDog*

**Functions and Methods**

* Should be lowercase.
* Words should be separated with underscores \_ . For example: *my\_function*
* Camel case is sometimes allowed. Each word has a capital letter. For example, *myFunction*

Declaring Variables

In Python, variables are case-sensitive. This means capital (big) letters are important.

**Error!**

Wrong case, and therefore this a different variable than checked\_temperature

num1 = 5  
my\_name = **"Sam"**checked\_temperature = **True**print(Checked\_Temperature)

Exercise 02-01

In some countries, when you pay for a meal, you need to calculate the tax and the tip for the meal:

* Cost of meal: $44.50
* Restaurant tax: 6.75%
* Tip: 15%

Write the code:

* Make a variable called ***meal*** and give it a value of 44.50
* Make a variable called ***tax*** and give it a value of the decimal value of 6.75%
* Make a variable called ***tip*** and give it a value of the decimal value of 15%
* Make ***meal*** equal to ***meal*** plus ***meal*** \* ***tax***
* Make a variable called ***total*** and make it equal to ***meal*** plus ***meal*** \* ***tip***
* Print out the value of ***total*** to 2 decimal places:

Total cost of meal: $54.63

Scope

In Python, the code that is **indented** is inside a **block**. If you make a variable inside a block, it cannot be used or seen outside that block. These variables inside the block are called **Local variables**.

A block makes a **scope**. **Scope** is the part of a program where a local variable can be used and can be seen. It also controls the **lifetime** of a variable.

Example

**def** my\_function():  
 age = 10 print(age)

Data Type Conversion

In Python, you can **convert** or **cast** variable to a different data type.

To change to a different **data type**, just write the new type and put the variable or value in brackets:

x = 2.15315  
y = int(x)  
print(y)

This will print **2**, because the *float* x was changed to an *int*. The data inside the float variable is lost. It was 2.15315 but because we **cast** it to an *int*, the decimal part is lost.

a = 5  
b = float(a)  
print(b)

This will print **5.0**, because the *int* a was changed to a *float*.

Sometimes we need to change a number to a string. We can print out a string together with a number by writing this:

x = 3  
y = 2.1531  
  
print(**"x = "** + str(x))  
print(**"y = "** + str(y))

x = 3

y = 2.1531

You can see that we must **cast** the numbers to strings first, otherwise there will be errors.

We may also need to change strings to numbers:

a = **"135.31421"**b = **"133.1112223"**c = float(a) + float(b)  
print(c)

This will print out **268.4254323** because the strings were changed to floats, and added together.

Here are some of the conversion functions:

|  |  |  |
| --- | --- | --- |
| **Function** | **Description** | **Example** |
| int(x) | Converts x to an integer | int('2014')  2014  int(3.141592)  3 |
| float(x) | Converts x to a floating point number | float('1.99')  1.99  float(5)  5.0 |
| str(x) | Converts x to a string. x can be of the type float, integer or long. | str(3.141592)  '3.141592' |

Exercise 02-02

Add these numbers together and print out the answer:

x = **"23.556"**y = 6.26  
z = 5

Exercise 02-03

Make the output say **Hello, I am 20 years old** using the message variable and the age variable.

message = **"Hello, I am "**age = 20.523

Exercise 02-04

x = 10.0  
y = 3.0  
i = 5  
j = 5  
  
i = int(x + y)  
x = j

1. What is the value of i now?
2. What is the value of x now

Exercise 02-05

Make the code print **Five plus seven equals 12.0** using only the variables. Remember, you can use many **conversion functions** together, for example:

str(float(int(x)))

add1 = **"Five "**num1 = **"5"**plus = **"plus "**add2 = **"seven "**num2 = **"7"**equals = **"equals "**

### Basic Operators in Python

These are the basic operators in Python:

|  |  |  |
| --- | --- | --- |
| Operators | Syntax | Meaning |
| Arithmetic Operators | **+ - \* /** | Addition, Subtraction, Multiplication, Division |
| Relational Operators | **> < <= >=** | Less than, Greater than, Less than or equal to, Greater than or equal to |
| **== !=** | Equal to, Not Equal to |
| Logical Operators | **and** | AND Boolean Logic, true only if both sides are true |
| **or** | OR Boolean Logic, true if one of the two sides is true |
| **not** | NOT Boolean Logic |
| Modulus | **%** | Returns the remainder from a division |
| Assignment Operators | **=**  **+=**  **-=**  **\*=**  **/=** | Simple assignment operator  Add AND assignment operator  Subtract AND assignment operator  Multiply AND assignment operator  Divide AND assignment operator |

Arithmetic and Modulus Operators

If you use an *int* and divide one number by another number, the answer will be a whole number:

7 / 4 = 1

16 / 5 = 3

To get the remainder number you can use the **modulus operator**:

7 % 4 = 3

16 % 5 = 1

8 % 2 = 0

You can use the **modulus operator** to see if a number is **odd** or **even**:

num1 = 5  
  
**if** num1 % 2 == 0:  
 print(**"{0} is even"**.format(num1))  
**else**:  
 print(**"{0} is odd"**.format(num1))

5 is odd

You can use the **modulus operator** to see if a number is a **factor** of another number:

num1 = 15  
num2 = 3  
  
**if** num1 % num2 == 0:  
 print(**"{0} is a factor of {1}"**.format(num2, num1))  
**else**:  
 print(**"{0} is NOT a factor of {1}"**.format(num2, num1))

num2 is a factor of num1

Assignment Operators

If you want to increase a number by a value, you can write it the long way:

num1 = num1 + 5

Or you can write it the short way using an Assignment Operator:

num1 += 5

Example

To decrease a number by 1, you can write:

num2 = 5  
num2 -= 1  
print(num2)

4

Exercise 02-06

num1 = 2  
num2 = 7  
  
print(**"Arithmetic operations"**)  
print(**"Number 1: "** + str(num1) + **" and Number 2: "** + str(num2))  
print()  
  
answer = num1 + num2  
print(**"Addition: "** + str(num1) + **" + "** + str(num2) + **" = "** + str(answer))  
  
<Make answer equal to num1 - num2>  
print(**"Subtraction: "** + str(num1) + **" - "** + str(num2) + **" = "** + str(answer))  
  
answer = num1 \* num2  
<Make the output say "Multiplication: 2 \* 7 = 14" using the variables num1, num2 and answer>

<Increase num1 by 5 using an Assignment Operator>

<Make answer equal to num2 / num1>  
print(**"Division: "** + str(num2) + **" / "** + str(num1) + **" = "** + str(answer))  
  
<Make answer equal to num2 modulus operator num1>  
print(**"Remainder of "** + str(num2) + **" % "** + str(num1) + **" = "** + str(answer))  
  
print(**"Division when using double data types"**)  
num3 = 2.0  
num4 = 7.0  
  
<Make answer2 equal to num4 / num3>  
  
print(**"Division: "** + str(num4) + **" div "** + str(num3) + **" = "** + str(answer2))

<Make answer2 equal to num4 modulus num3>

<Make the output say "Remainder of 7.0 mod 2.0 = 1.0" using the variables num3, num4 and answer2>

### Input Statements

We know about **Output Statements**. We use print() to print out text.

Programs sometimes need **input** from the user. To get input from keyboard, screen or user, Python uses input() with **type casting**.

input() is the input method used to read from the user. **Type casting** is used to **convert** input from one **data type** to the **data type** of the variable.

The input() function **ALWAYS** returns a string when it gets data from the user.

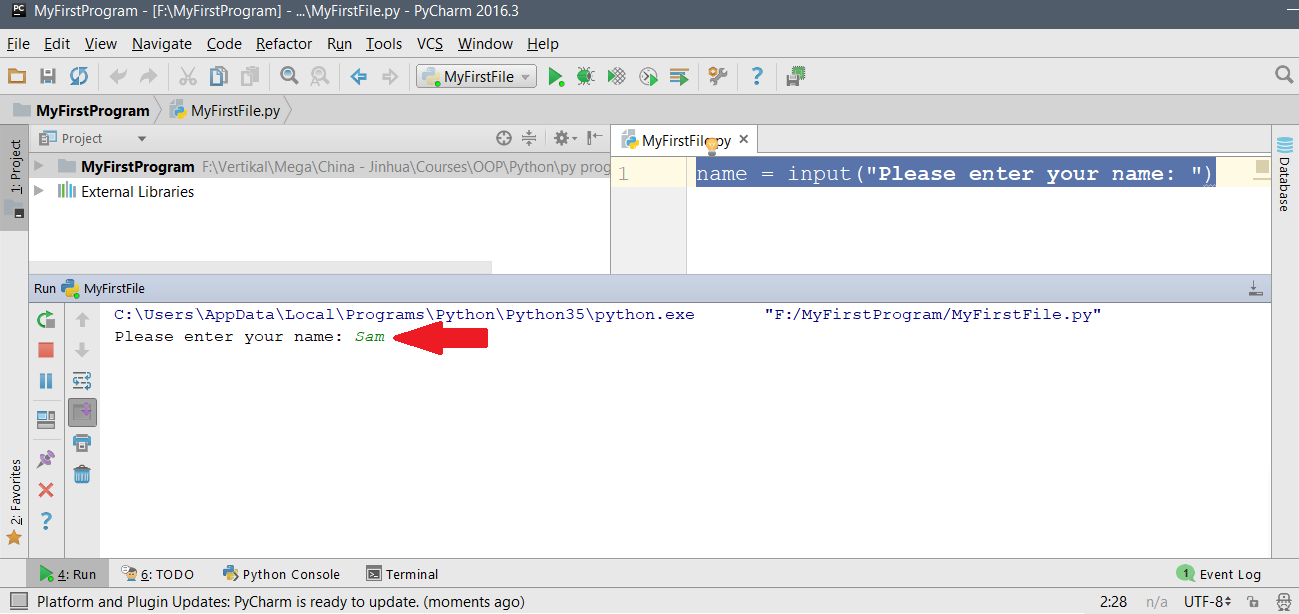
**Input Statement**

<variable> = input( <message to the user> )

Example

name = input(**"Please enter your name: "**)

You can enter the text input at the bottom of PyCharm in the *Run* window:



You can now use the *name* variable:

name = input(**"Please enter your name: "**)  
print(**"Hello: "** + name)

Please enter your name: Sam

Hello: Sam

If we want to get a number as input, it will become a *string*:

num1 = input(**"Please enter a number: "**)  
print(**"Number is: "** + num1)

Please enter a number: 5

Number is: 5

If we try to do calculations on the number, we will get an error. We cannot do a calculation on a string and a number:

num1 = input(**"Please enter a number: "**)  
num1 - 1

TypeError: unsupported operand type(s) for -: 'str' and 'int'

So we need to use both the cast to an int and the input function:

num1 = int(input(**"Please enter a number: "**))  
num2 = num1 - 1  
print(**"Answer = "** + str(num2))

Please enter a number: 5

Answer = 4

Here, we cast the input to a float:

x = float(input(**"Please enter a number: "**))  
y = x - 1  
print(**"Answer = "** + str(y))

Please enter a number: 3.14

Answer = 2.14

Exercise 02-07

* Get an animal as a string from the user and print it out:

Enter an animal: dog

dog

* Get 2 numbers from the user and add them together:

Enter the first number: 5

Enter the second number: 3

Answer = 8

* Get the length and width of a rectangle (as floats) and calculate and print the **area** and the **perimeter** of the rectangle:

Enter the length of the rectangle: 4.5

Enter the width of the rectangle: 3.5

Area = 15.75

Perimeter = 16.0

* Get the radius of a circle (as a float) and calculate and print the area of the circle. Use 3.14 for π:

Enter the radius of the circle: 4.5

Area = 63.585

# Section 03: Decision Making

### IF Statements

**Syntax**

The IF statement is used for **decision-making** and it is called a **control structure**. Control structures decide the flow of a program. Python has this syntax for **IF statements**:

*# IF statement with one statement***if** <Boolean-expression>:  
\_\_\_\_<statement>  
  
*# IF statement with many statements***if** <Boolean-expression>:  
\_\_\_\_<statement>

\_\_\_\_<statement>

\_\_\_\_<statement>

You can see that you need to indent the statements to make them be inside the IF statement.

The code that is indented will be run IF the boolean expression is TRUE.

Example: A program calculates a student’s grade:

If student’s grade is greater than or equal to 80

Output “A”

grade = float(input(**"Please enter the grade: "**))  
  
**if** grade >= 80:  
 print(**"A"**)

Please enter the grade: 84

A

**Exercise 03-01**

print(**"Program to show high and low numbers"**)  
  
*# Get input and calculate number*<Get an int from the user and put it into num1. Make it say "Enter a number">

**if** num1 > 50:  
 print(str(num1) + **" is a high number"**)  
  
<Make an if statement to see if num1 is less than 50>  
 print(str(num1) + **" is a low number"**)  
  
<Make an if statement to see if num1 is equal to 50>  
 <Print out "50 is in the middle" using the num1 variable>

### Relationship Operators

Relationship operators test the relationships between two variables or values in a Boolean expression. The result is either **TRUE** or **FALSE**.

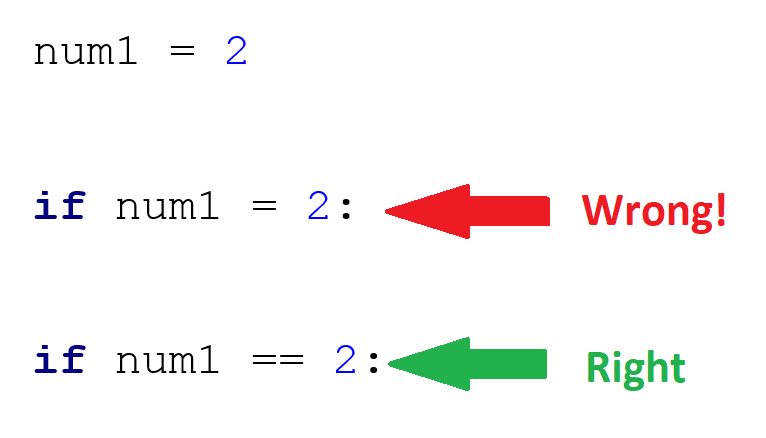
if <variable> <relationship operator> <value or variable> :

<statement>

|  |  |  |
| --- | --- | --- |
| Relationship Operator | Meaning | Explanation |
| a == b | Equal to | Are they equal? Is “a” equal to “b” |
| a != b | Not equal to | Are they **NOT** equal? Is “a” NOT equal to “b” |
| a > b | Greater than | Is the left greater than the right? Is “a” greater than “b” |
| a >= b | Greater than or equal to | Is the left greater than or equal to the right? Is “a” greater than or equal to “b” |
| a < b | Less than | Is the left less than the right? Is “a” less than “b” |
| a <= b | Less than or equal to | Is the left less than or equal to the right? Is “a” less than or equal to “b” |

Common Mistake!

Make sure you have 2 equal signs == if you use an IF Statement:



### IF Statements with Relationship Operators

**Exercise 03-02 OPTIONAL**

1. Test the program with these input numbers and look at the output displayed
2. Test 1

num1 = 10, num2 = 10

1. Test 2

num1 = 80, num2 = 50

1. Test 3

num1 = 110, num2 = 115

1. Test 4

num1 = 64, num2 = 64

*# Introduction messages*print(**"Welcome to programming in Python"**)  
print(**"This program shows If statements"**)  
  
*# Get input from User*num1 = int(input(**"Enter a number between 0 And 100: "**))  
  
<Get an int from the user and put it in num2. Make it say "Enter another number between 0 and 100">  
  
*# Test the numbers using relationship operators***if** num1 == num2:  
 print(**"The numbers are the same"**)  
  
<Make an if statement to see if num1 is not equal to num2>  
 print(**"The numbers are not equal to each other"**)

<Make an if statement to see if num1 is greater than or equal to num2>  
 print(**"Number 1 is greater than or equal to number 2"**)  
  
<Make an if statement to see if num1 is less than or equal to num2>  
 print(**"Number 1 is less than or equal to number 2"**)

### Logical Operators

Logical operators test the relationship in a Boolean expression.

The most common logical operators are **and**, **or** and **not**. Logical operators compare Boolean expressions. Logical operators do not compare values.

You can use round brackets **( )** to group logical operators.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| and | | |  | or | | |  | not | |
| A | **B** | **A and B** |  | **A** | **B** | **A or B** |  | **A** | **not(A)** |
| F | F | F |  | F | F | F |  | F | T |
| F | T | F |  | F | T | T |  |
| T | F | F |  | T | F | T |  | T | F |
| T | T | T |  | T | T | T |  |

Example

num = 44  
score = 66  
  
**if** num < 0 **or** num > 100:  
 print(**"Number is not between 0 and 100"**)  
  
**if** num > 0 **and** num < 50:  
 print(**"Number is between 0 and 50"**)  
  
**if** score >= 50 **and** score <= 100:  
 print(**"You passed"**)

Number is between 0 and 50

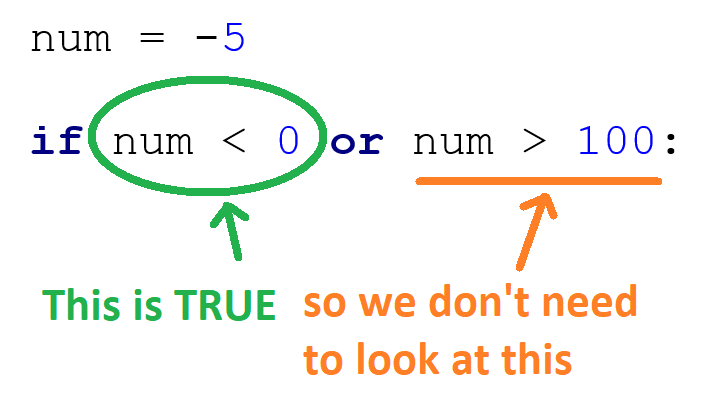
You passed

Logical Operators Short-Circuit

Sometimes Python does not need to look at some parts of an IF Statement that has more than one Logical Operator. This is called **short-circuiting**.

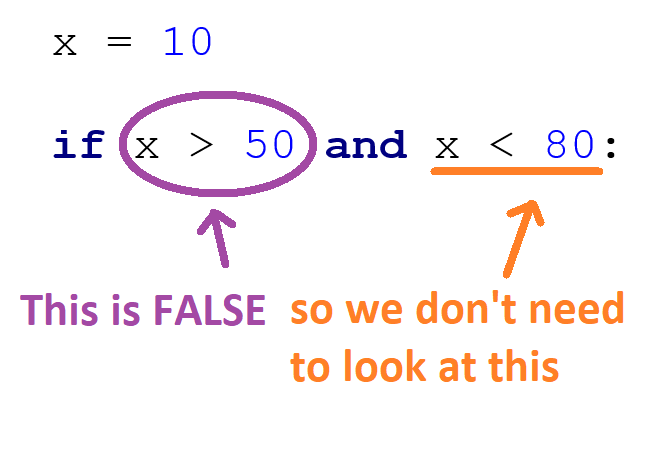
If we have an **or** statement with 2 **Relational Operators**, sometimes Python does not need to look at both Relational Operators. If the **LEFT** Relational Operator is **TRUE**, then Python does not need to check the **RIGHT** one.

num = -5  
  
**if** num < 0 **or** num > 80:



If we have an **and** statement with 2 **Relational Operators**, sometimes Python does not need to look at both Relational Operators. If the **LEFT** Relational Operator is **FALSE**, then Python does not need to check the **RIGHT** one:

x = 10  
  
**if** x > 50 **and** x < 80:



We can use this **Short-Circuit** is stop errors. For example, we want to stop an error from dividing by zero.

n1 = int(input(**"Enter a number"**))  
n2 = int(input(**"Enter another number"**))  
  
**if** n1 / n2 == 2: *# Risk divide by zero*

Enter a number: 5

Enter another number: 0

ZeroDivisionError: division by zero

We can do this instead:

n1 = int(input(**"Enter a number: "**))  
n2 = int(input(**"Enter another number: "**))  
  
**if** n2 != 0 **and** n1/n2 == 2: *# No risk of divide by zero*

### IF – ELSE IF – ELSE Statements

You can have many **else if** statements together. We use the word **elif** to make an else if statement. The last statement must be an **else**.

