

Book of Python

by Paul Baumgarten - pbaumgarten.com

2020 edition (draft of 30/01/2020)

The lessons in this book have been written using Python 3.7 and have been tested with Microsoft Windows 10 and Mac OSX 10.14 environments.

"Python" and the Python logo are trademarks of the Python Software Foundation and are used in accordance with their usage policy. <https://www.python.org/psf/trademarks/>

Copyright © 2018,2020 by Paul Baumgarten

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the publisher except for the use of brief quotations in a book review or scholarly journal.

First edition 2018, second edition 2020.

Published by Paul Baumgarten
pbaumgarten.com

Table of contents

1.	Getting started	5
2.	Numbers	7
3.	Strings	15
4.	Making decisions	23
5.	Repetition	29
6.	Lists	33
7.	Computational thinking	41
8.	Functions	55
9.	Files & folders	61
10.	Exceptions	67
11.	Dates & times	71
12.	Dictionaries	77
13.	Classes & objects	85
14.	Classes & objects (part 2)	89
15.	Networking	95
16.	GUIs	103
17.	What's next	119

1. Getting started

Welcome to my Book of Python!

This book is a passion project I have been slowly progressing over a number of years. It started as a compilation of teaching notes and exercises that was slowly fleshed out with detailed explanations. It has been written for an audience of teenage students, and has been drawn from 14 years of teaching computing to Secondary aged students. I have written it to as to address many of the issues that I have seen students struggle with as they embark on this journey.

As you progress through the book, don't fall into the error of just reading and nodding along. Any sample code written, you should attempt to write and check it works as expected. Along the way you will develop the skill of spotting and correcting syntax errors which is just as important as understanding the logic of what the program is doing at this early stage. Once you've read a particular lesson, and successfully gotten the code shown within working, it's time to try the problem sets at the end of every lesson. Don't skip the problem sets! They are important for developing your understanding and problem solving skills.

My number one suggestion for learning programming is to have a spirit of grit. As Michigan State University describes it: "To have grit means you have courage and show the strength of your character." and "A person with true grit has passion and perseverance." Passion is necessary for motivation, and perseverance is necessary so you don't quit when it gets hard. It will be hard at times. Going into it knowing that, and choosing to program anyway, and you are half way there!

Finally, I encourage you to check my website at pbaumgarten.com. I will post any updates/corrections to the published text there as well as solutions to the problem sets, bonus problems for you to attempt, and additional tutorials on numerous other topics that while fun and interesting would have made this text so large it would be overwhelming to include them all. I also have a YouTube channel you may be interested in where I post of programming tutorials for my students at youtube.com/pbaumgarten.

Good luck and best wishes!

Paul Baumgarten 2020

Installing Python

Visit <https://python.org/downloads>

- If you are using Windows, please ensure you select the x86-64 version.
- If you are using Mac, all your downloads will be 64 bit anyway.
- If you are using Linux, well, I assume that means you probably know what you are doing. That said, you'll probably be wanting to do something like `sudo apt-get install python3 python3-pip`.

Installing an editor

Like most programmers, my editor of choice changes often. At the time of writing I am enjoying using Microsoft VS Code. It doesn't really matter which editor you use. Find something you like. JetBrains PyCharm (Community edition is free) is another good choice.

Microsoft Visual Studio Code can be downloaded from <https://code.visualstudio.com/download>



For a video walk through to get Visual Studio Code working with Python, please watch <https://www.youtube.com/watch?v=-R6HFLp7tTs> or scan the QR code.

Your first program

If you've followed the video walk through, you will have already done this.

Your first Python program, just to test that everything is setup and working correctly, is as simple as one line of code!

```
print("Hello world!")
```

Save your file with the file extension `.py`, such as `first.py` and then right click, and select `Run Python file in Terminal`. If everything works successfully, Python should print the words "Hello world!" on your screen.

References

1. https://www.canr.msu.edu/news/what_does_it_mean_to_have_grit

2. Numbers

Let's drive right in and look at a line of Python programming code...

```
a = 15
```

What is happening here? In this line of code we are asking Python to create a variable called `a` and to assign it the value of `15`.

You are hopefully familiar with the idea of a variable from your mathematics classes where it is used in fields like algebra, geometry and trigonometry. For instance, using the formula `area = base * height / 2` you can determine the area of a right angled triangle. If the base of a triangle is `10` and the height is `8`, this formula will allow us to calculate the area to be `40` (ignoring units for now).

Being comfortable with the idea of variables is crucial for success in computer programming. Unlike math where the content of our variables are generally limited to just numbers, in programming we might have a range of different types of information stored in a variable. We might have text stored in a variable, such as `name = "Grace Hopper"`. We can even store more complex information in variables like dates, or images, music, and so forth. These more complex types of data are called objects and will be the subject of some lessons later on.

Regardless of what type of information we are storing inside a variable, the basic principle remains the same: **A variable is a tool we use to allocate names to different sections of the computers memory, that we can use to store and retrieve information.**

Returning to the idea of numeric variables, let's spend a little while looking at how they work, and the different sub-types available to us.

There are two main types of numeric variables: the integer and the floating point number. Integers, you may recall from mathematics, are whole numbers without a decimal or fraction component. `0`, `1`, `1312`, `-53` and `42` are all examples of integers. Floating point numbers is a technical term used in the computing industry used for numbers that contain decimals.

Run the following Python code...

```
a = 14
print( a )
print( type( a ) )

b = 14.0
print( b )
print( type( b ) )
```

If you type the previous into a Python file and execute it, you should get output like this...

(**remember**: please don't just read and nod along, you should check the code works as expected. Along the way you will develop the skill of spotting and correcting syntax errors which is just as important as understanding the logic of what the program is doing at this early stage)

```
14
<class 'int'>
14.0
<class 'float'>
```

We can see that Python has determined the value stored in variable `a` to be an integer, and the value stored in variable `b` to be a floating point number. (Note: For brevity we will typically just say *float* instead of *floating point number* from here on to refer to these types of numbers)

What happens if we add an integer and float? Try adding the following two lines...

```
print( a + b )
print( type( a + b ) )
```

And you should get the output of

```
28.0
<class 'float'>
```

We get an answer that is a float. This is because all integers have a decimal equivalent, but decimal numbers do not have an integer equivalent. We can tell Python to convert a float to an integer, but we would need to specify what we want done with any decimal portion of the number, such as to round to the nearest integer, or to truncate off the decimals.

When working with numeric values, we have all the common arithmetic operations you would be familiar with such as addition, subtraction, multiplication, division, and exponents. Examples of each in use are...

```
print( 2 + 3 )           # addition
print( 1.5 + 2.25 )     # addition
print( 7 - 2 )          # subtraction
print( 3 * 4 )          # multiplication
print( 10 / 2 )         # division
print( 4 ** 3 )         # exponent
```

So far so good? Returning to the idea of variables, how do we use these arithmetic operations with variables? We've already created a couple of simple numeric variables, `a` and `b`. Because these are simply named locations in the computers memory where we have a number stored, we can use the variable name wherever we want that value to be used. Python will automatically look up the value and substitute it into the calculation