**Syntax**

*# IF - ELSE IF – ELSE statement*

if <Boolean-expression>:

\_\_\_\_<statement>

elif <Boolean-expression>:

\_\_\_\_<statement>

elif <Boolean-expression>:

\_\_\_\_<statement>

else:

\_\_\_\_ <statement>

Example

Example: A program to calculate a student’s grade:

If student’s grade is NOT between 0 and 100

Output “Wrong grade entered”

Else If student’s grade is greater than or equal to 80

Output “A”

Else If student’s grade is greater than or equal to 60

Output “B”

Else If student’s grade is greater than or equal to 50

Output “C”

Else

Output “Fail”

grade = float(input(**"Please enter a grade: "**))  
  
**if** grade < 0 **or** grade > 100:  
 print(**"Wrong grade entered"**)  
**elif** grade >= 80:  
 print(**"A"**)  
**elif** grade >= 60:  
 print(**"B"**)  
**elif** grade >= 50:  
 print(**"C"**)  
**else**:  
 print(**"Fail"**)

**Exercise 03-03**

*# Introduction Messages*print(**"Welcome to programming in Python"**)  
print(**"This program shows IF and logical operators"**)  
  
print(**"Enter a number between 0 and 100: "**)  
num = int(input(**"Enter a number between 0 and 100: "**))  
  
*# Test the numbers using relationship and logical operators***if** num < 0 **or** num > 100:  
 print(**"Number "** + str(num) + **" is not between 0 and 100"**)  
  
**if** num >= 0 **and** num < 50:  
 <Make the code output "Number num is between 0 and 50" using the num variable>  
  
<Make an ELSE IF statement to test if num is greater than or equal to 50 AND less than or equal to 100>  
 print(**"Number "** + str(num) + **" is between 50 and 100"**)  
  
<Make an ELSE statement>  
 <Make the code output "Number num is in the middle" using the num variable>

### Nesting

**Nesting** means using code blocks inside other code blocks. When you use nesting, you need to use **indenting** carefully.

**Syntax**

**if** <condition\_1>:  
 **if** <condition\_2>:  
 <statement>  
 **else**:  
 <statement>  
**else**:  
 <statement>

Example

score = int(input(**"Enter a score (0-100)"**))

**if** score >= 0 **and** score <= 100:  
 **if** score >= 50:  
 print(**"Pass"**)  
 **else**:  
 print(**"Fail"**)  
**else**:  
 print(**"Invalid score - must be 0-100"**)

Example: An application is needed to calculate a student’s grade:

If student’s score is greater than or equal to 85 and less than or equal to 100

Output “A++”

Else

If student’s score is greater than or equal to 80 and less than 85

Output “A+”

Else

If student’s score is greater than or equal to 70 and less than 80

Output “A”

Else

If student’s score is greater than or equal to 60 and less than 70

Output “B”

Else

If student’s score is greater than or equal to 50 and less than 60

Output “C”

Else

Output “F”

score = float(input(**"Enter a score (0-100)"**))

**if** score >= 85 **and** score <= 100:  
 grade = **"A++"  
else**:  
 **if** score >= 80 **and** score < 85:  
 grade = **"A+"  
 else**:  
 **if** score >= 70 **and** score < 80:  
 grade = **"A"  
 else**:  
 **if** score >= 60 **and** score < 70:  
 grade = **"B"  
 else**:  
 **if** score >= 50 **and** score < 60:  
 grade = **"C"  
 else**:  
 grade = **"D"**print(grade)

We can write this using **IF-ELSE-IF** statements:

score = float(input(**"Enter a score (0-100)"**))  
  
**if** score >= 85 **and** score <= 100:  
 grade = **"A++"  
elif** score >= 80 **and** score < 85:  
 grade = **"A+"  
elif** score >= 70 **and** score < 80:  
 grade = **"A"  
elif** score >= 60 **and** score < 70:  
 grade = **"B"  
elif** score >= 50 **and** score < 60:  
 grade = **"C"  
else**:  
 grade = **"D"**print(grade)

Using the **IF-ELSE IF** statements is better because:

* It is more simple and easy to read.
* As soon as the TRUE condition is found, the rest of the statements are not run.

**Exercise 03-04**

Make a program to get the age from the user and write IF-ELSE IF-ELSE statements to output the correct text:

If age < 2  
“You are a baby”

If age < 13  
“You are a child”

If age < 20  
“You are a teenager”

Else  
“You are an adult”

Get age from user

Enter your age: 20

You are an adult

### Ternary Operators

A short way of writing an IF statement is using the **Ternary Operator**, also called the **Conditional Operator**.

**Syntax**

|  |
| --- |
| <expression1> **if** <condition> **else** <expression2> |

Example

|  |
| --- |
| num1 = 5  **if** num1 > 0:  message = **"Positive" else**:  message = **"Negative"**  This is the same as:  message = **"Positive" if** num1 > 0 **else "Negative"** |

# Section 04: Loops

### Loops

**Loops** are another type of **branching control structure**. They are like the IF statement because they are used to make a branch for a **decision**. Looping means that a loop structure runs the same statements many times while a condition is **TRUE**. When the condition becomes **FALSE**, the loop stops.

IF statements use **Selection Statements**. They only run once. Just using Selection Statements for validating user input is not recommended.

Loops are **Iteration** **Statements**. They keep running until the condition is FALSE.

Python uses these loops:

* For
* While

### While Loops

The **While** statement is used when the amount of looping is not known. They test for if the condition is TRUE or FALSE inside the loop. The **While** loop checks the condition before running the statements inside the loop. This means that the code inside the **While** loop will continue running until the condition becomes FALSE.

header :

body

while <condition> :

<statements>

<set up statement>  
**while** <continue condition>:  
 <body statements>  
 <update the continue condition>

Example

If we want to print the sentence “Programming is fun!” 100 times, we can write a **While loop**:

count = 0 *# set up***while** count < 100: *# continuation condition* print(**"Programming is fun!"**) *# action to repeat* count = count + 1 *# update the condition*

Programming is fun!

Programming is fun!

Programming is fun!

Programming is fun!

...

Think about a program where a user enters a login password. How many times must the user enter in the password before they get the password correct? The answer could be 1 (they know the password), or 2 or 3 (they have made a mistake when entering the password). Maybe they could be allowed to enter it hundreds of times, but this might mean they are guessing or trying to hack the password. A **For** loop would not be good for this program. A **While** loop is better because the loop be can be set by a condition.

Example

answer = input(**"Are you hungry? (Enter yes/no): "**)  
**while** answer != **"yes" and** answer != **"no"**:  
 print(**"Invalid"**)  
 answer = input(**"Are you hungry? (Enter yes/no): "**)  
  
print(**"Input is good: "** + answer);

Are you hungry? (Enter yes/no): maybe

Invalid

Are you hungry? (Enter yes/no): don't know

Invalid

Are you hungry? (Enter yes/no): yes

Input is good: yes

**Exercise 04-01**

The program will ask a user to enter in the correct password before the program will continue.

Test the code with 1111 and then test with a wrong password.

correctPsw = 1111 *# set password*<Make an int called pswEntered and make it equal to 0>  
  
print(**"Program to Check Password"**)  
  
*# check password*<Make a while loop to see if pswEntered is NOT equal to correctPsw>  
 <Get an int from the user and put it in pswEntered. Make the message say "Please enter password: ">

print(**"Password accepted!"**)

Program to Check Password

Please enter password: 1234

Please enter password: 111

Please enter password: 1111

Password accepted!

Break and Continue

The **break** statement in Python stops the loop and goes to the next statement after the loop.

In the example below, var will start at 10 and decrease. If var is 5, the while loop will stop and it will go to the next statement after the loop while will print “Goodbye!”:

var = 10  
  
**while** var > 0:   
 print(**'Current variable value: '** + str(var))  
 var = var -1  
 **if** var == 5:  
 **break**print(**"Goodbye!"**)

Current variable value: 10

Current variable value: 9

Current variable value: 8

Current variable value: 7

Current variable value: 6

Good bye!

The **continue** statement in Python makes to code go to the beginning of the while loop. The continue statement does not let the other statements in the current iteration of the loop to run, and moves the code back to the top of the loop.

In the example below, var will start at 10 and decrease. If var is 5, the while loop will go back to the top of the while loop and the code will NOT print Current variable value: 5

var = 7  
  
**while** var > 0:   
 var -= 1  
 **if** var == 5:  
 **continue** print(**"Current variable value: "** + str(var))  
print(**"Goodbye!"**)

Current variable value: 6

Current variable value: 4

Current variable value: 3

Current variable value: 2

Current variable value: 1

Current variable value: 0

Goodbye!

Exercise 04-02

You need to write a program so a user can guess a number. This number is 44. They have 5 tries to guess the number. If they guess the number correctly, the while loop with exit using the **break** statement:

<Make answer equal 44>  
<Make count equal 5>  
  
<Make a while loop for while count is greater than 0>  
 <Get an int from the user and put it in guess. Make the message say "Enter a number: ">  
 <Make an if statement to see if guess equals answer>   
 <Print "You are correct">  
 **break** <Decrease count by 1>  
 print(**"Wrong. Try again. You have "** + str(count) + **" more tries."**)

Exercise 04-03

Write a while loop to print the numbers from 100 to 51. If the number is a multiple of 5, do not print it. You will need to use **continue** and the **modulus operator**. The output will be:

99

98

97

96

94

93

92

91

89

...

...

56

54

53

52

51

**Exercise 04-04 OPTIONAL**

This program will ask the user to enter a new password 2 times. They have only 3 chances to change their password. Test the code with different input passwords to test the loop and the conditions.

i = 3 *# counter for number of logins*<Make a variable called done and make it equal to false>print(**"Program to Change Password"**);  
print(**"Your password has expired and you have "** + str(i) + **" chances left to change your password"**)  
  
*# While password is not entered correctly AND more than 0 chances*<Make a while loop to say while not done and i is greater than 0>  
 newPsw = input(**"Please enter your new password: "**)  
  
 <Get input from the user and put it in pswEntered>  
 *# change password correct* <Make an IF statement to see if newPsw equals pswEntered>

<Make done equal to true>

**else**:  
 print(**"Error in entering password, please try again"**)  
 <Make i decrease by 1>  
 <Print out you have i chances left>  
  
  
<Make an IF statement to see if i is less than or equal to 0>

print(**"Password not changed"**) *# password accepted***else**:  
 print(**"Password accepted!"**)

Program to Change Password

Your password has expired and you have 3 chances left to change your password

Please enter your new password: 1234

Please enter password again to confirm: 12

Error in entering password, please try again

You have 2 chances left

Please enter your new password: 1234

Please enter password again to confirm: 1234

Password accepted!

### Nesting

**Nesting** means putting code blocks inside another code block. You can nest *IF statements*, *for* *loops* and *while* *loops*.

Example

**for** i **in** range(0,4): *# do this 4 times* **for** j **in** range(1,7): *# do this 6 times* print(j, end=**""**) *# print out j on the same line* print() *# make a new line*

123456

123456

123456

123456

* The first **for** loop makes the code inside it run 4 times.
* The code inside is another **for** loop, which goes from 1 to 6, and prints out numbers 1 to 6 on the same line.
* Still inside the first **for** loop, we print a new line.

**Exercise 04-10: Multiplication Tables (乘法表)**

* Write a program that will get a number from the user to print a multiplication table (乘法表).
* You will need to use **Nested For Loops**
* To make the table formatted correctly, you may need to use:
  + print(????, end = **"\t"**)

Enter the number for the times table: 5

1 2 3 4 5

2 4 6 8 10

3 6 9 12 15

4 8 12 16 20

5 10 15 20 25

**Exercise 04-11: Calendar OPTIONAL**

Write a program that will print a calendar for January 2007. The calendar will look like this:

M T W T F S Su

1 2 3 4 5 6 7

8 9 10 11 12 13 14

15 16 17 18 19 20 21

22 23 24 25 26 27 28

29 30 31

1. You can use loops and IF statements.
2. A good way is to use a FOR loop and an IF statement.
3. You can use the **modulus operator** (for example 7 % 4 = 3)
4. To get big spaces use the tab key ("\t")

**Exercise 04-12: Nesting OPTIONAL**

Make a program that will print a diamond ⧫. The code given below makes the top half of the diamond. You will need to make the code for the bottom half of the diamond. Formulas have been given for the top half.

**Margin** **Shape Row Counter Formula Number of \* Spaces**

\* **1** 0 (2\***0**)+1 = *1* *1 8*

\*\*\* **2** 1 (2\***1**)+1 = *3* *3 7*

\*\*\*\*\* **3** 2 (2\***2**)+1 = *5* *5 6*

\*\*\*\*\*\*\* **4** 3 (2\***3**)+1 = *7* *7 5*

\*\*\*\*\*\*\*\*\* **5** 4 (2\***4**)+1 = *9* *9 4*

\*\*\*\*\*\*\*\*\*\*\* **6** 5 (2\***5**)+1 = *11* *11 3*

\*\*\*\*\*\*\*\*\*\*\*\*\* **7** 6 (2\***6**)+1 = *13* *13 2*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* **8** 7 (2\***7**)+1 = *15* *15 1*

\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*

\*\*\*

\*

**for** i **in** range(8): *# do this 8 times  
 # print spaces first* **for** j **in** range(0,7-i): *# go from 0 to 7-i* print(**" "**, end=**""**) *# print space on same line  
  
 # print stars* **for** k **in** range(0, (2\*i + 1)): *# go from 0 to (2\*i + 1)* print(**"\*"**, end=**""**) *# print star on same line* print() *# print a new line  
  
# now print the bottom half of the diamond*

\*

\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*

\*\*\*

\*

**Exercise 04-13: Calculations in a Loop OPTIONAL**

1. Write a program to get many numbers from the user.
2. The user will type in a number and press the ENTER key.
3. They can type in many numbers until they type “n” or “N”.
4. After they type “n” or “N”, the program will calculate the sum and the average of the numbers they typed in.
5. You will need a WHILE loop.
6. You will also need a variable to count the number of numbers the user types in.

Here is an example:

Enter a number: 5

Do you want to enter another number?: y

Enter a number: 8

Do you want to enter another number?: y

Enter a number: 4

Do you want to enter another number?: n

Sum = 17

Average = 5.666666666666667

**Get a number from the user**

**Ask if the user wants to enter another number**

**Calculate sum and average and show to user**

**No**

**Yes**

### For Loops

The **For Loop** statement loops for a set number of times. The For Loop statement is used when the programmer knows how many times a loop will run. For example, an application that needs to get 10 inputs from the user would repeat the input statement 10 times. Another example is recording grades for a class of 24 students. The program would repeat grade inputs and look at each grade 24 times. In both examples, the number of loops is known.

The **For Loop** statement has 3 parts:

* The <variable> that is changed and that can be used inside the loop.
* The <sequence> is a range of values that the loop looks at one by one.
* The <statements> inside the loop.

**for** <variable> **in** <sequence>:  
 <statements>

Example

**for** i **in** range(0,5):  
 print(i)

0

1

2

3

4

You can see that range(0,5) goes from 0 to 4. It does not go to 5.

Range

The range() function is used to control how many times a loop is repeated. The range() function can have 1, 2 or 3 parameters:

range(<start>, <stop>, <step>)

* <start> is the starting value
* <stop> is the value to stop before (this value is NOT included in the loop)
* <step> tells the loop how much to increment or decrement (go up or down)

Example

**for** i **in** range(20, 56, 5):  
 print(i)

20

25

30

35

40

45

50

55

We can also use a **for loop** to go through each letter of a string:

animal = **'elephant'  
  
for** letter **in** animal:  
 print(letter)

e

l

e

p

h

a

n

t

If you want to keep the text on the same line, you need to write:

, end=""

before the final round bracket:

word = **"Mouse"  
for** l **in** word:  
 print (l, end = **""**)

Mouse

**Exercise 04-05**

<Get an int from the user and put it in studentNum. Make the message say "Enter the number of student grades: ">  
  
*# loop for the number of students*<Make a for loop from 0 to studentNum>  
 <Get an int from the user and put it in grade. Make the message say "Enter grade 1" or "Enter grade 2" ... (use the i variable)>  
  
 <if grade is greater than or equal to 50, print out "Student has passed", else print "Student has failed">

Enter the number of student grades: 4

Enter grade 1: 56

Student has passed

Enter grade 2: 89

Student has passed

Enter grade 3: 48

Student has failed

Enter grade 4: 75

Student has passed

**Exercise 04-06 OPTIONAL**

Use a for loop to print out this:

23

27

31

35

39

43

47

51

55

**Exercise 04-07 OPTIONAL**

Use a for loop to print out this:

100

94

88

82

76

70

**Exercise 04-08 OPTIONAL**

* Get a word or sentence from the user.
* Use a for loop to look at each letter.
* If the letter is a vowel (a, e, i, o, u), print out an **X**
* If the letter is a consonant, print it out.
* Count the number of vowels in the word or sentence.
* If the user types in: *A dog is a man's best friend* the program will print:

A dXg Xs X mXn's bXst frXXnd

There are 7 vowels in the word or sentence.

Else

The **While Loop** and the F**or Loop** can have else statements. It is similar to the **if-else** statement.

There is a difference: the **else** block will run anytime the loop condition is **False**. This means that the **else** block it will run if the loop is never entered or if the loop exits normally. If the loop exits because of a **break**, the **else** will not be run.

The example below shows an **else** statement with a while statement that prints a number while count is less than 5, otherwise the **else** statement gets run:

While loop

count = 0  
  
**while** count < 5:  
 print(count, **" is less than 5"**)  
 count += 1  
**else**:  
 print(**"Now count is not less than 5"**)

0 is less than 5

1 is less than 5

2 is less than 5

3 is less than 5

4 is less than 5

Now count is not less than 5

The example below shows an **else** statement with a for statement that prints a number while count is less than 5, otherwise the **else** statement gets run:

For loop

**for** count **in** range(0,5):  
 print(count, **" is less than 5"**)  
**else**:  
 print(**"Now count is not less than 5"**)

0 is less than 5

1 is less than 5

2 is less than 5

3 is less than 5

4 is less than 5

Now count is not less than 5

**Exercise 04-09 OPTIONAL**

* Get a word or sentence from the user. Make the message say “**"Please enter a word or sentence: "**
* Make a for loop to look at each letter of the word or sentence.
* If the letter is an ‘x’, print “This has the letter ‘x’ in it”, and break the loop.
* Make an else statement for the loop (not the if statement), with the message “This does not have the letter ‘x’ in it”

Please enter a word or sentence: extra

This has the letter 'x' in it

Please enter a word or sentence: elephant

This does not have the letter 'x' in it

# Section 05: Lists and Dictionaries

### Lists

**Lists** are a collection of variables or **elements**. A list is a variable and it has one name. Each **element** can be read by using the list **name** and **index**.

The first element is always 0. Lists use square brackets **[ ]**

Making Lists

# 

*# Make- an empty list*my\_list = []  
  
*# Make a list with some elements*num\_list = [2, 45, 9, 10]  
  
*# Make a list of strings*str\_list = [**"red"**, **"green"**, **"blue"**]  
  
*# Make a list of mixed data*record = [**"01132567"**, **"John Smith"**, 95.5]

The num\_list list above will look like this:

|  |  |  |  |
| --- | --- | --- | --- |
| **2** | **45** | **9** | **10** |
| **0** | **1** | **2** | **3** |

If we want to put a value from the list into a variable, we can do this:

str\_list = [**"red"**, **"green"**, **"blue"**]  
first\_colour = str\_list[0]  
print(first\_colour)

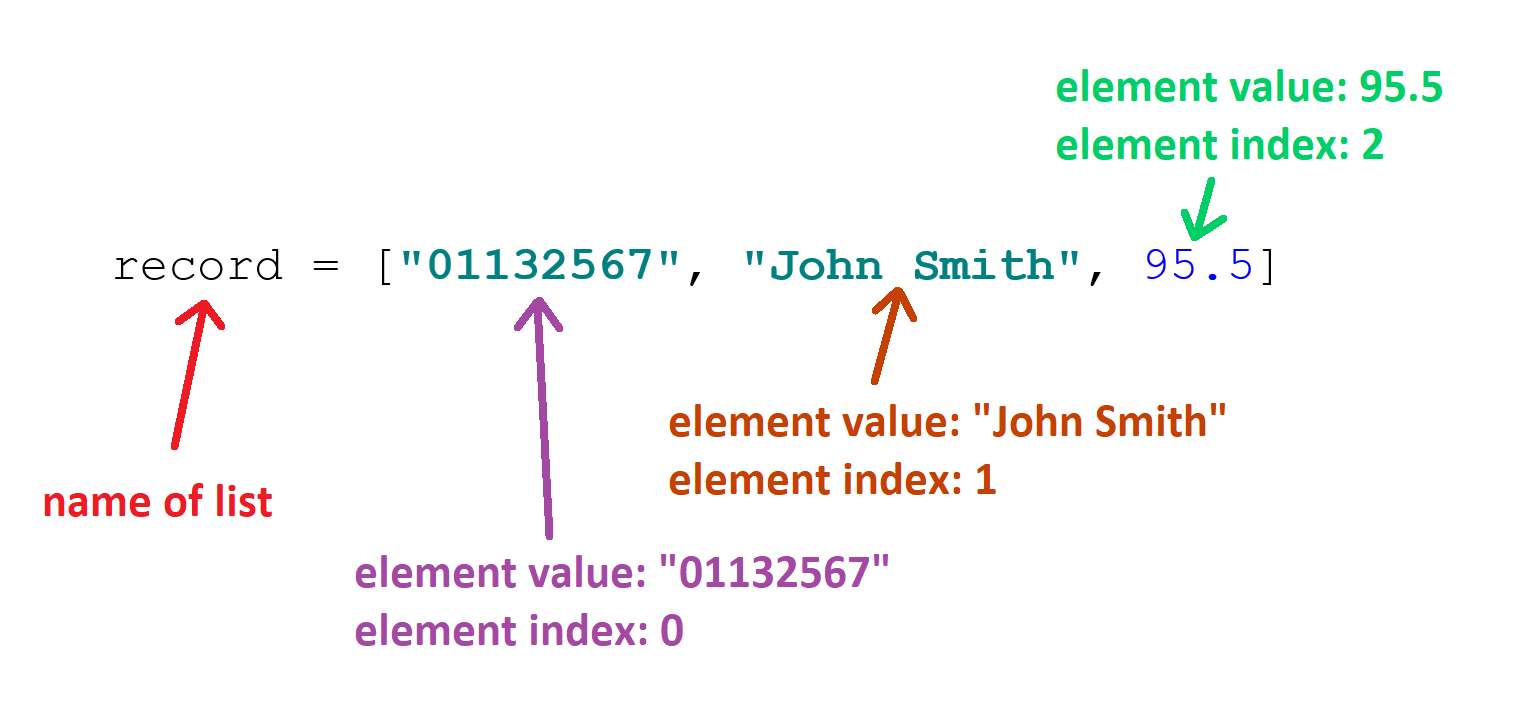
This will print out **red**, because red is at the first index of the list, which is index 0.

To change the value of an element, we write the list name with the index that we want to change, and give it a new value:

str\_list = [**"red"**, **"green"**, **"blue"**]  
str\_list[1] = **"orange"**print(str\_list)

['red', 'orange', 'blue']

Parts of a List



Length of a List

If we want to know the length of a list, we can use the len() function. The length of this list is 4:

my\_list = [2, 45, 9, 10]  
print(len(my\_list))

4

Adding a record to a List

We can add or **append** a record to the end of a list using the append() function:

my\_list = [2, 45, 9, 10]  
my\_list.append(23)

Using the For statement

We can go through (**iterate**) each element of a list using a **for** loop:

**for** <item> **in** <a list of items>:  
 < statements to do something each <item> >

Example

my\_list = [2, 3, 4, 5]  
  
**for** num **in** my\_list:  
 print(2 \* num)

4

6

8

10

Example

This program has a list of names. The user types in a name to search in the list. This must be made lowercase so we don’t have to worry about capital letters. A for loop goes through each name in the list. If the user’s name is in the list, the found variable is set to the name, and the loop is broken using break. If the name was found, the message “Found” is printed.

name\_list = [**'John'**, **'Alan'**, **'Paul'**, **'Andy'**]  
name = input(**'Enter a name: '**).lower() *# get name and make it lower*  
found = **None** *# make a placeholder***for** e **in** name\_list:  
 **if** name == e.lower(): *# found the name* found = e *# update placeholder to hold the found name* **break** *# stop the loop***if** found == **None**:  
 print(**'Not found'**)  
**else**:  
 print(**'Found'**)

Enter a name: John

Found

Enter a name: Sam

Not found

Exercise 05-01

* Make a list with these values: 77, 46, 76, 36, 98
* Print out the third element value.
* Use a for loop to calculate the total of the values
* Print out the total

Third element is: 76

Total is: 333

Exercise 05-02

<get an int from the user and put it in n. Make the message say "Enter the number of values to insert: "

<make an empty list called myList>

<make a for loop from 0 to n>  
 <get an int from the user and put it in get\_num. Make the message say "Enter a number to insert: ">

<append get\_num into the list>  
  
<make a variable called sum and make it equal to sum(myList)  
<make a variable called average and make it equal to the average value  
<print out the sum>  
<print out the average>

Enter the number of values to insert: 5

Enter a number to insert: 3

Enter a number to insert: 4

Enter a number to insert: 5

Enter a number to insert: 6

Enter a number to insert: 7

Sum is: 25

Average is: 5.0

### Dictionaries

**Dictionaries** are like **lists**, but you get values by looking up a **key**, instead of an **index**.  A **key** can be any string or number. Dictionaries have curly brackets around them **{ }**

Making a Dictionary

d = {**'key1'** : 1, **'key2'** : 2, **'key3'** : 3}

This is a dictionary called **d** with 3 key-value pairs. The **key** called 'key1' has the value 1, 'key2' has the value 2, and so on.

dict = {**'Name'**: **'Sam'**, **'Age'**: 20, **'Sex'**: **'Male'**}

This is a dictionary called **dict** with 3 **key-value pairs**. The **key** called 'Name' has the value 'Sam', the **key** 'Age' has the value 2, and the **key** 'Sex' has the value 'Male'.

Getting a value from a dictionary

To get a value out of a dictionary, we write the name of the dictionary, then a **square** **bracket** **[** and then the key:

dict = {**'Name'**: **'Sam'**, **'Age'**: 20, **'Sex'**: **'Male'**}  
  
age\_value = dict[**'Age'**]  
  
print(age\_value)

20

Adding key-value pairs to a dictionary

If you want to add a new **key-value pair** to a dictionary, use this code:

dict[**'<key\_name>'**] = <value>

Example

dict = {**'Name'**: **'Sam'**, **'Age'**: 20, **'Sex'**: **'Male'**}  
  
dict[**'Height'**] = 170  
  
print(dict)

{'Name': 'Sam', 'Sex': 'Male', 'Age': 20, 'Height': 170}

Deleting a key-value pair from a dictionary

To delete a key-value pair from a dictionary, we can use:

**del** dict[**'<key\_name>'**]

Example

dict = {**'Name'**: **'Sam'**, **'Age'**: 20, **'Sex'**: **'Male'**}  
  
**del** dict[**'Sex'**]  
  
print(dict)

{'Name': 'Sam', 'Age': 20}

Length of a dictionary

To find the length of a dictionary (how many values are in the dictionary), you can use:

dict = {**'Name'**: **'Sam'**, **'Age'**: 20, **'Sex'**: **'Male'**}  
  
length\_dict = len(dict)  
  
print(length\_dict)

3

Using For Loops on Dictionaries

You can also use a for loop on a dictionary to loop through its keys. Dictionaries are unordered, so sometimes they do not print out in order. To print out the **key**:

d = {**"key\_1"** : **"value\_1"**, **"key\_2"** : **"value\_2"**, **"key\_3"** : **"value\_3"**}  
  
**for** key **in** d:   
 print(key)

key\_1

key\_3

key\_2

To print out the **value**:

d = {**"key\_1"** : **"value\_1"**, **"key\_2"** : **"value\_2"**, **"key\_3"** : **"value\_3"**}  
  
**for** key **in** d:   
 print(d[key])

value\_2

value\_1

value\_3

Example

prices = {**'apples'**: 15, **'bananas'**: 35, **'grapes'**: 12}  
  
**for** p **in** prices:  
 print(**"Price of "** + p + **" is: "** + str(prices[p]))

Price of bananas is: 35

Price of grapes is: 12

Price of apples is: 15

Exercise 05-03

* Make an empty dictionary called *student*
* Add a key called “Name” and the value “James”
* Add a key called “Age” and the value 21
* Add a key called “Course” and the value “IT”
* Add a key called “ID” and the value “2016A001”
* Print the whole dictionary
* Print the value in the “Name” key and then the value in the “ID” key
* Delete the key-value pair in the “Course” key
* Use a for loop to print each value in the dictionary

{'Course': 'IT', 'ID': '2016A001', 'Age': 21, 'Name': 'James'}

James: 2016A001

2016A001

21

James

Exercise 05-04

Here is a table of students and their marks:

|  |  |
| --- | --- |
| Name | Mark |
| Sam | 90.5 |
| Jane | 85.4 |
| Max | 92.3 |
| Alice | 64.5 |
| John | 69.4 |

* Make a **dictionary** called **marks** and put these values into it. The *Name* is the key and the *Mark* is the value.
* Make a variable called *sum* and make it equal to 0
* Use a for loop to go through the dictionary and add the mark to *sum*
* Print the *sum*
* Calculate the average and print it (use the len() function).

Sum: 402.1

Average: 80.42

Exercise 05-05

You are going shopping for fruit. You know the prices of the fruit and how many you need (quantity):

|  |  |  |
| --- | --- | --- |
| Fruit | Price | Quantity |
| Banana | 4 | 1 |
| Apple | 2 | 0 |
| Orange | 1.5 | 3 |
| Pear | 3 | 2 |

* Make a dictionary called *price* and use the **name** of the fruit for the **key** and the **price** of the fruit for the **value**.
* Make a dictionary called *quantity* and use the ***name*** of the fruit for the **key** and the **quantity** of the fruit for the **value**.
* Make a variable called sum and make it equal to 0
* Because both dictionaries have the same **keys**, we can use a for loop to calculate the total cost of the fruit. Make a **for loop** to go through the *price* dictionary and make sum equal to the *price* of the fruit \* the *quantity* of the fruit
* Print out the total cost of the fruit.

Total cost: 14.5

Lists and Dictionaries

Lists:

* Can be ordered
* Just have values (no keys)

Dictionaries:

* Have a key and a value
* Easy to search using a key

Lists compared to Dictionaries

|  |  |  |
| --- | --- | --- |
|  | Dictionary | List |
| Make | nums = {} | nums = [] |
| Assign element value | nums[**"key1"**] = 10 nums[**"key2"**] = 20 nums[**"key3"**] = 30 | nums.append(10) nums.append(20) nums.append(30) |
| Get elements | print(nums[**"key1"**]) | print(nums[0]) |
| Iteration (over elements) | **for** key **in** nums:  print(nums[key]) | **for** n **in** nums:  print(n) |

**Exercise 05-06**

* Make a program to print out the Fibonacci Sequence.
* The Fibonacci Sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21 …

**Make an empty list**

**Append 0 to the list**

**Append 1 to the list**

**For index = 2 to 9**

**Make the element for the current index equal to the previous element plus the element before the previous element**

**Increase index by 1**

**Use another FOR LOOP to print out the list**

**Loop**

<make an empty list called fib>  
  
<append 0 to the fib list>  
<append 1 to the fib list>  
  
<make a for loop from 2 to 10>  
 <append to the fib list the previous element plus the element before the previous element>

<make a for loop to print out the elements in the fib list>

0, 1, 1, 2, 3, 5, 8, 13, 21, 34,

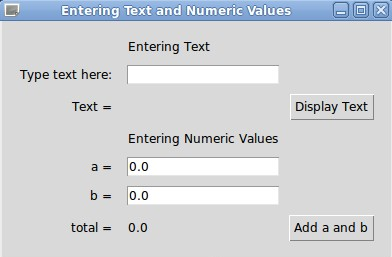
# Section 06: Graphical User Interface (GUI)

Graphical User Interface (GUI)

Python has many options for developing **Graphical User Interfaces (GUIs).**

**Tkinter** is the standard GUI library for Python. Python used with Tkinter provides a fast and easy way to create GUI applications. Tkinter has a powerful **object-oriented** interface to the Tk GUI toolkit.

Using Tkinter with PyCharm lets you make applications like **windows** and **forms**. Inside the windows you can add **widgets**, which are controls like buttons, radio buttons, text boxes and labels.



Applications made with Tkinter are **event-driven**. This means the application does tasks when the user uses the application. Programmers must write the code like **event handlers** to do the tasks when the user makes an **event**.

To make a GUI application, we need to write Python code to make the window, widgets and even handlers. We cannot use a toolbox with drag and drop.

Making a basic Tkinter application

Creating a GUI application using Tkinter is an easy task. All you need to do is do these steps:

* Import the Tkinter module. This lets us work with Tkinter.
* Make the GUI application main window.
* Add some of the widgets to the GUI application.
* Make the main event loop to listen to events made by the user.

We start by importing the Tkinter module. It contains all classes, functions and other things needed to work with the Tkinter toolkit:

**from** tkinter **import** \*

To start Tkinter, we have to create a Tk root widget. This is a window, with a title bar. You should only make one root widget for each program, and it must be created before any other widgets.

root = Tk()

Next, we make a **Label** widget inside the root window. A label can show text or an image. Here we will put some text. Next, we call the **pack** method on this widget. This tells it to size itself to fit the text, and make itself visible:

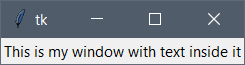
myLabel = Label(root, text=**"This is my window with text inside it"**)

myLabel.pack()

To make the window open and stay open, we use the code:

root.mainloop()

When we run the code, it will make this window appear:



Here is the full code:

**from** tkinter **import** \* *# get everything in tkinter module  
  
# make a window with text inside it*root = Tk() *# make an empty window object*myLabel = Label(root, text=**"This is my window with text inside it"**) *# make some text, put it in the window*myLabel.pack() *# put the text in the window*root.mainloop() *# keep window always open*

Organizing the layout

We need to know how to organize the layout of widgets in our window. When we start making our window, we always need this code:

**from** tkinter **import** \* *# get everything in tkinter module*root = Tk() *# make an empty window object*

<code...>

root.mainloop() *# keep window always open*

We use **Frames** to organize our layout. A **Frame** is a rectangle that you cannot see, and you can put things inside it. **Frames** help keep things tidy. For example, we have our window, and a Frame at the top and a Frame at the bottom:

**Window**

**Top Frame**

**Bottom Frame**

To make the top frame, we can write:

topFrame = Frame(root)

This makes a container we cannot see, called **topFrame**, and it goes in the main window (root). To make it show, we have to use the pack() function again:

topFrame.pack()

Now we can make the bottom frame:

bottomFrame = Frame(root)  
bottomFrame.pack(side=BOTTOM)

There is a parameter inside the pack() function. This says exactly where to put the frame. So we use side=BOTTOM. We don’t need to write topFrame.pack(side=TOP) for the top frame.

Now that we have 2 frames, we can put some buttons into the frames. To make a button, we write:

button1 = Button(topFrame, text=**"Button 1"**, fg=**"red"**)

* The name of the button is **button1**
* We tell the button to go into the **topFrame**
* We make the text inside the button say “Button 1”
* We make the text colour red using **fg**

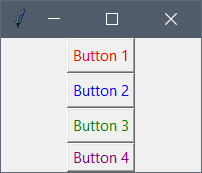
Here are 3 more buttons. Button 4 is inside the **bottomFrame**:

button2 = Button(topFrame, text=**"Button 2"**, fg=**"blue"**)  
button3 = Button(topFrame, text=**"Button 3"**, fg=**"green"**)  
button4 = Button(bottomFrame, text=**"Button 4"**, fg=**"purple"**)

Of course now we need to use the pack() function to make the buttons be seen:

button1.pack()  
button2.pack()  
button3.pack()  
button4.pack()

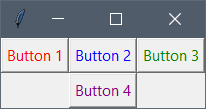
If we run the program now, we see this:



This is OK, but we want button 1, 2 and 3 to be on the same line. To do this we must change the code to:

button1.pack(side=LEFT)  
button2.pack(side=LEFT)  
button3.pack(side=LEFT)

This makes the buttons got to as left as possible:

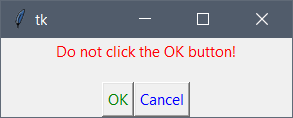


Here is the full code:

**from** tkinter **import** \* *# get everything in tkinter module*root = Tk() *# make an empty window object  
  
# make the top frame*topFrame = Frame(root)  
topFrame.pack()  
  
*# make the bottom frame*bottomFrame = Frame(root)  
bottomFrame.pack(side=BOTTOM) *# put it at the bottom  
  
# make 4 buttons. The first 3 are in the top frame  
# and button 4 is in the bottom frame*button1 = Button(topFrame, text=**"Button 1"**, fg=**"red"**)  
button2 = Button(topFrame, text=**"Button 2"**, fg=**"blue"**)  
button3 = Button(topFrame, text=**"Button 3"**, fg=**"green"**)  
button4 = Button(bottomFrame, text=**"Button 4"**, fg=**"purple"**)  
  
*# make the buttons show*button1.pack(side=LEFT)  
button2.pack(side=LEFT)  
button3.pack(side=LEFT)  
button4.pack()  
  
root.mainloop()

Exercise 06-01

Make this application:



* The label is in the **topFrame**
* The 2 buttons are in the **bottomFrame**

Widgets

Here are some basic widgets. There are many more available in Tkinter like **radio** **buttons**, **checkboxes** and **menus**:

|  |  |  |
| --- | --- | --- |
| **Widget** | **Syntax** | **Example** |
| Label | Label(location, text=**"..."**) | Label(root, text=**"My text"**) |
| Button | Button(location, text=**"..."**, fg=**"..."**) | Button(root, text=**"My text"**, fg=**"red"**) |
| Text Box | Entry(location) | Entry(root) |
| List Box | Listbox(location)  listbox.insert(location, **"..."**) | Listbox(root)  Listbox1.insert(0, **"First"**)  Listbox(root)  Listbox1.insert(END, **"Second"**) |

**Don’t forget to use** pack() **on your widgets to make them show in the window.**

Events

**Events** are code that runs when the user does something on a **widget**. For example, if the user clicks on a button, this is an event.

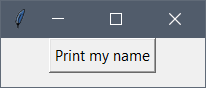
The **events** can run if we make some code inside a widget. We can write the code to be run when the event is **triggered**. For example, the button is clicked. We use the command parameter to tell the widget what to do if it is used:

button1 = Button(root, text=**"Print my name"**, command=printName)

Here, button1 has the text “Print my name” and we want it to use a **function** called printName when it is clicked. This **function** must be written by you. This is the printName function:

**def** printName():  
 print(**"My name is Sam"**)

We will look at **functions** later. So when the button is clicked, Python will go to this **function** and run the code in it. It will print “My name is Sam” in the *Run* window at the bottom. Here is the window:



When the button is clicked, the message will show in the *Run* window:

My name is Sam

Here is the full code:

**from** tkinter **import** \* *# get everything in tkinter module*root = Tk() *# make an empty window object  
  
# function to print message***def** printName():  
 print(**"My name is Sam"**)  
  
*# button that uses the printName function to print out message*button1 = Button(root, text=**"Print my name"**, command=printName)  
button1.pack()  
  
root.mainloop()

If we have a text box (**Entry**) and we want to **get** the text that the user wrote inside the **Entry**, then we can use this code:

user\_text = Entry.get(myEntry)

This goes to the **Entry** box called myEntry and gets the text inside and puts it in a variable called user\_text.

If we want to **set** the text inside the myEntry **Entry**, we can use this code:

myEntry.insert(0, **"Some text"**)

This puts the text “Some text” at the first position of the myEntry **Entry**.

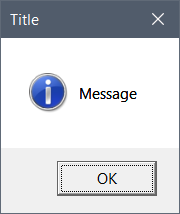
Message Boxes

To make a message box using Tkinter, we must import the message box class at the top:

**import** tkinter.messagebox

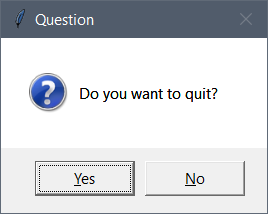
To make a message box, we use this code:

tkinter.messagebox.showinfo(**"Title"**, **"Message"**)



We can use the message box to ask the user a question like “Do you want to quit?” We must put the answer into a variable. Then we can use an if statement to see if the user chose *Yes* or *No*:

answer = tkinter.messagebox.askquestion(**"Question"**, **"Do you want to quit?"**)  
  
**if** answer == **"yes"**:  
 print(**"OK"**)



If the user clicked the *Yes* button, this will show in the *Run* window:

OK

Here is the full code:

**from** tkinter **import** \* *# get everything in tkinter module***import** tkinter.messagebox *# import the messagebox class*root = Tk() *# make an empty window object  
  
# make a simple message box*tkinter.messagebox.showinfo(**"Title"**, **"Message"**)  
  
*# make a message box with yes and no buttons*answer = tkinter.messagebox.askquestion(**"Question"**, **"Do you want to quit?"**)  
  
**if** answer == **"yes"**:  
 print(**"OK"**)  
  
root.mainloop()

Exercise 06-02

We are going to make this form:

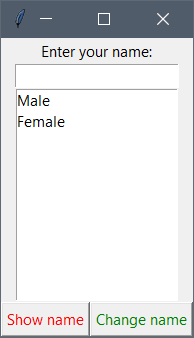
Label

List box with 2 items:  
Male  
Female

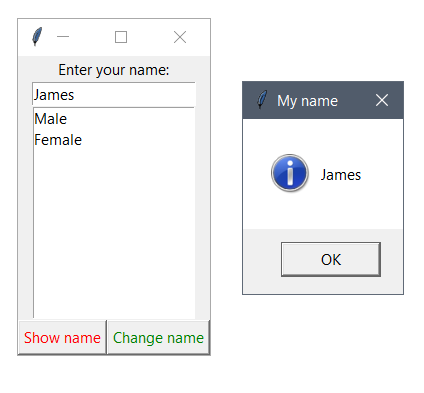
Button

Button

Entry



1. Import *tkinter*
2. Import the *tkinter.messagebox* class
3. Make the *root* window object
4. Make a top **Frame**
5. Make a bottom **Frame**
6. Make a **Label** that says “Enter your name” and put it in the top **Frame**
7. Make an **Entry** text box and put it in the top **Frame**
8. Make a **Listbox** and put it in the top **Frame**
   1. Put “Male” into the Listbox
   2. Put “Female into the Listbox
9. Make a **Function** called *showName*
   1. Get the text inside the **Entry** and put it in a variable
   2. Make a **messagebox** with the title that says “My name” and put the text inside the **Entry** into the **messagebox**
10. Make a **Function** called *changeName*
    1. Change the text inside the **Entry** to “Sam”
11. Make a **Button** and put it in the bottom **Frame**
    1. Make the button text say “Show name”
    2. Make the colour red
    3. Make the command use the *showName* **Function**
12. Make a **Button** and put it in the bottom **Frame**
    1. Make the button text say “Change name”
    2. Make the colour green
    3. Make the command use the *changeName* **Function**
13. Run the *mainloop* function of *root*



Exercise 06-03

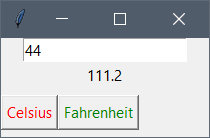
We are going to make this form to convert temperatures between Celsius and Fahrenheit:

Entry

Label

Button

Button



The user types in a temperature and clicks the *Celsius* button to change the temperature to *degrees* *Celsius,* or clicks the *Fahrenheit* button to change the temperature to *degrees Fahrenheit.*

1. Import *tkinter*
2. Make the *root* window object
3. Make an **Entry** text box
4. Make a **Label** that says ""
5. Make a **Function** called *calcC*
   1. Get the text inside the **Entry**, make it a float and put it in a variable
   2. Calculate the temperature in *degrees Fahrenheit*
   3. Put the new temperature in the label using this code:

myLabel.config(text=str(new\_temp))

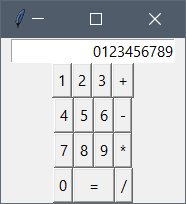
1. Make a **Function** called *calcF*
   1. Get the text inside the **Entry**, make it a float and put it in a variable
   2. Calculate the temperature in *degrees Celsius*
   3. Put the new temperature in the label using this code:

myLabel.config(text=str(new\_temp))

1. Make a **Button**
   1. Make the button text say “Celsius”
   2. Make the colour red
   3. Make the command use the *calcC* **Function**
2. Make a **Button**
   1. Make the button text say “Fahrenheit”
   2. Make the colour green
   3. Make the command use the *calcF* **Function**
3. Run the *mainloop* function of *root*

Exercise 06-04

1. Make a simple calculator.
2. The instructions below will help you:



1. Make a new project called *Calculator*
2. Make the window look like the calculator. Look at the example above to help you.
3. The diagrams show how the program will work.
4. Use the code below to help you:

|  |
| --- |
| **from** tkinter **import** \* root = Tk()  *# Global variables* <Make a variable called num1 and make it equal to 0><Make a variable called num1 and make it equal to 0>  <Make a variable called operator and make it equal to ''>  *# Put the number of the button into the text box* **def** insertText(num):  <Make a variable called typedNum and make it equal to the value in myEntry plus num> myEntry.delete(0, END) *# Delete the value in the text box*  myEntry.insert(0, typedNum) *# Put the new value in the text box*  *# Set the operator and clear the text box* **def** makeOperator(op):  **global** num1 *# We need to use the global variables*  **global** operator  <Make num1 equal to the value in the text box cast to an int>  <Make operator equal to op>  <Delete the value inside the text box>  *# Calculate the answer and put it in the text box* **def** calculateAnswer():   **global** reset *# We need to use the global variable*  <Make num2 equal to the value in the text box cast to an int>   **if** operator == **"+"**:  answer = num1 + num2  **elif** <finish this if statement for -, \* and / >   <Delete the value inside the text box>  <Put the answer into the text box>  *# Make the frames* topFrame = Frame(root) topFrame.pack() frame1 = Frame(root) frame1.pack() frame2 = Frame(root) frame2.pack() frame3 = Frame(root) frame3.pack() frame4 = Frame(root) frame4.pack() frame5 = Frame(root) frame5.pack(side=BOTTOM)  *# Make the text box* myEntry = Entry(frame1, justify=RIGHT) myEntry.pack()  *# Make the buttons* button1 = Button(frame2, text=**"1"**, command=**lambda**: insertText(**'1'**)) button2 = Button(frame2, text=**"2"**, command=**lambda**: insertText(**'2'**)) button3 = Button(frame2, text=**"3"**, command=**lambda**: insertText(**'3'**)) buttonPlus = Button(frame2, text=**"+"**, command=**lambda**: makeOperator(**'+'**))  button1.pack(side=LEFT) button2.pack(side=LEFT) button3.pack(side=LEFT) buttonPlus.pack(side=LEFT)  <Make button 4,5,6 and – and put them into frame3>  <Make button 7,8,9 and \* and put them into frame4>  button0 = Button(frame5, text=**"0"**, command=**lambda**: insertText(**'0'**)) buttonEquals = Button(frame5, text=**"="**, width=4, command=**lambda**: calculateAnswer()) buttonDivide = Button(frame5, text=**"/"**, command=**lambda**: makeOperator(**'/'**))  button0.pack(side=LEFT) buttonEquals.pack(side=LEFT) buttonDivide.pack(side=LEFT)  root.mainloop() |

This will call insertText()

Call a method insertText()with a string parameter "6"

The method insertText()makes a string called *typedNum*

typedNum is put into the textbox

Click 6

This will repeat every time a number is clicked

Inside insertText():

typedNum is equal to the value inside the textbox joined to the parameter num

Click +

This will call makeOperator()

Call a method makeOperator()with a string parameter "+"

The method makeOperator() puts the value of the textbox into the *num1* variable

The method makeOperator() puts the op parameter into the variable operator1

The text inside the textbox is cleared

Then a new number is clicked. The number code will be run.

Make these *global* variables:

* num1 (int)
* num2 (int)
* operator (string)

Click =

The method calculateAnswer() will be called

Call the method   
calculateAnswer()

The method calculateAnswer():

* puts the value in the textbox into num2
* looks at the operator variable and uses an IF statement to see if it should add, subtract, multiple or divide *num1* and *num2*

The method then does the calculation and shows the answer in the textbox

EXTRA: Write some code to make the program start a new calculation after it has made one calculation.

1. Find a way to reset the calculator after it has finished with one calculation. This will let the user make a new calculation. It is recommended you use a global Boolean that is set to True after the = button is clicked. When a number button is clicked, the *insertText()* function is called. You can write some code in here to see if the Boolean is TRUE. If it is TRUE, then you can make the textbox blank.
2. For the  button, this is called a *factorial.* For example, 4! or 4 *factorial* is equal to 4 x 3 x 2 x 1 = 24. If this button is clicked, the calculator should take the number that is inside the textbox and calculate the factorial of that number, and then show the answer. You will need a loop to do this calculation.

# Section 07: Functions

### Functions

What is a Function?

A **function** is a block of code that is used to perform a task. It can be used many times.

As you already know, Python gives you many built-in functions like print(). You can also make your own functions. These functions are called **user-defined functions**. Functions let you make more efficient programs and let you reuse lots of code.

name = input(**"Enter your name: "**)  
  
print(name)

print() function

input() function

Making a function

To make a function we write the keyword def, then the function name, and then round brackets with parameters:

**def** function\_name(parameters):  
 <code>

The first line is the function **header**. The code inside the function is the function **body**.

Sometimes the function returns something, shown by the **return** keyword. For example:

**def** max\_num(num1, num2):  
 **if** num1 > num2:  
 largest = num1  
 **else**:  
 largest = num2  
   
 **return** largest

This function is called max\_num, and it has 2 parameters called num1 and num2.

It finds the largest number and puts it in the variable largest.

It then returns the variable largest.

**You must define the function before you use it, otherwise you will get an error.**

Calling a function

To use a function, we must **call** it. When we call the function, we must give the call the correct parameters. To call the max\_num function, we must give it 2 numbers:

x = 5  
y = 8  
  
my\_max = max\_num(x, y)  
print(my\_max)

8

Value-Returning Functions and Non-Value-Returning Functions

The function max\_num **returns** a value. Some functions like print() do not return a value. If the function returns a value, we must put it in a variable:

my\_max = max\_num(x, y)

If the function does NOT return a value, we do not need to put anything into a variable:

**def** print\_hello():  
 print(**"Hello"**)  
  
*# call function print\_hello()*print\_hello()

Hello

Scope of Parameters and Local Variables

**Parameters** can only be seen and used in their **function**. They cannot be seen or used outside of their function. This is because the **scope** of the **parameters** is just their function.

**Local variables** in a function are the same. **Local variables** can only be seen and used in their **function**. They cannot be seen or used outside of their function. This is because the **scope** of the **Local variables** is just their function:

**def** printWord(word):  
 message = **"The word is: "** print(message + word)  
  
num1 = 5  
printWord(**"Dog"**)  
  
print(num1) *# no problem: scope of num1 is whole program*print(word) *# error: scope of word is only printWord() function*print(message) *# error: scope of message is only printWord() function*

Exercise 07-01

* Make a function called max\_num that has 3 parameters.
* It finds the largest number from the 3 parameters and returns this number.
* Make some variables x, y and z, and call the max\_num function with these parameters to test your code.
* This might help you:

a = 3, b = 5, c = 9  
  
**if** a is larger than b  
 largest is a  
**else** largest is b  
  
**if** c is larger than largest  
 largest is c  
  
**return** largest

Maximum number is 9

Exercise 07-02

* Make a function called calculate\_area\_rectangle that has 2 parameters, width and height.
  + It will **return** the **area** of the rectangle.
* Make a function called calculate\_perimeter\_rectangle that has 2 parameters, width and height.
  + It will **return** the **perimeter** of the rectangle.
* Make a function called calculate\_area\_circle that has 1 parameter, radius.
  + It will **return** the **area** of the circle.
* Make a function called calculate\_circumference that has 1 parameter, radius.
  + It will **return** the **circumference** of the circle.
* Make some variables for width, height and radius, and **call** your functions to test them.

Enter the width of the rectangle: 4

Enter the length of the rectangle: 5

Enter the radius of the circle: 5

The area of the rectangle is: 20

The perimeter of the rectangle is: 18

The area of the circle is: 78.525

The circumference of the circle is: 31.41

# Section 08: Classes, Objects and Encapsulation

### Object Oriented Programming

**Object Oriented Programming (OOP)** is good for large scale software development. OOP involves programming using **objects**.

The three main things involved in OOP are:

* Encapsulation
* Inheritance
* Polymorphism

### Objects

An **object** is a thing that can be identified. For example, a building, car, student or circle.

An object has:

* **State**: values of its **attributes** or **fields**. They are **descriptions of the object**.
* **Behaviour**: the **methods**. They are **what can be done** with the object. They are usually verbs.

Here is an example of an object:

**Dog**



**State**

Breed

Name

Sex

Colour

. . .

**Behaviour**

Sit

Bark

Sniff

Eat

. . .

### Classes

A **class** is a template, a blueprint, an abstraction that defines what the common fields and methods will be for the same type of **objects**. A class is not real. It is just a plan for an object. An object is real.

In Python, **objects** are made using **classes**. An object is an **instance** of a class. Objects and **instances** are same. This means an object is something real that comes from a class. An object has the real data but classes are the **definitions**.

An example is that you want to make some chairs.

* The diagram that shows how to build a chair and put it together is like a **class**. It is not a real chair. The diagram just tells you how to make a chair.
* Maybe you want to make 5 chairs like in the diagram. Each real chair is an **object**.
* Each chair has a different number written on it to identify each chair. Chair 3 is one **instance** of a chair object.



**Class**

***Diagram of chair***

**Object**

***Real chair***

chair3

### Encapsulation

**Encapsulation** means all the **properties** of an object, including **attributes (data)** and **functions** are grouped together. Objects can talk with each other, but objects normally are not allowed to know how other objects are **implemented** because implementation details are hidden inside the objects.

To understand this, think about how you drive a car. You do not need to know the engine works. The only knowledge needed is how to drive the car and what the car needs for it to go. The **encapsulation** of a car object would be the licence plate, registration, model, forward, reverse, stop, turn left, turn right… The information hidden from the driver is the how the engine works, how the brakes work.

Another example is a shape object. A rectangle object would calculate the area, perimeter, and the dimensions (length and width). However, the object would need the length and the width before it can do the calculations.

### Simple Class

This is a simple class. It does nothing. A class is made using the class keyword. The **pass** keyword means there is nothing in the class.

**class** Pet:  
 **pass**

### Attributes and Methods

Objects can have data using variables that belong to the object. Variables that belong to an object or class are called **attributes**. These are **descriptions of the object**.

Objects can also have functions that belong to a class. These functions are called **methods** of the class. These are what the **object can do**.

There are two types of **attributes**: **class variables** and **object variables.**

**Class variables** are owned by the class. They are shared because they can be used by all instances of that class (objects). There is only one copy of the **class variable**.

**Object variables** are owned by each individual object of the class. Each object has its own copy of the variable. The variable is not shared and are not related in any way to the variable of the same name in a different object.

A class is made using the class keyword. The fields and methods of the class are listed in an indented block.

Example

We can make a class for a pet. The **fields** are:

* name
* age

**class** Pet:  
 name = **""** number\_of\_legs = 0

These attributes are called **class variables**. **Class variables** can be used anywhere inside the class. Their **scope** is the class. **Local variables** can only be used in the method where they are made.

Making objects

If we have a class called Pet, and we want to make an object of the Pet class, we write this code:

**class** Pet:  
 name = **""** number\_of\_legs = 0  
  
my\_pet = Pet()  
my\_pet.number\_of\_legs = 4  
  
print(**"My pet has "** + str(my\_pet.number\_of\_legs) + **" legs"**)

My pet has 4 legs

* We have made a *Pet* class with 2 **class variables**.
  + We need to define values or we will get an error.
* We then make an object outside of the class.
  + **my\_pet** is an instance (object) of the *Pet* class
  + The name of the object is **my\_pet** and we make the object by using the class name and **()**
* Now that we have an object, we want to change its variables.
  + We write the object name **my\_pet** then a dot, then the name of the variable number\_of\_legs.
  + We can give the number\_of\_legs variable the value 4
* Now that we have set the variable, we can print out the variable using print()

We can make many objects from the Pet class and give their **class variables** different values:

**class** Pet:  
 name = **""** number\_of\_legs = 0  
  
my\_pet = Pet()  
my\_pet.number\_of\_legs = 4  
  
my\_pet2 = Pet()  
my\_pet2.number\_of\_legs = 2  
  
print(**"My first pet has "** + str(my\_pet.number\_of\_legs) + **" legs"**)  
print(**"My second pet has "** + str(my\_pet2.number\_of\_legs) + **" legs"**)

My first pet has 4 legs

My second pet has 2 legs

Each object has their own **class variables** which can have different values.

Example

We can make a class for a *Person*:

**class** Person:  
 **pass** *# Do nothing*p = Person()  
print(p)

<\_\_main\_\_.Person object at 0x000001C52AD1D2E8>

* We have made a **class** called *Person*
* To make the example easier, we just write **pass** which is like an empty block.
* Then we make an object or instance of the Person class.
  + The name of the object is **p** and we make the object by using the class name and **()**
* When we print **p**, it tells us we have a *Person* object in the \_\_main\_\_ module

Exercise 08-01

* Make a class called ***Person***
* Make a class variable called ***name*** and make it equal to ""
* Make a class variable called ***age*** and make it equal to 0
* Outside the class, make a ***Person*** object called ***sam***
* Make the variable ***name*** equal to “Sam” for the ***sam*** object
* Make the variable ***age*** equal to 20 for the ***sam*** object
* Make a ***Person*** object called ***james***
* Make the variable ***name*** equal to “James” for the ***james*** object
* Make the variable ***age*** equal to 21 for the ***james*** object
* Print out the ***name*** and ***age*** variables for the ***sam*** object
* Print out the ***name*** and ***age*** variables for the ***james*** object

Sam is 20

James is 21

The self

**Class methods** are a little different from ordinary functions. They must have an extra word that has to be added to the beginning of the parameter list. This word is **self**.

This **self** means the object that the method will do something on. If we have a Pet class, and we make a Pet object called **my\_pet**, then we call the **print\_name()** method, then this method needs to know that it will print the name inside the **my\_pet** object.

When you **call** the method, you do not give a value for this **self** parameter because Python will do it for you.

Class Methods

We have said that classes and objects can have **methods**, which are like **functions**, but methods are inside classes. These are called **Class Methods**. Another difference is that when we write **Class Methods**, we always need to use the **self** variable as an argument for the method:

Example

**class** Pet:

number\_of\_legs = 0  
  
 **def** sleep(self):  
 print(**"zzzzz"**)  
  
my\_pet = Pet()  
my\_pet.sleep()

zzzzz

When Python calls a method, it passes the current object **self** to that method as the first argument. When we call **my\_pet.sleep()** Python is going to pass the object **my\_pet** as an argument to the sleep method.

With a method, you always have to have an argument called **self** first in the list (if you want to add more arguments, you can add them afterwards, exactly like if you were passing many arguments to a function).

If you don’t include that argument, when you run the code, you will get an error.

If we want to use the method **sleep()**, we use an object of the **Pet** class. Just like the number\_of\_legs variable, we write the name of the object **my\_pet** , then a dot, then the name of the method with brackets (). We do not need to call **sleep()** using arguments, but Python is going to add in that argument by itself.

**Why do we need this self variable?**

We will make a new method called count\_legs() to print out the number of legs of the pet. Before, we got the number of legs using the number\_of\_legs variable from outside the class using my\_pet.number\_of\_legs. However, if a method **inside** the class wants to use the number\_of\_legs variable, we need to tell the class which object’s number\_of\_legs variable needs to be used. This is where we need to use the **self** variable:

**class** Pet:  
  
 number\_of\_legs = 0  
  
 **def** sleep(self):  
 print(**"zzzzz"**)  
  
 **def** count\_legs(self):  
 print(**"I have {0} legs"**.format(self.number\_of\_legs))  
  
my\_pet = Pet()  
my\_pet.number\_of\_legs = 4  
my\_pet.count\_legs()

I have 4 legs

Here, the **self** variable holds the my\_pet object. So when we call my\_pet.count\_legs() the **self** is replaced with my\_pet object. Now we can make another object:

your\_pet = Pet()  
your\_pet.number\_of\_legs = 8  
your\_pet.count\_legs()

I have 8 legs

Now, the **self** variable holds the your\_pet object. So when we call your\_pet.count\_legs() the **self** is replaced with your\_pet object.

Exercise 08-02

* Continue with Exercise 08-01
* Delete the code that prints out the people.
* In class ***Person*** add a new **Class Method** called message. This method will print “Hello”
* In class ***Person*** add a new **Class Method** called details. This method will print out “My name is … and I am … years old.”
* Outside class ***Person*** call the message method on the ***sam*** object.
* Call the details method on the ***sam*** object.
* Cal the message method on the ***james*** object.
* Call the details method on the ***james*** object.

Hello

My name is Sam and I am 20 years old.

Hello

My name is James and I am 21 years old.

The \_\_init\_\_ method

There are many method names which are special in Python classes.

The **\_\_init\_\_** method is run as soon as an object of a class is instantiated (made). There are 2 underscores \_ at the start and 2 underscores at the end of the word *init*. The purpose of the **\_\_init\_\_** method is to give values to the **object variables.**

**class** Pet:  
  
 **def** \_\_init\_\_(self, name):  
 self.name = name  
  
 **def** say\_hi(self):  
 print(**"Woof, I am a"**, self.name)  
  
p = Pet(**"Dog"**)  
p.say\_hi()

Woof, I am a Dog

* Here, we define the **\_\_init\_\_** method with 2 parameters. It needs the **self** parameter and we can give it our own parameter called **name**.
* Then we just make a new **object variable** also called **name**.
  + Notice these are two different variables even though they are both called 'name'.
  + There is no problem because the dotted field **self.name** means that there is something called *name* that is part of the object called *self* and the other *name* is a local variable.
* When we make the object **p**, of the class *Pet*, we use the class name, then the arguments in the brackets:
  + p = Pet(**"Dog"**)
* We do not need to call the **\_\_init\_\_** method.

Exercise 08-03

* Continue with Exercise 08-02
* Delete the **Class Variables**

name = **""**

age = 0

* Delete the code that makes our Person objects:

sam = Person()  
sam.name = **"Sam"**sam.age = 20  
  
james = Person()  
james.name = **"James"**james.age = 21

* In class ***Person*** make the **\_\_init\_\_** method:
  + It will have 2 parameters: name and age
  + It will set the object variables self.name and self.age
* Outside class ***Person*** make the sam object and the james object with the arguments:
  + “Sam”, 20
  + “James”, 21

Hello

My name is Sam and I am 20 years old.

Hello

My name is James and I am 21 years old.

Access Modifiers

Most object-oriented programming languages have some **access control**.

**Access Modifiers** control how the variables and methods of a class can be accessed (used). They are:

* **Public**: the variables and methods can be used anywhere.
* **Private**: the variables and methods can be used only in the class, not outside the class.
* **Protected**: the variables and methods can be used only in the class, and subclasses (later).

**public**

Python doesn't use **private** or **protected**. Python doesn't believe in making rules like this. Instead, it suggests you use best practices. If we want to suggest that a method should not be used publicly, we can put a note in the comments to say that the method is private.

By default, all variables and methods are **Public** in Python. So to make something **Public**, you don’t have to do anything. If you really want to make some things hidden from outside the object in Python, you can make the changes below.

**Private**

Variables can be **Private**, which can be useful. A **Private** variable can only be changed inside a **class method** and not outside of the class. To make a variable or method **Private**, you must write 2 underscores **\_\_** before the name:

**class** Pet:  
  
 **def** \_\_init\_\_(self, name, age):  
 self.\_\_name = name *# private variable* self.age = age *# public variable*myPet = Pet(**"Sam"**, 5)  
print(myPet.age)  
print(myPet.\_\_name)

5

AttributeError: 'Pet' object has no attribute '\_\_name'

You can see we have an error. We cannot use the \_\_name variable because it is **Private**.

However, if we really want to use the \_\_name variable from outside the class we can do a “hack”:

**class** Pet:  
  
 **def** \_\_init\_\_(self, name, age):  
 self.\_\_name = name *# private variable* self.age = age *# public variable*myPet = Pet(**"Sam"**, 5)  
myPet.\_Pet\_\_name = **"Max"**print(myPet.\_Pet\_\_name)

Max

So you can see if you really want to change or print the private \_\_name variable from outside the class, you can use this:

<object\_name>.**\_**<class\_name>**\_\_**<variable>

This means that nothing in Python is really **private**.

Here is a **Private** method called \_\_petFeeling() which has 2 underscores:

**class** Pet:  
  
 **def** \_\_init\_\_(self, name, age):  
 self.\_\_name = name *# private* self.age = age *# public* **def** \_\_petFeeling(self): *# private method*  
 print(**"Happy"**)  
  
myPet = Pet(**"Sam"**, 5)  
  
myPet.\_\_petFeeling()

AttributeError: 'Pet' object has no attribute '\_\_petFeeling'

You can see that this also gives an error because the \_\_petFeeling() method is **Private**.

**Protected**

**Protected** variables and methods are very similar to **Private** ones. You probably will not use **Protected** variables or methods very often. A variable that is **Protected** can only be accessed by its own class and any classes derived from it.

Protected variables begin with 1 underscore.

**class** Pet:  
  
 **def** \_\_init\_\_(self, name, age):  
 self.\_name = name *# protected variable* self.age = age *# public variable*myPet = Pet(**"Sam"**, 5)  
  
print(myPet.\_name)

Sam

You can see that there is no error here. The **Protected** variable does not behave the same as a **Private** variable. Because the **Protected** variable has an underscore, it shows that it should not be used like the example above.

So you can see that if you want to show that a variable or method is private or protected, you can use underscores. But because they can be “hacked”, they can only be used as a suggestion.

**Be careful!**

You are allowed to make new object variables from outside the class. Sometimes it looks like you are changing the private variable but you are actually making a new **object variable**.

**class** Pet:  
 **def** \_\_init\_\_(self, age):  
 self.\_\_age = age  
  
 **def** get\_age(self):  
 **return** self.\_\_age  
  
  
p = Pet(4)  
  
*# This makes a new object variable called age  
# This is different to \_\_age*p.age = 10  
print(p.age)  
  
*# This will print the real \_\_age variable*print(p.get\_age())

10

4

Exercise 08-04

* Continue with Exercise 08-03
* Delete this code:

sam.message()  
sam.details()  
  
james.message()  
james.details()

* Make the **name** **object variable** **private** using 2 underscores \_\_
* Make the **age** **object variable** **private** using 2 underscores \_\_
* You will need to change the code in the details method.
* Try to print Sam’s name using print(sam.name)
* Use the “hack” to print Sam’s name

AttributeError: 'Person' object has no attribute 'name'

Sam

Properties

Some programming languages use code to **get** and **set** private **object** **variables**. If you want to use the private object variables outside the class, you can use **public** methods called **Getters** and **Setters** to see them. This is **Encapsulation**.

Let’s look at an example. We have a class called Pet, that has an **object variable** called age:

**class** Pet:  
 **def** \_\_init\_\_(self, age):  
 self.age = age  
  
p = Pet(3)  
print(p.age)

3

What if someone tries to set the age to a number less than 0 or greater than 100? We want to show an error. So we need to change our code:

**class** Pet:  
 **def** \_\_init\_\_(self, age):  
 self.age = age  
  
 **def** get\_age(self):  
 **return** self.age  
  
 **def** set\_age(self, value):  
 **if** value < 0 **or** value > 100:  
 print(**"Error - age is wrong"**)  
 **else**:  
 self.age = value  
  
p = Pet(4)  
p.age = 144  
print(p.age)

144

But the user can still set an age greater than 100. We need to make age **private**:

**class** Pet:  
 **def** \_\_init\_\_(self, age):  
 self.set\_age(age)  
  
 **def** get\_age(self):  
 **return** self.\_\_age  
  
 **def** set\_age(self, value):  
 **if** value < 0 **or** value > 100:  
 print(**"Error - age is wrong"**)  
 **else**:  
 self.\_\_age = value  
  
p = Pet(4)  
print(p.get\_age())  
  
p.set\_age(8)  
print(p.get\_age())  
  
p.set\_age(144)  
print(p.get\_age())

4

8

Error - age is wrong

8

So now we must use get\_age() and set\_age() to get and set the **age**. You will notice in the \_\_init\_\_ method, instead of setting self.\_\_age = age, we use the new set\_age() method to set the age.

* The **getter** is called get\_age. It has the **self** parameter (like all **class methods**).
* It returns the **private** \_\_age variable.
* The **setter** is called set\_age. It has the **self** parameter (like all **class methods**), and it has a parameter called value that we will use to change the **private** \_\_age variable.
* It sets the **private** \_\_age variable to the value variable.
* First we make a new *Pet* object with **5**
* We print the **private** \_\_age variable using the get\_age() **getter**, and it shows **4**
* Then we set the **private** \_\_age variable to **8** using the set\_age() **setter**
* Then we print the **private** \_\_age variable using the get\_age() **getter**, and it shows **8**
* If we try to set the **private** \_\_age variable to **144** using the set\_age() **setter**, we get an error because it is too large.
* Then we print the **private** \_\_age variable using the get\_age() **getter**, and it still shows **8**

But now we have a problem. Maybe our program was used by many programmers.

To get the age, they now cannot use:

p.age

They must use:

p.get\_age()

To set the age, they cannot use:

p.age = 5

They must use:

p.set\_age(5)

So maybe their code is broken. What can we do? We can add a **Property**:

**class** Pet:  
 **def** \_\_init\_\_(self, age):  
 self.set\_age(age)

**def** get\_age(self):  
 print(**"Getting the age"**)  
 **return** self.\_\_age  
  
 **def** set\_age(self, value):  
 print(**"Setting the age"**)  
 **if** value < 0 **or** value > 100:  
 print(**"Error - age is wrong"**)  
 **else**:  
 self.\_\_age = value  
  
 age = property(get\_age, set\_age)  
  
p = Pet(5)  
p.age = 7  
print(p.age)

Setting the age

Getting the age

7

A print() function was put inside get\_age() and set\_age() to see what is happening.

The last line of the code:

age = property(get\_age, set\_age)

makes a **property** called age. The **property** makes the get\_age() and set\_age() be used if we try to get or set the age.

If we write:

print(p.age)

then get\_age() will be called.

If we write:

p.age = 7

then set\_age() will be called.

So now the code of other developers is not broken. We can make changes to get\_age() and set\_age() without breaking the developers’ code.

Decorators

We can make a **property** in another way. We can use a **decorator** which tells Python that the code is a **property**.

To make a **get** method:

@property  
**def <variable\_name>**(self):  
 **return** self.\_\_**<variable\_name>**

To make a **set** method:

@**<variable\_name>**.setter  
**def <variable\_name>**(self, value):  
 self.\_\_**<variable\_name>** = value

So we can change our code to use the @property **decorator**:

**class** Pet:  
 **def** \_\_init\_\_(self, age):  
 self.age = age  
  
 @property  
 **def** age(self):  
 print(**"Getting the age"**)  
 **return** self.\_\_age  
  
 @age.setter   
 **def** age(self, value):  
 print(**"Setting the age"**)  
 **if** value < 0 **or** value > 100:

print(**"Error - age is wrong"**)  
 self.\_\_age = 0  
 **else**:  
 self.\_\_age = value  
  
p = Pet(5)  
p.age = 7  
print(p.age)

Setting the age

Getting the age

7

Exercise 08-05

* Continue with Exercise 08-04
* Delete this code:

print(sam.\_Person\_\_name)

* Make a method get\_name() that returns the \_\_name variable.
* Make a method set\_name() that has an extra parameter called value.
  + It first checks if value is equal to ""
    - If value is equal to "", then print an error message
    - Else set the \_\_name variable to value.
* Make a method get\_age() that returns the \_\_age variable.
* Make a method set\_age() that has an extra parameter called value.
  + It first checks if value is less than 0 or greater than 100
    - If value is less than 0 or greater than 100, then print an error message
    - Else set the \_\_age variable to value.
* Make a **property** for name:
  + name = property(...
* Make a **property** for age
* Outside the ***Person*** class, make sure you are making an object called sam with the name “Sam” and the age 20.
* Print the **name** of the sam object using sam.name
* Set the **age** of the sam object to 30 using sam.age = 30
* Print the **age** of the sam object
* Try to set the **age** of the sam object to 130
* Print the **age** of the sam object

Sam

30

Error - wrong age

30

Exercise 08-06

* Continue with Exercise 08-05
* You need to use the **property** **decorators**
* Change get\_name() to a **property** by changing the name of the method to name and putting @property above it.
* Change set\_name() to a **property** by changing the name of the method to name and putting @name.setter above it.
* Change get\_age() to a **property**.
* Change set\_age() to a **property**.
* You can delete:
  + name = property(get\_name, set\_name)
  + age = property(get\_age, set\_age)
* Test your code.

Sam

30

Error - wrong age

30

Class Methods and Class Variables

Remember there are 2 types of **fields**: **Class variables** and **Object variables**. We have mainly used **Object Variables**. **Class Variables** can be shared by all objects of that class. If an object changes the value of a **Class Variable**, then all other objects will see that change. **Object Variables** are owned only by each object.

If we want to use a **Class Variable** inside the class, we need to write the **name** of the class first, then a **dot**, then the **name** of the **Class Variable**.

Example

**class** Robot:  
  
 *# A class variable, counting the number of robots* population = 0  
  
 **def** \_\_init\_\_(self, name):  
 *# Initializes the data* self.name = name  
 print(**"Making {0}"**.format(self.name))  
  
 *# When this robot is made, the robot  
 # population increases by 1* Robot.population += 1  
  
 **def** die(self):  
 *# Destroy robot* print(**"{0} is being destroyed"**.format(self.name))  
  
 Robot.population -= 1  
  
 **def** say\_hi(self):  
 print(**"Hello, my name is {0}."**.format(self.name))  
  
  
robot1 = Robot(**"R2-D2"**)  
robot1.say\_hi()  
  
robot2 = Robot(**"C-3PO"**)  
robot2.say\_hi()  
  
print(Robot.population)  
print(robot1.population)  
print(robot2.population)  
  
Robot.population = 66  
  
print(Robot.population)  
print(robot1.population)  
print(robot2.population)  
  
print(**"Robots have finished their work. So let's destroy them."**)

robot1.die()  
robot2.die()  
  
print(Robot.population)

Making R2-D2

Hello, my name is R2-D2.

Making C-3PO

Hello, my name is C-3PO.

2

2

2

66

66

66

Robots have finished their work. So let's destroy them.

R2-D2 is being destroyed

C-3PO is being destroyed

64

Here, population belongs to the ***Robot*** class and so it is a **class variable**. The name variable belongs to the **object** and so it is an **object variable**.

So when we use the population **class variable**, we write Robot.population and not self.population.

When we use the **object variable** name, we write self.name in the methods of that object. Remember this simple difference between class and object variables.

You can see that the **\_\_init\_\_** method is used to make the ***Robot*** object with a name variable. In this method, we increase the population count by 1, since we have one more robot being added.

You can see that the values of self.name are specific to each object. This means each object has a different *name* because it is an **object variable**.

In the die() method, we decrease the Robot.population **class variable** by 1.

Class Methods

We can also use **Class Methods**.

We can define a **method** as a **Class Method** using the code:

@classmethod

Why do we need **Class Methods**? **Class Methods** are for when you need to have methods that can change something for all objects, not just one object.

**class** Robot:  
  
 *# A class variable, counting the number of robots* population = 0  
  
 **def** \_\_init\_\_(self, name):  
 *# Initializes the data* self.name = name  
 print(**"Making {0}"**.format(self.name))  
  
 *# When this robot is made, the robot  
 # population increases by 1* Robot.population += 1  
  
 **def** die(self):  
 *# Destroy robot* print(**"{0} is being destroyed"**.format(self.name))  
  
 Robot.population -= 1  
  
 **def** say\_hi(self):  
 print(**"Hello, my name is {0}."**.format(self.name))  
  
 @classmethod  
 **def** count\_robots(cls):  
 *# Prints the current population* print(**"We have {0} robots."**.format(cls.population))  
  
  
robot1 = Robot(**"Iron Man"**)  
robot1.say\_hi()  
Robot.count\_robots()  
  
robot2 = Robot(**"Ultron"**)  
robot2.say\_hi()  
Robot.count\_robots()  
  
print(**"\nRobots have finished their work. So let's destroy them."**)  
robot1.die()  
robot2.die()  
  
Robot.count\_robots()

Making R2-D2

Hello, my name is R2-D2.

We have 1 robots.

Making C-3PO

Hello, my name is C-3PO.

We have 2 robots.

Robots have finished their work. So let's destroy them.

R2-D2 is being destroyed

C-3PO is being destroyed

We have 0 robots.

The count\_robots() method is actually a method that belongs to the class and not to the object.

We have made the count\_robots() method a **Class Method** using a **decorator**. A decorator tells the code that the method is a **Class Method**.

When you make a **Class Method**, you need to write the **decorator** and give it the class variable called cls. Inside the method, if you want to change a **Class Variable**, you need to write cls and then a **dot** and then the **name** of the **Class Variable**:

@classmethod  
**def** count\_robots(cls):  
 *# Prints the current population* print(**"We have {0} robots."**.format(cls.population))

If we want to call this **Class Method**, we need to write the class **name**, then a **dot**, then the **name** of the **Class Method**:

Robot.count\_robots()

We have 2 robots.

Static Methods

**Static Methods** are like **Instance Methods** and **Class Methods**, but they do not need a self or cls parameter. They do not have to have any parameters because they do not know any information about the class or an object.

If you do not know if a method should be an **Instance** Method, **Class Method** or a **Static Method**, look at the parameters:

* If one of the parameters is self, then it should be a **Instance Method**.
* If one of the parameters is cls, then it should be a **Class Method**.
* If there is no self or cls parameter, it should be a **Static Method**.

So **Static Methods** are used to change something in the class that all objects can see, but **Static Methods** do not need to use the class or object. An example for the ***Pet*** class is a method to calculate the *weight* of the pet. Maybe the *weight* of the pet is in pounds, not kilograms, so we need to convert it. We do not need a class or object parameter. We just need the weight in pounds:

**class** Pet:  
  
 @staticmethod  
 **def** pounds\_to\_kg(pounds):  
 **return** pounds / 2.205  
  
print(Pet.pounds\_to\_kg(4))

1.8140589569160996

You can see that we use the **decorator** @staticmethod and we call the method using the name of the class. You can also see that we do not even need to make an object first. We can call the **Static Method** without making an object.

Instance Methods, Class Methods and Static Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Type of Method** | **Parameter** | **Parameter name** | **Example** |
| Instance Method | An object | self | **def** say\_hi(self):  print(**"Hi, I am"** + self.name) |
| myPet.say\_hi() |
| Class Method | A class | cls | number\_of\_pets = 0  @classmethod **def** increasePets(cls, num1):  cls.number\_of\_pets += num1 |
| Pet.increasePets(5) |
| Static Method | – | – | @staticmethod **def** pounds\_to\_kg(pounds):  **return** pounds / 2.205 |
| print(Pet.pounds\_to\_kg(4)) |

Exercise 08-07

* Continue with your code from Exercise 08-06
* Make a **class variable** called number\_of\_people and make it equal to 0
* Inside the **\_\_init\_\_** method:
  + Increase the number\_of\_peoplevariable by 1
  + Print “Added 1 person”
* Change the message()method to a **static method**
* Make a **class method** called count\_population() that has the **cls** parameter
  + It will print out “There are ? people” using the **class variable** number\_of\_people to print out the number of people.
* Outside the delete all of the code.
* **Call** the **static method** message()
* Make a new *Person* object with the name “Tim” and age 29
* Call the class method count\_population()
* Make a new *Person* object with the name “Alice” and age 25
* Call the class method count\_population()

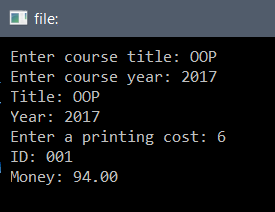
Hello

Added 1 person

There are 1 people

Added 1 person

There are 2 people



# Section 09: Inheritance

### Inheritance

Inheritance in the real world can mean passing on something to your children. If parents have brown hair, then the children will probably inherit the brown hair. The children *share* the same features of the parents.

In programming, **Inheritance** means you can make one class, then make another class that uses the same fields and methods of the first class. The second class *inherits* from the first class.

Think about an example with a *Pet Shop*. A class made for *Pet* (宠物) might have the variables *ID, number of legs,* and *species* (种类)*.* Then if a *Mammal* (哺乳动物) class was made, it can use these variables. The *Mammal* class can **inherit** from the *Pet* class. The *ID* could be “Max”,and *number of legs* would be 4. The *Mammal* class does not need to use all of the variables and methods of the *Pet* class. The *Mammal* class can have its own variables and methods.

Inheriting Classes

**Inheritance** lets us make new classes based on another class.

Dog

+ string : name

+ int : age

+ \_\_init\_\_()

Pet

+ string : name

+ int : age

+ \_\_init\_\_()

Base class

Sub class

**class** Pet:  
  
 **def** \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age

**class** Dog(Pet):  
 **pass** *# do nothing*

my\_dog = Dog(**"Sam"**, 5)   
print(my\_dog.name)

Sam

In the example, ***Pet*** is the **super class, base class** or **parent class**.

The class ***Dog***is a **subclass** or **child** class, because it inherits the ***Pet*** class.

We will use the words **super class** and **sub class**. To make a sub class inherit from a super class, we write the **super class** in round brackets.

* We made the ***Dog*** class inherit the ***Pet*** class
* The **pass** keyword means we will write some more code later.
* Because we inherited from the ***Pet*** class, we inherited all of the code from the ***Pet*** class
  + The ***Dog*** class will inherit all of the variables and methods of the ***Pet*** class
* So now we can make an object from the ***Dog*** class and use the name **object** **variable** even though this method is in the ***Pet*** **super class**.
* The ***Dog*** class does not have an \_\_init\_\_ method, but that is OK.
  + Python automatically looked at the super class for the \_\_init\_\_ method and used that method.

Many classes can inherit the ***Pet*** class:

Dog

+ string : name

+ int : age

+ \_\_init\_\_()

Bird

+ string : name

+ int : age

+ \_\_init\_\_()

Pet

+ string : name

+ int : age

+ \_\_init\_\_()

Sub classes

Base class

**class** Pet:  
  
 **def** \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
  
**class** Dog(Pet):  
 **pass** *# do nothing***class** Bird(Pet):  
 **pass** *# do nothing*my\_dog = Dog(**"Sam"**, 5)  
print(my\_dog.name)  
  
my\_bird = Bird(**"Jack"**, 3)  
print(my\_bird.name)

Sam

Jack

Adding functionalities to sub classes

**Sub classes** can have their own **object variables** and **methods** that the **super class** and other classes do not have:

Dog

+ string : name

+ int : age

+ int : weight

+ \_\_init\_\_()

Bird

+ string : name

+ int : age

+ string : colour

+ \_\_init\_\_()

Sub classes

Base class

Pet

+ string : name

+ int : age

+ \_\_init\_\_()

**class** Pet:  
  
 **def** \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age

**class** Dog(Pet):  
  
 **def** \_\_init\_\_(self, name, age, weight):  
 Pet.\_\_init\_\_(self, name, age)

self.weight = weight  
  
worker1  
  
**class** Bird(Pet):  
  
 **def** \_\_init\_\_(self, name, age, colour):  
 Pet.\_\_init\_\_(self, name, age)  
 self.colour = colour  
  
  
myDog = Dog(**"Sam"**, 5, 23)  
print(myDog.name)  
print(myDog.weight)  
  
myBird = Bird(**"Jack"**, 3, **"Green"**)  
print(myBird.name)  
print(myBird.colour)

Sam

23

Jack

Green

* We have to make the **\_\_init\_\_** method in the sub classes:
  + You can copy top line of the **\_\_init\_\_** method from the ***Pet*** **super class** and put in the new parameter for the **sub class** called weight
  + Then we want to bring in the same code from the **\_\_init\_\_** method of the ***Pet*** **super class**.
  + To do this, we don’t need to copy the code from the ***Pet*** class.
  + We just write the super class name (***Pet***) then a dot, then **\_\_init\_\_** and the same parameters as the **\_\_init\_\_** method of the ***Pet*** class.
  + Then we can make the **object variable** weight
* So when we make the my\_dog object, we need to give it a weight value.
* We can now use the name variable from the ***Pet* super class** and the weight variable from the ***Dog* sub class**.

Exercise 09-01

There is a **super class** called ***Person*** and **sub classes** called ***Student*** and ***Worker***.

**class** Person:  
  
 **def** \_\_init\_\_(self, name):  
 self.name = name  
  
 <Make a method say\_name>  
 <It will print "Hi, my name is ...">

**class** Student(Person):  
  
 **def** \_\_init\_\_(self, name, id):  
 Person.\_\_init\_\_(self, name)  
 self.id = id  
  
  
<Make a class called Worker that inherits the Person class>  
  
 <Make the \_\_init\_\_ method with a new parameter called salary>  
 <Call the \_\_init\_\_ method of the Person class>  
 <Make the object variable for salary>  
  
  
student1 = Student(**"James"**, **"2016A1234"**)  
print(student1.name)  
print(student1.id)  
<Call the say\_name() method on the student1 object>  
  
<Make a worker object called worker1 with the name "Max" and a salary of 30000>  
<Print out the name of the worker>  
<Print out the salary of the worker>  
<Call the say\_name() method on the worker1 object>

James

2016A1234

Hi, my name is James

Max

30000

Hi, my name is Max

Specialization and Generalization

When we make classes, we can choose to use **Specialization** or **Generalization**.

**Specialization** is when a very general class is made with not much detail. This is our **Pet** class.

Then we make more detailed, specialized classes like **Mammal** and **Bird**. They are **sub classes** and they have extra variables and methods like **weight** and **colour**. These sub classes inherit the members of the super class. This is called **specialization**.

**General class**

**Specialized classes**

Specialization makes an **IS-A relationship** between the super class and the sub classes. For example, a **Mammal** **IS-A** **Pet**.

**Generalization** is when a group of classes are made first. For example, **Rectangle**, **Triangle** and **Circle**. The developer sees things that are common in all the classes and makes a **super class** like Shape to put in the common things. This is **generalization**.

Rectangle

- double: height

- double: width

- double: area

+ double: Height

+ double: Width

+ double: Area

**Shape**

Circle

- double: radius

- double: area

+ double: Radius

+ double: Area

Triangle

- double: height

- double: baseline

- double: area

+ double: Height

+ double: Baseline

+ double: Area

Multiple Inheritance in Python

* A **super class** can have many **sub classes**:

Super class

Sub class

Sub class

Sub class

* A **sub class** can have many **super classes**. This is called **Multiple Inheritance**:

Super class

Super class

Sub class

* Multiple inheritance is possible in Python, but not possible in other languages like C#.
* The diamond-problem:

A

C

B

D

* Class A has a method called *Display()* and both class B and class C inherit class A. They have a method *Display()* that **overrides** the method in class A. Class D inherits from class B and class C. If class D wants to use *Display()*, which version of *Display()* it use, the one from class B or the one from class C?

Example of Multiple Inheritance

* We have an ***Animal*** class with an **object variable** called number\_of\_legs
* There is a ***Pet*** class with an **object variable** called owner\_name
* Now we have a ***Dog*** class. A dog is both an Animal and a Pet, so it can **inherit** from both the ***Animal*** class and the ***Pet*** class.
* The ***Dog*** class can have its own **object variables**. It has an **object variable** called age.

**Pet**

**Animal**

**Dog**

**class** Animal:  
  
 **def** \_\_init\_\_(self, number\_of\_legs):  
 self.number\_of\_legs = number\_of\_legs  
  
  
**class** Pet:  
  
 **def** \_\_init\_\_(self, owner\_name):  
 self.owner\_name = owner\_name  
  
  
**class** Dog(Animal, Pet):  
  
 **def** \_\_init\_\_(self, number\_of\_legs, owner\_name, age):  
 Animal.\_\_init\_\_(self, number\_of\_legs)  
 Pet.\_\_init\_\_(self, owner\_name)  
 self.age = age  
  
  
my\_dog = Dog(4, **"John"**, 2)  
print(my\_dog.number\_of\_legs)  
print(my\_dog.owner\_name)  
print(my\_dog.age)

4

John

2

* When we have **multiple inheritance**, we list the classes we want to inherit in the round brackets:

**class** Dog(Animal, Pet):

* The **\_\_init\_\_** method of the ***Dog*** class must have all of the parameters from both of its **super classes**.
* To bring in the **\_\_init\_\_** method from the ***Animal*** class, we write the name of the **super class** ***Animal*** then a dot, then **\_\_init\_\_** and the same parameters as the **\_\_init\_\_** method of ***Animal*** class.
* To bring in the **\_\_init\_\_** method from the ***Pet*** class, we write the name of the **super class** ***Pet*** then a dot, then **\_\_init\_\_** and the same parameters as the **\_\_init\_\_** method of ***Pet*** class.
* Then we make the **object variable** age
* We can use the ***Dog*** object my\_dog to get the super class variables:
  + owner\_name from the ***Animal*** **super class**
  + number\_of\_legs from the ***Pet*** **super class**

Private, public and protected

Private

**class** Super\_class:  
  
 **def** \_\_private\_method(self):  
 print(**"Private"**) *# Can be used in this class only* **def** public\_method(self):  
 self.\_\_private\_method() *# No problem***class** Sub\_class(Super\_class):  
  
 **def** some\_method(self):  
 self.\_\_private\_method() *# Error - cannot see \_\_private\_method()  
 # because it is private in the Super\_class*sc = Super\_class()  
sc.public\_method() *# No problem*  
sc.\_\_private\_method() *# Error - cannot see \_\_private\_method()  
 # because it is private in the Super\_class*

super\_c = Super\_class()  
super\_c.public\_method() *# No problem*super\_c.\_\_private\_method() *# Error - cannot see \_\_private\_method()  
 # because it is private in the Super\_class*sub\_c = Sub\_class()  
sub\_c.some\_method() *# Error - cannot see \_\_private\_method()*

Public

**class** Super\_class:  
  
 **def** public\_method(self):  
 print(**"Public"**) *# Can be used in this class and everywhere* **def** another\_method(self):  
 self.public\_method() *# No problem***class** Sub\_class(Super\_class):  
  
 **def** some\_method(self):  
 self.public\_method() *# No problem*super\_c = Super\_class()  
super\_c.public\_method() *# No problem*super\_c.another\_method() *# No problem*sub\_c = Sub\_class()  
sub\_c.some\_method() *# No problem*

The table below shows where members can be seen:

|  |  |  |  |
| --- | --- | --- | --- |
| Modifiers | Class | Subclass | World |
| public | Y | Y | Y |
| private | Y | **N** | **N** |

Exercise 09-02

There are **super classes** called ***Computer*** and ***Phone****,* and a **sub class** called ***Smartphone***.

**Phone**

**Computer**

**Smartphone**

**class** Computer:  
  
 **def** \_\_init\_\_(self, company):  
 self.\_\_company = company  
  
 <Make a PRIVATE method called \_\_private\_say\_company>  
 <Print "The company is called " and the name of the company>

<Make a PUBLIC method called public\_say\_company()>  
 <Call \_\_private\_say\_company()>

<Make a class Phone>  
  
 <Write the \_\_init\_\_ method with a parameter called phone\_number>  
 <Make a private object variable \_\_phone\_number and set it to phone\_number>  
  
 <Make a method called get\_phone\_number() that returns phone\_number>  
  
  
<Make a class Smartphone that inherits Computer and Phone>  
  
 <Make the \_\_init\_\_ method using the parameters from Computer and Phone, and it also has a parameter called name>

<Call the Computer \_\_init\_\_ method>

<Call the Phone \_\_init\_\_ method>

<Make the private object variable called \_\_name>

<Make a method called get\_name() that returns \_\_name>  
  
  
<Make a Smartphone object called sp1 with the company "Apple", phone number of 15257918341, and name of "iPhone X">

<Find a way to print out the company>

<Print out the phone number and the name>

The company is Apple

15257918341

iPhone X

Exercise 09-03 OPTIONAL

Use the UML diagram below to make a program. It should print out the text shown below if you use the code below to make the objects.

Shape

+ string : colour

+ \_\_init\_\_()

Triangle

+ int : width

+ int : height

+ \_\_init\_\_()  
+ float: calculate\_area()

Rectangle

+ int : width

+ int : height

+ \_\_init\_\_()

+ float: calculate\_area()

rec1 = Rectangle(**"red"**, 4, 5)  
print(rec1.colour)  
print(rec1.calculate\_area())  
  
tri1 = Triangle(**"blue"**, 6, 8)  
print(tri1.colour)  
print(tri1.calculate\_area())

red

20

blue

24.0

# Section 10: Polymorphism and Abstract

### Polymorphism

The word **Polymorphism** means having many forms.

A class defines a **type**. A sub class defines a **sub type**. A super class defines a **super type**.

Polymorphism is making a class to be used by sub classes that have different types.

Example

Fruit can be eaten, but different types of fruit are eaten in different ways. An apple, which is a fruit, can be eaten (because it is a fruit). A banana can also be eaten (because it is also a fruit), but in a different way from an apple. You peel it first.

So if you have a super class called *Fruit* which has a method called *eat(),* then you can have sub classes called *Apple* and *Banana.* The method *eat()* in the sub class *Apple* will be different to the method *eat()* in *Banana.*

Fruit

+ Eat()

Banana

+ Eat()

Base class

Sub classes

Apple

+ Eat()

**class** Fruit:  
 *# code ...***class** Apple(Fruit):  
 *# code ...***class** Banana(Fruit):  
 *# code ...*f1 = Fruit()  
f2 = Apple()  
f3 = Banana()

Method Overriding

Method Overriding is when there is a method in the super class, and the sub classes inherit this method but they change the method. Changing a method is called overriding the method.

Fruit

+ Eat()

Banana

+ Eat()

Base class

Sub classes

Apple

+ Eat()

In the code below, there is **no** **Method Overriding**. The same area() method is inherited by the *Rectangle* class and the *Circle* class.

**class** Shape:  
  
 **def** \_\_init\_\_(self, colour):  
 self.colour = colour  
  
 **def** area(self):  
 **return** 2  
  
  
**class** Rectangle(Shape):  
  
 **def** \_\_init\_\_(self, colour, length, width):  
 Shape.\_\_init\_\_(self, colour)  
 self.length = length  
 self.width = width  
  
  
**class** Circle(Shape):  
  
 **def** \_\_init\_\_(self, colour, radius):  
 Shape.\_\_init\_\_(self, colour)  
 self.radius = radius  
  
  
s1 = Rectangle(**"red"**, 5, 4)  
print(s1.area())  
  
s2 = Circle(**"blue"**, 7)  
print(s2.area())

Now we can change the method area() in the sub classes to make the correct calculations. The methods have been changed to calculate the correct area of a rectangle and a circle. This is **Method Overriding**.

**import** math  
  
**class** Shape:  
  
 **def** \_\_init\_\_(self, colour):  
 self.colour = colour  
  
 **def** area(self):  
 **return** 2  
  
  
**class** Rectangle(Shape):  
  
 **def** \_\_init\_\_(self, colour, length, width):  
 Shape.\_\_init\_\_(self, colour)  
 self.length = length  
 self.width = width  
  
 **def** area(self):  
 **return** self.length \* self.width  
  
  
**class** Circle(Shape):  
  
 **def** \_\_init\_\_(self, colour, radius):  
 Shape.\_\_init\_\_(self, colour)  
 self.radius = radius  
  
 **def** area(self):  
 **return** math.pi \* self.radius \* self.radius  
  
  
s1 = Rectangle(**"red"**, 5, 4)  
print(s1.area())  
  
s2 = Circle(**"blue"**, 7)  
print(s2.area())

So the object s1 will call the area() method in the ***Rectangle*** class and the object s2 will call the area()method in the ***Circle*** class.

So all 3 classes have an area() method, but we have used **Method Overriding** to make specific area calculations for each type of shape.

Exercise 10-01

**class** Person:  
  
 **def** \_\_init\_\_(self, name):  
 self.name = name  
  
 <Make a method called greeting() that prints "Hello">  
  
 <Make a method called display\_info() that prints out "Name: ... + the name variable>  
  
  
**class** Customer(Person):  
  
 <Make the \_\_init\_\_ method with name and age>  
 <Call the Person \_\_init\_\_ method>  
 <Set the age object variable>  
  
 <Make a method called greeting() that prints "Dear customers, I am ?? years old" with the age variable>  
  
  
<Make a customer object with the name "John Smith" and the age 20>

<Call the greeting() method>

<Call the display\_info() method>

Dear customers, I am 20 years old

Name: John Smith

Exercise 10-02

Using the same format as Exercise 10-01, make these classes:

Class Animal (super class)

* The \_\_init\_\_ method has 1 parameter called **type**. It has an **object variable** called **type**
* It has a method called **sound()**. This method prints out “No sound”.
* It has a method called **display\_info()** which prints out “Type: …” and **type**.

Class Dog (sub class)

* This class inherits from class **Animal**
* The \_\_init\_\_ method has the parameter **type** and it has an **object variable** called **weight**
* It has a method called **sound()**. This method prints out “Woof”.

Class Cat (sub class)

* This class inherits from class **Animal**
* The \_\_init\_\_ method has the parameter **type** and it has an **object variable** called **colour**
* It has a method called **sound()**. This method prints out “Meow”.

Outside the classes

* Make a **Dog** object called **my\_dog**, using the parameters “Labrador” and 22
* Call the **sound()** method called using **my\_dog**
* Call the **display\_info()** method called using **my\_dog**
* Make a **Cat** object called **my\_cat**, using the parameters “Siamese” and “white”
* Call the **sound()** method called using **my\_cat**
* Call the **display\_info()** method called using **my\_cat**

Woof

Type: Labrador

Meow

Type: Siamese

Abstract Classes

**Abstract** classes are similar to **super classes** but abstract classes do not have **instances**. Abstract classes are not instantiated, which means you would never create an **object** of an abstract class. They are made so that all of the sub classes that inherit them are similar.

For example, you may have a *Shape* super class that has **sub classes** like *Circle*, *Rectangle* and *Triangle*. The *Shape* class is not useful and so we would not make an object of class *Shape*.

Abstract classes can be instantiated with non-abstract classes only. Abstract classes have **abstract methods** which MUST be overridden by the sub classes.

An abstract method has no implementation (no code inside them). Sub classes must override the method and write the code. Abstract methods can only be in abstract classes.

To make an abstract class in Python, we need a module called ABC (Abstract Base Classes). The super class must inherit this module. To make a method abstract, we need to use the decorator @abstractmethod

**from** abc **import** ABC, abstractmethod  
  
**class** Shape(ABC):  
  
 @abstractmethod  
 **def** say\_name(self):  
 **pass  
  
  
class** Circle(Shape):  
 **pass**s1 = Shape()  
s1.say\_name()  
  
s2 = Circle()  
s2.say\_name()

TypeError: Can't instantiate abstract class Shape with abstract methods say\_name

TypeError: Can't instantiate abstract class Circle with abstract methods say\_name

* There are 2 errors here.

1. The first error is because you cannot make an object from an **abstract class**.
2. The second error is because there is no sayName() method in the ***Circle*** class. If a method is abstract in the **super class**, then the **sub classes** MUST make that method also, and they MUST override it.

Here is the code fixed:

**from** abc **import** ABC, abstractmethod  
  
**class** Shape(ABC):  
  
 @abstractmethod  
 **def** say\_name(self):  
 **pass  
  
  
class** Circle(Shape):  
  
 **def** say\_name(self):  
 print(**"I am a circle"**)  
  
s1 = Circle()  
s1.say\_name()

I am a circle

Exercise 10-03

**from** abc **import** ABC, abstractmethod  
  
<Make an abstract class called Vehicle>  
  
 **def** \_\_init\_\_(self, name, number\_of\_wheels):  
 self.name = name  
 self.number\_of\_wheels = number\_of\_wheels  
  
 <Make an abstract method called display\_vehicle(). Use pass> **class** Car(Vehicle):  
  
 **def** \_\_init\_\_(self, name, number\_of\_wheels, number\_of\_doors):  
 Vehicle.\_\_init\_\_(self, name, number\_of\_wheels)  
 self.number\_of\_doors = number\_of\_doors  
  
 **def** display\_vehicle(self):  
 print(**"{0} has {1} wheels and {2} doors"**.format(self.name, self.number\_of\_wheels, self.number\_of\_doors))  
  
  
<Make a class called Bike that inherits vehicle>  
   
 <Make the \_\_init\_\_ method that calls the Vehicle \_\_init\_\_ method>  
 <It also has the parameter **type**>  
  
 <Override the display\_vehicle() method. Make it say "??? has ?? wheels and is a ??? bike>  
   
  
  
<Make an object called v1 from the Car class with name="Car", number of wheels = 4 and number of doors = 4>

<Make an object called v2 from the Bike class with name="Bike", number of wheels = 2 and type = "mountain">

<Call the display\_vehicle() method on v1>  
<Call the display\_vehicle() method on v2>

Car has 4 wheels and 4 doors

Bike has 2 wheels and is a mountain bike

Exercise 10-04

* Make an abstract super class called Electronic
* It has an object variable called company
* It has an abstract method called turn\_on
* Make a class called Smartphone that inherits Electronic
* It has an object variable called GB
* The turn\_on method prints "Hold button on side”
* Make a class called TV that inherits Electronic
* It has an object variable called size
* The turn\_on method prints "Push button on remote”
* Use this code to make the objects:

e1 = Smartphone(**"Meizu"**, 64)  
e2 = TV(**"Samsung"**, 50)  
e1.turn\_on()  
e2.turn\_on()

Hold button on side

Push button on remote

*Electronic*

+ company

+ turn\_on()

TV

+ company  
+ size

+ turn\_on()

Abstract super class

Sub classes

Smartphone

+ company  
+ GB

+ turn\_on()

# Section 11: Interfaces

### Interfaces

**Interfaces** define what a class should look like.

An interface’s **data members** are only a set of signatures and they are not implemented. This means **Interfaces** never have a super class. They do not inherit other classes.

**Access modifiers** (public, private and protected) are not used for interface members because all interface members are **public** and **abstract**.

Interfaces cannot have **constructors** or \_\_init\_\_ methods, and cannot be instantiated.

A class that **implements** an interface must **define** the interface’s methods with signatures the same as the interface, and they must **implement** the methods.

Python does not have **Interfaces** because Python can use **Multiple Inheritance**. However, we can show how Interfaces work using **Abstract** classes.

**Interfaces** have these rules:

* You cannot make an object from an **Interface**.
* Must have **ONLY** **abstract methods** that **MUST** be made (**overriden**) in the classes that use the **Interface**.
* **Abstract methods** in the **Interface** have no code in them.

What is the difference between an Interface and an Abstract Class?

* An **abstract class** can have **abstract methods** and **normal methods**.
* An **interface** can only have **abstract methods** that must be **overridden** in the sub classes.

**from** abc **import** ABC, abstractmethod  
  
*# Interface***class** I\_Shape(ABC):  
  
 @abstractmethod  
 **def** area(self):  
 **pass** @abstractmethod  
 **def** perimeter(self):  
 **pass**s = I\_Shape()

TypeError: Can't instantiate abstract class I\_Shape with abstract methods area, perimeter

We have an error here because we tried to make an object from I**nterface** class **I\_Shape**. You cannot make an object from an **Abstract** class.

**from** abc **import** ABC, abstractmethod  
  
*# Interface***class** I\_Shape(ABC):  
  
 @abstractmethod  
 **def** area(self):  
 **pass** @abstractmethod  
 **def** perimeter(self):  
 **pass  
  
  
class** Rectangle(I\_Shape):  
  
 **def** \_\_init\_\_(self, width, height):  
 self.width = width  
 self.height = height  
  
rec = Rectangle()

TypeError: Can't instantiate abstract class Rectangle with abstract methods area, perimeter

We have an error here because we have not made the methods area() and perimeter() that **override** the methods in the **abstract** class **I\_Shape**.

**from** abc **import** ABC, abstractmethod  
  
*# Interface***class** I\_Shape(ABC):  
  
  *@abstractmethod* **def** area(self):  
 **pass**  *@abstractmethod* **def** perimeter(self):  
 **pass  
  
  
class** Rectangle(I\_Shape):  
  
 **def** \_\_init\_\_(self, width, height):  
 self.width = width  
 self.height = height  
  
 **def** area(self):  
 **return** self.width \* self.height  
  
 **def** perimeter(self):  
 **return** 2 \* self.width + 2 \* self.height  
  
  
rec = Rectangle(5, 6)  
print(rec.area())  
print(rec.perimeter())

30

22

Now our program runs correctly with no errors because we have used **overriding** methods for area() and perimeter() in the ***Rectangle*** class.

We can actually make another class that inherits the **Rectangle** class:

**from** abc **import** ABC, abstractmethod  
  
*# Interface***class** I\_Shape(ABC):  
  
  *@abstractmethod* **def** area(self):  
 **pass**  *@abstractmethod* **def** perimeter(self):  
 **pass  
  
  
class** Rectangle(I\_Shape):  
  
 **def** \_\_init\_\_(self, width, height):  
 self.width = width  
 self.height = height  
  
 **def** area(self):  
 **return** self.width \* self.height  
  
 **def** perimeter(self):  
 **return** 2 \* self.width + 2 \* self.height  
  
  
**class** Square(Rectangle):  
  
 **def** \_\_init\_\_(self, side):  
 Rectangle.\_\_init\_\_(self, side, side)  
 self.side = side  
  
  
rec = Rectangle(5, 6)  
print(rec.area())  
print(rec.perimeter())  
  
s = Square(7)  
print(s.area())  
print(s.perimeter())

30

22

49

28

The **Square** class also inherits the **I\_Shape** **interface** class, and so it can use the area() and perimeter() methods.

The **\_\_init\_\_** method for **Square** needs only one parameter called side because the width and the height are the same for a square. It then uses the **\_\_init\_\_** method from the **Rectangle** class, giving it the side variable as parameters for width and height. The Square object uses the area() and perimeter() methods from the **Rectangle** class.

If we use an **overriding** method area() in the **Square** class, we can see that this method will be used, not the area() method in the **Rectangle** class:

**from** abc **import** ABC, abstractmethod  
  
*# Interface***class** I\_Shape(ABC):  
  
  *@abstractmethod* **def** area(self):  
 **pass**  *@abstractmethod* **def** perimeter(self):  
 **pass  
  
  
class** Rectangle(I\_Shape):  
  
 **def** \_\_init\_\_(self, width, height):  
 self.width = width  
 self.height = height  
  
 **def** area(self):  
 **return** self.width \* self.height  
  
 **def** perimeter(self):  
 **return** 2 \* self.width + 2 \* self.height  
  
**class** Square(Rectangle):  
  
 **def** \_\_init\_\_(self, side):  
 Rectangle.\_\_init\_\_(self, side, side)  
 self.side = side  
  
 **def** area(self):  
 print(**"Using Square area method"**)  
 **return** self.side \* self.side  
  
  
s = Square(7)  
print(s.area())  
print(s.perimeter())

Using Square area method

49

28

Example

36.5

Whale swimming

60

36.5

Lion running

80

**from** abc **import** ABC, abstractmethod  
  
**class** Mammal:  
  
 **def** \_\_init\_\_(self, body\_temp):  
 self.body\_temp = body\_temp  
  
  
**class** I\_land\_mammal(ABC): *# INTERFACE*  
  
 @abstractmethod  
 **def** run(self):  
 **pass**

**class** I\_sea\_mammal(ABC): *# INTERFACE*  
  
 @abstractmethod  
 **def** swim(self):  
 **pass**

**class** Whale(Mammal, I\_sea\_mammal):  
  
 **def** \_\_init\_\_(self, body\_temp, swim\_speed):  
 Mammal.\_\_init\_\_(self, body\_temp)  
 self.swim\_speed = swim\_speed  
  
 **def** swim(self):  
 **return "Whale swimming"**  
**class** Lion(Mammal, I\_land\_mammal):  
  
 **def** \_\_init\_\_(self, body\_temp, run\_speed):  
 Mammal.\_\_init\_\_(self, body\_temp)  
 self.run\_speed = run\_speed  
  
 **def** run(self):  
 **return "Lion running"**  
  
m1 = Whale(36.5, 60)  
print(m1.body\_temp)  
print(m1.swim())  
print(m1.swim\_speed)  
  
m2 = Lion(36.5, 80)  
print(m2.body\_temp)  
print(m2.run())  
print(m2.run\_speed)

Exercise 11-01

**from** abc **import** ABC, abstractmethod  
<Make a class called Vehicle>  
  
 <Make the \_\_init\_\_ method with a parameter colour. Set self.colour>

*# Interface*<Make an interface class called I\_air\_vehicle>  
  
 <Make an abstract method called fly(). Use pass>

*# Interface*<Make an interface class called I\_land\_vehicle>  
  
 <Make an abstract method called drive(). Use pass>

**class** Plane(Vehicle, I\_air\_vehicle):  
  
 **def** \_\_init\_\_(self, colour, air\_speed):  
 Vehicle.\_\_init\_\_(self, colour)  
 self.air\_speed = air\_speed  
  
 **def** fly(self):  
 **return "Plane flying"**

<Make class car that inherits Vehicle and I\_land\_vehicle>  
  
 <Make the \_\_init\_\_ method with the land\_speed parameter>

<Call the \_\_init\_\_ method from the Vehicle class class>

<Set land\_speed>

<Make the drive() method which returns "Car driving">

<Make an object from Plane called v1 with colour="red" and air speed=200>  
<Print the colour>  
<Print the fly() method>  
<Print the air speed>  
  
<Make an object from Car called v1 with colour="green" and land speed=100>  
<Print the colour>  
<Print the drive() method>  
<Print the land speed>

Red

Plane flying

200

Green

Car driving

100

# Section 12: Lists of Objects

### Parameter Passing

In Python, there are **Value Types** and **Reference Types**.

**Value Types** are actual values like 23 or 5.14

**Reference Types** are references to memory locations – they are not values.

Python uses **Value Types** on data types that are **immutable**. **Immutable** means NOT able to be changed. **Immutable** types are:

* int
* float
* string

Python uses **Reference Types** on data types that are **mutable**. **Mutable** means able to be changed. **Mutable** types are:

* list
* dictionary

Value Types

**def** change\_number(x):  
 x = 42  
 print(x)  
  
x = 5  
print(x)  
change\_number(x)  
print(x)

5

42

5

* The variable x is set to **5** and printed.
* The variable x is then passes to the change\_number() function as a parameter. Now a new variable x is made and it has a different location in memory. This is because the int x is **immutable**.
* The new variable x is set to **42** and printed.
* After we exit the change\_number() function, we print the original x and it is still **5**.

**5**

**x**

**5**

**x**

**42**

**new  
x**

**5**

**x**

Reference Types

**def** change\_list(x):  
 x[0] = 999  
 print(x)  
  
x = [1,2,3,4,5]  
print(x)  
change\_list(x)  
print(x)

[1, 2, 3, 4, 5]

[999, 2, 3, 4, 5]

[999, 2, 3, 4, 5]

* The variable x is set to a list of **1,2,3,4,5** and printed.
* The variable x is then passes to the change\_list() function as a parameter. Now the list x inside the function is the same as the list x outside the function. This is because the list x is **mutable**.
* The same list x is changed so that the first element is set to **999**, and printed.
* After we exit the change\_list() function, we print x and we can see it has been changed.

**1,2,3,4,5**

**x**

**999,2,3,4,5**

**x**

**999,2,3,4,5**

**x**

### Lists

Remember that we can make an empty **List**, then **append** values to it, and print out each element using a **For Loop**:

my\_marks = []  
my\_marks.append(54)  
my\_marks.append(87)  
my\_marks.append(67)  
my\_marks.append(63)  
  
**for** mark **in** my\_marks:  
 print(mark)

54

87

67

63

Exercise 12-01

|  |
| --- |
| **Mark** |
| 75 |
| 87 |
| 74 |
| 63 |
| 87 |
| 71 |

<Make an empty list called student\_marks>  
  
<Put the marks from the table above into the list>  
<Change the third mark to 55>  
  
<Use a for loop to print out each mark>  
  
<Make a variable called sum\_marks>  
  
<Use a for loop to find the sum of the marks>  
  
<Print the sum of the marks>  
<Print the average mark>

75

87

55

63

87

71

Sum: 438

Average: 73.0

Extra Exercise

Find the largest mark, the smallest mark and how many marks are between 70 and 80.

Largest mark: 87

Smallest mark: 55

There are 2 marks between 70 and 80

Polymorphic Lists using Inheritance

We can do something very similar to **Lists** that have **objects** inside them. First we will make a **super** **class** and some **sub** **classes**:

**class** Animal:  
  
 **def** \_\_init\_\_(self, name):  
 self.name = name  
  
 **def** talk(self):  
 print(**"Nothing"**)  
  
  
**class** Dog(Animal):  
  
 **def** \_\_init\_\_(self, name, breed):  
 Animal.\_\_init\_\_(self, name)  
 self.breed = breed  
  
 **def** talk(self):  
 print(**"Woof"**)  
  
  
**class** Duck(Animal):  
  
 **def** \_\_init\_\_(self, name, colour):  
 Animal.\_\_init\_\_(self, name)  
 self.colour = colour  
  
 **def** talk(self):  
 print(**"Quack"**)

Then we can make a **list** of animals:

animal\_list = [Animal(**"Animal"**), Dog(**"Doge"**, **"Labrador"**),  
 Duck(**"Donald"**, **"Green"**), Snake(**"Python"**, 5)]

Now we can use a for loop to print the name and call the polymorphic method:

**for** animal **in** animal\_list:  
 print(animal.name)  
 animal.talk()

Animal

Nothing

Doge

Woof

Donald

Quack

Python

Sssssssss

Exercise 12-02

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Class** | **Name** | **Age** | **Student ID** | **Subject** |
| Person | Alice | 23 |  |  |
| Student | Carl | 19 | "2017A121" |  |
| Teacher | Tom | 32 |  | "IT" |

<Make a class called Person>  
  
 <Make the init method with object variables name and age>  
  
  
<Make a class called Student that inherits Person>  
  
 <Make the init method with object variables name,age,student\_id>  
 <Call the Person init method>  
 <Set the student\_id object variable>  
  
  
<Make a class called Teacher that inherits Person>  
  
 <Make the init method with object variables name,age,subject>  
 <Call the Person init method>  
 <Set the subject object variable>  
  
  
<Make an empty list called people>  
<Put 3 objects into the list using the data in the table>   
  
<Make a variable called sum\_ages>  
  
<Make a for loop to find the sum of the age>  
  
<Print out the sum of the age>

Sum of ages: 74

Extra Exercise

Find the average age, the largest age, the smallest age and how many ages are between 20 and 30.

Largest age: 32

Smallest age: 19

Number of ages between 20 and 30: 1

Exercise 12-03 **(EXTRA)**

|  |  |
| --- | --- |
| **Name** | **Salary** |
| Tim | 65000 |
| Jane | 52000 |
| Sam | 48000 |

<Make a class called Worker>  
  
 <Make the init method with object variables name and salary>  
  
  
<Make an empty list called workers>  
<Put 3 worker objects into the list using the data in the table>   
  
<Make a variable called sum\_salary>  
  
<Make a for loop to find the sum of the salary>  
  
<Print out the sum of the salary>

Sum of the salaries: 165000

Extra Exercise

Find the average salary, the largest salary, the smallest salary and how many salaries are between 50000 and 70000.

Largest salary: 65000

Smallest salary: 48000

Number of salaries between 50000 and 70000: 2

Polymorphic Lists using an Abstract Class or Interface

Here is the same example with the **Animal** class as an **Abstract** **Class** or **Interface**:

There is a class called Animal that has an **abstract** method talk()

The Cat class and the Dog class both inherit the Animal class. They override the talk() method with their own method talk()

We can make a **List** of **Animals**. We know they may be different *types* of **animals**, but we also know they have a talk() method. When we call the talk() method it calls the correct talk() method for that object:

**from** abc **import** ABC, abstractmethod  
  
**class** Animal(ABC):  
  
 **def** \_\_init\_\_(self, name):  
 self.name = name  
  
 @abstractmethod  
 **def** talk(self):  
 **return "Nothing"  
  
  
class** Cat(Animal):  
  
 **def** talk(self):  
 **return "Meow"  
  
  
class** Dog(Animal):  
  
 **def** talk(self):  
 **return "Woof"**animals = [Cat(**"Carl the Cat"**),  
 Dog(**"Doge the Dog"**)]  
  
**for** animal **in** animals:  
 print(animal.name + **": "** + animal.talk())

Carl the Cat: Meow

Doge the Dog: Woof

Exercise 12-04

**from** abc **import** ABC, abstractmethod  
  
**class** Shape(ABC):  
  
 **def** \_\_init\_\_(self, colour):  
 self.colour = colour  
  
 **def** fill(self):  
 print(**"Shape filled with {0}"**.format(self.colour))  
  
 @abstractmethod  
 **def** get\_area(self):  
 **pass** <Make an abstract method draw()> **class** Rectangle(Shape):

**def** \_\_init\_\_(self, colour, length, width):  
 Shape.\_\_init\_\_(self, colour)  
 self.length = length  
 self.width = width  
  
 **def** get\_area(self):  
 **return** self.length \* self.width  
  
 **def** draw(self):  
 print(**"Drawing a rectangle"**)  
  
<Make a class Circle that inherits the Shape class>

<Make the \_\_init\_\_ method with the colour parameter and another parameter called radius>

<Call the \_\_init\_\_ method from the Shape class>

<Make the object variable radius>  
  
 <Make the get\_area() method that calculates the area of the circle>  
  
 <Make the draw() method that prints "Drawing a circle">  
  
shapes = [Circle(**"Blue"**, 10),  
 Circle(**"White"**, 20),  
 Rectangle(**"Black"**, 3, 5),  
 Rectangle(**"Red"**, 7, 9)]  
  
<Make a for loop to call the fill() method and the draw() method on each object in the shapes list>  
  
total\_area = 0  
max\_area = shapes[0].get\_area()  
<Make min\_area equal to the get\_area() method of the first element in the shapes list>

<Make a for loop to go through each shape object in the shapes list>

<Make area1 equal to the get\_area() method of the shape object>  
 <Make total\_area equal to total\_area + area1>  
  
 <Make an if statement to see if area1 is greater than max\_area. If it is, make max\_area equal to area1>

<Make an if statement to see if area1 is less than min\_area. If it is, make min\_area equal to area1>

<Calculate the average area and put it in average\_area>

<Print these variables to 2 decimal places:  
total\_area  
average\_area  
max\_area  
min\_area>

Shape filled with Blue

Drawing a circle

Shape filled with White

Drawing a circle

Shape filled with Black

Drawing a rectangle

Shape filled with Red

Drawing a rectangle

1648.80

412.20

1256.64

15.00

# Section 13: Recursion

**Recursion** is a way of programming or coding a problem, where a function **calls** **itself** one or more times. Usually, it is returning the return value of this function call.

We know that in Python, a function can call other functions. It is even possible for the function to call **itself**. These are called **recursive functions**.

**Recursion** is basically the process of a function calling itself. For example:

**def** my\_function(x):

print(x)  
 my\_function(x)  
  
my\_function(5)

5

5

5

5

...

This is a bad example, because this loop will go on forever. A good **recursive function** will have a condition that causes it to stop, just like a **loop**.

Here is a better function:

**def** my\_function(x):  
  
 print(x)  
 **if** x==0:  
 **return**

x-= 1  
 my\_function(x)  
  
my\_function(5)

5

4

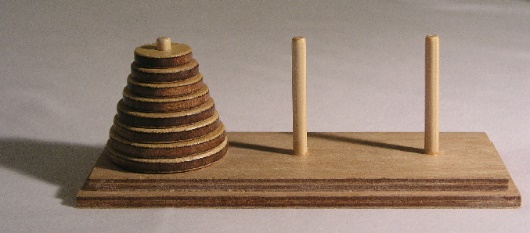
3

2

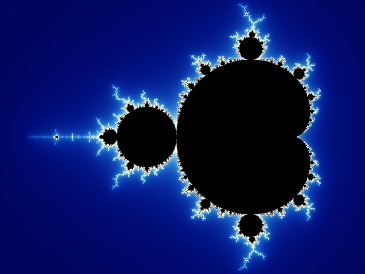
1

0

The function has the parameter x. If x is equal to zero, the function stops. Otherwise, the function is called again, but the value of x is decreased by 1.

Why do we use Recursion?

There are many problems that can use **recursion**:

* Factorial: 5! = 5 \* 4 \* 3 \* 2 \* 1
* Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ...
* The Tower of Hanoi
* Mandelbrot set

Recursive solutions are often simple.

Recursion is a common topic in job interviews.

Example

**def** countdown(n):  
  
 **if** n <= 0:  
 print(**"Go!"**)  
 **else**:  
 print(n)  
 countdown(n-1)  
  
countdown(5)

5

4

3

2

1

Go!

**Here is how this works:**

1. Function is passed n=5 countdown(5)
2. if statement is False, so print 5, calls itself with n=4 countdown(4)
3. if statement is False, so print 4, calls itself with n=3 countdown(3)
4. if statement is False, so print 3, calls itself with n=2 countdown(2)
5. if statement is False, so print 2, calls itself with n=1 countdown(1)
6. if statement is False, so print 1, calls itself with n=0 countdown(0)
7. if statement is **True**, so print "Go!“
8. return from countdown(0)
9. return from countdown(1)
10. return from countdown(2)
11. return from countdown(3)
12. return from countdown(4)
13. return from countdown(5)

Exercise 13.01

Write a **Reclusive** **Function** to count from 1 to 10. When the function has printed 10, make it print “Finished!”

Use the example for countdown() above to help you.

1

2

3

4

5

6

7

8

9

10

Finished!

Example

**def** get\_sum(my\_list):  
  
 **if** len(my\_list) == 0:  
 **return** 0 *# stop* **else**:  
 new\_list = my\_list[1:len(my\_list)]  
 **return** my\_list[0] + get\_sum(new\_list)  
  
  
numbers = [10, 20, 30]  
sum\_list = get\_sum(numbers)  
print(sum\_list)

60

**Here is how this works:**

**60**

**10 + 20 + 30 + 0**

**20 + 30 + 0**

**30 + 0**

**0**

It begins with the call to get\_sum()

get\_sum([10, 20, 30])

if = False

10 + get\_sum([20, 30])

if = False

20 + get\_sum([30])

if = False

30 + get\_sum([])

if = True

**0**

Parts of a Recursive Function

**Base Case:**

It stops the loop of the recursion

**Recursive Case:**

The function calls itself

**def** countdown(n):  
  
 **if** n <= 0:  
 print(**"Go!"**)  
 **else**:  
 print(n)  
 countdown(n-1)  
  
countdown(5)

What happens if the Base Case is missing?

**def** countdown(n):  
  
 **if** n <= 0:  
 print(**"Go!"**)  
 **else**:  
 print(n)  
 countdown(n-1)  
  
countdown(5)

5

4

3

2

1

0

-1

-2

-3

-4

...

There MUST be a **Base Case** or the program will run forever.

What happens if the Recursive Case does not call itself?

The **Recursive Case** MUST call the same function.

**def** countdown(n):  
  
 **if** n <= 0:  
 print(**"Go!"**)  
 **else**:  
 print(n)  
 countdown(n-1)  
  
countdown(5)

5

The Recursive Case must go towards the Base Case

**def** countdown(n):  
  
 **if** n <= 0:  
 print(**"Go!"**)  
 **else**:  
 print(n)  
 countdown(n-1)  
  
countdown(5)

countdown(n) *# stays on the same number forever*countdown(n+1) *# goes away from the base case and goes forever*

The variable n must go towards 0, or the function will call itself forever.

Exercise 13.02

Write a **Reclusive** **Function** to multiply **\*** all the values in a list together. For example, if the list is:

* 1.2, 2.0, 3.5, 5.3
* The answer is 1.2 \* 2.0 \* 3.5 \* 5.3 = **44.52**

Use the example for get\_sum() above to help you.

44.52

Example 1

Write a **recursive** function that calculates the sum of all digits in an integer. For example, if the number is 235, then the function should calculate 2 + 3 + 5 = 10

First, get the number from the user. Then give this number to the **recursive** function, which will return the answer. Print out the answer.

Help:

**num % 10** will give you the last digit of **num**

For example,

**25 % 10 = 5**

**523 % 10 = 3**

Help:

**int(num / 10)** will give you all of the numbers except for the last digit of **num**

For example,

**int(25 / 10) = 2**

**int(523 / 10) = 52**

So you need to use the recursive function to do **int(num / 10)** to remove the last digit and give it to the recursive function.

**def** sum\_digits(num):  
  
 **if** num <10:  
 **return** num  
 **else**:  
 number\_to\_add = num%10  
 cut\_number = (int)(num/10)  
 **return** number\_to\_add + sum\_digits(cut\_number)  
  
  
print(sum\_digits(234))

Example output:

Enter a number: 234

The sum of all of the digits of 234 is 9

Enter a number: 1895

The product of all of the digits of 1895 is 23

Example 2

Make a class called Animal. It should have an \_\_init\_\_ method with a parameter number\_of\_legs. Set the object variable number\_of\_legs.

Then write the code that uses a recursive function called add\_legs() to calculate the total number of legs in a list of animals.

**class** Animal:  
  
 **def** \_\_init\_\_(self, number\_of\_legs):  
 self.number\_of\_legs = number\_of\_legs  
  
  
**def** add\_legs(animals):  
 **if** len(animals) == 0:  
 **return** 0  
 **else**:  
 **return** animals[0].number\_of\_legs + add\_legs(animals[1:])  
  
  
animals = []  
animals.append(Animal(4))  
animals.append(Animal(2))  
animals.append(Animal(8))  
  
total\_legs = add\_legs(animals)  
print(**"Total number of legs: {0}"**.format(total\_legs))

Example output:

Total number of legs: 14

Recursion Rules

If you want to print numbers one by one, use this:

**def** recursive\_function(x):  
  
 **if** ??????:

return **or**  
 print "message"

**else:** recursive\_function(????)  
  
recursive\_function(???)

Example

5

4

3

2

1

OK

**def** recursive\_function(x):  
  
 **if** x == 0:  
 print(**"OK"**)  
 **else**:  
 print(x)  
 recursive\_function(x-1)  
  
recursive\_function(5)

If you want to print a total or the product of many numbers, use this:

**def** recursive\_function(x):  
  
 **if** x == 0:  
 return 0 if adding a total  
 return 1 if multiplying together  
 **else**:  
 **return** x + **or** \* recursive\_function(????)  
  
print(recursive\_function(???))

Example

**def** recursive\_function(x):  
  
 **if** x == 0:  
 **return** 0  
 **else**:  
 **return** x + recursive\_function(x-1)  
  
print(recursive\_function(5))

15

Basically, the first **return** means you need to **return** what you want the last thing to do.

Exercise 13.03

Make a class called Person. It should have an \_\_init\_\_ method with a parameter name and a parameter money. Set the object variables name and money.

Then write the code that uses a **recursive function** called add\_money() to calculate the total money in an list of people.

**class** Person:  
  
 <Make the \_\_init\_\_ method>  
  
<Make the add\_money() method>  
  
people = []  
people.append(Person(**"Sam"**, 23))  
people.append(Person(**"Mary"**, 142))  
people.append(Person(**"John"**, 74))  
  
total\_money = add\_money(people)  
print(**"Total money: {0}"**.format(total\_money))

Total money: 239

Exercise 13.04

In Mathematics you can calculate the factorial of a number:

5! = 5 \* 4 \* 3 \* 2 \* 1

x! = x \* (x-1) \* (x-2) \* (x-3) \* (x-4)

We can write a **for loop** to calculate this:

**def** factorial(x):  
 total = 1  
 **for** i **in** range(x, 0, -1):  
 total \*= i  
  
 **return** total  
  
print(factorial(5))

120

However, if you look closely, the factorial function can use **Recursion**:

factorial(x) = x \* (x-1) \* (x-2) \* ... \* 2 \* 1

but what is (x-1) \* (x-2) \* ... \* 2 \* 1 ?

It is equal to (x-1)!

So x! = x \* (x-1)!

We can write a Recursive Function now:

**def** factorial(x):  
  
 Stop if x is less than or equal to 0 *# this is not Python code*

return ???   
 Else

return x \* (x-1)!   
  
print(factorial(5))

120

Exercise 13.05

Make a Recursive Function to calculate the Fibonacci Sequence.

* The Fibonacci Sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21 …
* So for 21, this is equal to 8 + 13
* This means f(9) = f(7) + f(8)
* But f(7) = f(5) + f(6)
* So you can see we can use a **Recursive Function**

<Make a function called fibonacci with a parameter n>  
  
 <if n is less than or equal to 1, return n>  
  
 <else return the fibonacci function for n-1 + the fibonacci function for n-2>  
  
<Make n = 20>  
  
<Make a loop from 0 to n to print out the fibonacci function for each number>

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181,

# Section 14: File I/O

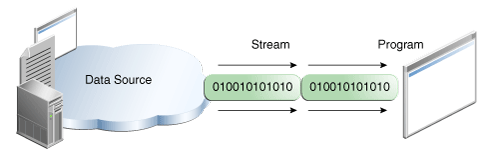
Python can read and write to text files. This is called **File I/O (File Input / Output)**. Python has basic functions and methods that can read and write to files. You can do most of the file changes using a **file object**.

### Streams

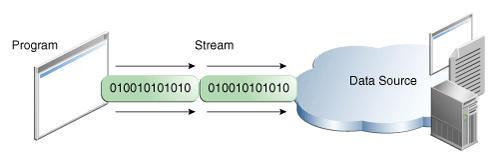
Python uses streams for I/O.

* A **stream** is a series of data (in bytes) that flows into a **buffer.**
* Data enters the stream one character at a time.

Load data into a program



Write data to somewhere



### Open

The first thing you’ll need to do is use Python’s built-in open() function to get a **file object**.

The open() function opens a file.

When you use the open() function, it returns a **file object**. **File objects** have methods and attributes that can be used to get information about the file you opened. They can also be used to change the file.

When we use the open() method, we usually use 2 arguments:

file\_object = open(**"<filename>"**, **"<mode>"**)

* **"<filename>"** is the name of the text file.
  + It can have an **absolute** path, for example “C:\Temp\my\_file.txt”
  + Or it can have a **relative** path if the file is in the same folder as your Python .py file, for example “my\_file.txt”
* **"<mode>"** is the way the file will be used:
  + **"r"** is the default, and it means **Read**
  + **"w"** means **Write**, if we want to write some text into the file
* The data in the text file is put into the variable file\_object

f = open(**"C:\Temp\my\_file.txt"**, **"r"**)  
print(f.read())

This is my file. It has some words in it.

This is a new line.

This is the third line. That is all.

This opens the file and puts the all of the data into variable f. We can print out the data using f.read()

If we want to read only the first line, we can use readline()

f = open(**"C:\Temp\my\_file.txt"**, **"r"**)  
print(f.readline())

This is my file. It has some words in it.

Putting text file data into Lists

The text file might have a list of numbers like this:

1

2

3

4

5

If we want to put each number into a list, we can do this:

f = open(**"C:\Temp\myNumbers.txt"**, **"r"**)  
my\_numbers = []  
  
**for** x **in** f.readlines():  
 my\_numbers.append(int(x))

print(my\_numbers)

[1, 2, 3, 4, 5]

We use a **for loop** to go through each line and **append** the number to our my\_numbers list.

Exercise 14-01

* Make a text file called ***my\_text.txt*** and put this inside and save it somewhere in your C: or D: drive:

The Time Machine by H.G. Wells

The Time Traveller (for so it will be convenient to speak of him) was expounding a recondite matter to us.

His grey eyes shone and twinkled, and his usually pale face was flushed and animated.

The fire burned brightly, and the soft radiance of the incandescent lights in the lilies of silver caught the

bubbles that flashed and passed in our glasses.

Our chairs, being his patents, embraced and caressed

us rather than submitted to be sat upon, and there was that luxurious after-dinner atmosphere when

thought runs gracefully free of the trammels of precision.

And he put it to us in this way-marking the points

with a lean forefinger-as we sat and lazily admired his earnestness over this new paradox

(as we thought it) and his fecundity.

Make a program to print out this text.

Exercise 14-02

* Make a text file called ***my\_text2.txt*** and put this inside and save it somewhere in your C: or D: drive:

77

54

88

65

87

98

Make a program to find the sum and print it out.

Sum: 469

Split

If you need to break a large string down into smaller strings, you can use the split() function. It looks for a **separator** to cut the string into pieces and put them into a **List**. Here is an example where the **separator** is a comma **,**

x = **"blue,red,green"**y = x.split(**","**)  
print(y)

['blue', 'red', 'green']

If you want to cut a sentence into words, the **separator** is a space: " "

sentence = **"Here are four words"**sentenceList = sentence.split(**" "**)  
  
print(sentenceList)  
  
**for** word **in** sentenceList:  
 print(word)

['Here', 'are', 'four', 'words']

Here

are

four

words

Reading objects

We might have a text file with data about people. For example:

Brian,30

Julia,22

Tom,35

Joe,40

We can read these lines and make objects from them using a class **Person:**

**class** Person:  
  
 **def** \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age

The text file has the name and the age of a person, so we need to:

* Open the file
* Read each line
* Split the line using a comma **,**
* Put the first word into a variable for the name
* Put the second word into a variable for the age
* Make a new **Person** objectwith the name and age variables
* Put this object into a list of objects
* Print out the name and age for each object

f = open(**"C:\Temp\my\_objects.txt"**, **"r"**)  
my\_objects = []  
  
**for** line **in** f.readlines():  
 split\_line = line.split(**","**)  
 object\_name = split\_line[0]  
 object\_age = split\_line[1]  
 new\_object = Person(object\_name, int(object\_age))  
 my\_objects.append(new\_object)  
  
**for** obj **in** my\_objects:  
 print(obj.name + **": "** + str(obj.age))

Brian: 30

Julia: 22

Tom: 35

Joe: 40

Exercise 14-03

* Make a text file called ***my\_pets.txt*** and put this inside and save it somewhere in your C: or D: drive:

Dog,Max,4

Cat,Alice,4

Fish,Nemo,0

Spider,Fred,8

Make this program:

<Make a class called Pet>  
  
 <Make the \_\_init\_\_ method with the parameters self, type, name, and number\_of\_legs>  
 <Make the object variables for the parameters>  
  
  
<Open the my\_pets.txt file>  
<Make an empty List called my\_pets>  
  
<Make a for loop for each line in the file>  
 <Make a variable called split\_line and make it equal to the line split using a comma ,>  
 <Make pet\_type equal to the first element in the split\_line list>  
 <Make pet\_name equal to the second element in the split\_line list>  
 <Make pet\_legs equal to the third element in the split\_line list>  
  
<Make a variable called new\_pet and make it equal to a new Pet object, giving it the correct parameters>  
<Append the new\_pet object to the my\_pets List>  
  
<For each object in the my\_pets List, print the type, name and number of legs of the pet>

Dog: Max has 4 legs

Cat: Alice has 4 legs

Fish: Nemo has 0 legs

Spider: Fred has 8 legs

Writing to text files

If we want to write to a file, we have to use a different **mode**.

f = open(**"C:\Temp\my\_file.txt"**, **"w"**)  
print(f.write(**"..."**))

The write **mode** (**w**) will write completely new contents to a file, deleting the old text. After writing to the file, we need to close it. This is like saving the file. Then to read it, we need to open it again:

f1 = open(**"C:\Temp\my\_file.txt"**, **"w"**)  
f1.write(**"New words\nNew line\nLast line"**)  
f1.close()  
  
f2 = open(**"C:\Temp\my\_file.txt"**, **"r"**)  
print(f2.read())

New words

New line

Last line

### Debugging and Error Handling

There are 3 types of errors in Python:

1. **Syntax errors**: caught by the Python compiler before the program runs.
2. **Runtime** **errors**: caught by the Python interpreter after the program has run.
3. **Logical** **errors**: caught by you.

Exceptions

An **exception** is an error that happens during the running of a program. When that error happens, Python makes an **exception** that can be handled, which stops your program crashing.

Exceptions are just classes that have information about the problem inside them. Exception Handling is just the task of **catching** those errors, fixing them or letting the program finish by telling the user what went wrong.

Example

If we try to open a text file that is missing, we get an error:

f1 = open(**"C:\Temp\missing\_file.txt"**, **"r"**)

FileNotFoundError: [Errno 2] No such file or directory: 'C:\\Temp\\missing\_file.txt'

To catch an error, we can write this code:

**try**:  
 <code>  
  
**except** <type\_of\_error> **as** <a\_variable\_for\_the\_error\_message>:  
 print(**"something"**)

To fix our missing file error, we can use the error type called IOError

**try**:  
 f1 = open(**"C:\Temp\missing\_file.txt"**, **"r"**)

**except** IOError **as** my\_err:  
 print(**"The file cannot be found"**)  
 print(my\_err)

The file cannot be found

[Errno 2] No such file or directory: 'C:\\Temp\\missing\_file.txt'

There are many error types. Here we are dividing by 0 and we get an error:

x = 32  
y = 0  
  
z = x/y

ZeroDivisionError: division by zero

We can use the ZeroDivisionError error type to handle this exception:

x = 32  
y = 0  
  
**try**:  
 z = x/y  
**except** ZeroDivisionError **as** my\_err:  
 print(**"Sorry, you cannot divide by zero"**)

Sorry, you cannot divide by zero

In Python, you do not always need to have **Exception Handling**. You can use them if:

* You know an error may happen, but you cannot stop it. For example, a database server is down, or the internet disconnects.
* You want to do something special if an error happens, like:
  + Make a log (record) of the error
  + Reverse an action, for example, a bank transaction
  + Tell the user what error happened.

Exercise 14-04

Make a text file called ***my\_words.txt*** in your C: or D: drive.

Use the write() function to put this text into the text file:

The thing the Time Traveller held in his hand was a glittering metallic

framework,

scarcely larger than a small clock,

and very delicately made.

Use the read() function to print out the words:

The thing the Time Traveller held in his hand was a glittering metallic

framework,

scarcely larger than a small clock,

and very delicately made.

Exercise 14-05

Make a text file called ***my\_numbers.txt*** in your C: or D: drive and put this text inside

21 0 4 64 34

<Make an error handler around this code using error type ValueError>  
 x = int(input(**"Please enter a number: "**))  
  
 <Make the error say "That was not valid number. Try again...">  
 exit(0) *# stop program*<Make an error handler around this code using error type IOError>  
 <Make f equal to open the file my\_numbers.txt>  
  
 <Make the error say "Cannot find file">  
 exit(0) *# stop program*

<Read the line in the text file and put it in a variable called line>  
<Split the text into an list called line\_list>  
<Make y equal to the second element of the line\_list. You must cast it to an int>  
  
<Make an error handler around this code using error type ZeroDivisionError>  
 z = x/y  
  
 <Make the error say "You cannot divide by zero">   
 exit(0) *# stop program*

If user types ***c***

Please enter a number: c

That was not a number. Try again...

If the ***my\_numbers.txt*** file cannot be found

Please enter a number: 4

Cannot find file

When the program divides by 0

Please enter a number: 7

You cannot divide by zero

Extra 1

**class** IntNumberClass:  
  
 **def** \_\_init\_\_(self, n):  
 self.n = n  
  
 **def** get\_cube(self):  
 **return** self.n \* self.n \* self.n  
  
**class** Wage:  
  
 **def** \_\_init\_\_(self, wh, hr):  
 self.wh = wh  
 self.hr = hr  
  
 **def** calc\_wage(self):  
 **return** self.wh \* self.hr  
  
  
**def** calculate\_sum\_integers(int\_list):  
  
 **if** len(int\_list) == 0:  
 **return** 0  
 **else**:  
 first\_int = int\_list[0]  
 new\_list = int\_list[1:]  
 total = first\_int + calculate\_sum\_integers(new\_list)  
 **return** total  
  
**def** calculate\_sum\_objects(obj\_list):  
  
 <Make a recursive function that calculates the sum of the n variables in the list of objects>

**def** get\_total\_cube(obj\_list):  
  
 **if** len(obj\_list) == 0:  
 **return** 0  
 **else**:  
 first\_int = obj\_list[0]  
 new\_list = obj\_list[1:]  
 total = first\_int.get\_cube() + get\_total\_cube(new\_list)  
 **return** total  
  
**def** get\_total\_wage(obj\_list):  
  
 <Make a recursive function that calculates the sum of the calc\_wage() method for each object in the list of objects>  
  
nums = [2,3,4]  
total = calculate\_sum\_integers(nums)  
print(total)  
  
num\_objects = [IntNumberClass(2), IntNumberClass(3), IntNumberClass(4)]  
total\_obj = calculate\_sum\_objects(num\_objects)  
print(total\_obj)  
  
total\_cube = get\_total\_cube(num\_objects)  
print(total\_cube)  
  
wage\_list = [Wage(15.5, 18), Wage(23.5, 19.5), Wage(35, 25)]  
total\_wages = get\_total\_wage(wage\_list)  
print(total\_wages)

9

9

99

1612.25

Extra 2

**class** Course:  
  
 **def** \_\_init\_\_(self, title, year):  
 self.title = title  
 self.year = year  
  
 **def** show\_course(self):  
 print(**"The course is {0} {1}"**.format(self.title, str(self.year)))  
  
  
**class** Course\_app:  
  
 course\_list = []  
  
 **def** \_\_init\_\_(self, fn):  
 self.get\_course\_data(fn)  
  
 **def** get\_course\_data(self, file\_name):  
  
 <Open the file>  
  
 <for each line>  
 <Split the line into line\_list>  
 <Put the first element into t>  
 <Put the second element into y. Use a cast to int>  
 <Make c equal to a new Course object using t and y>  
 <Append c to course\_list>  
  
 **def** show\_course\_lis t(self):  
  
 for each Course object in course\_list,  
 <call the show\_course() method>  
textFile = **"C:\Temp\my\_courses.txt"**app = Course\_app(textFile)  
app.show\_course\_list()

Output: ***my\_courses.txt:***

The course is English 2015

The course is Maths 2014

The course is Science 2016

The course is Economics 2017

English,2015

Maths,2014

Science,2016

Economics,2017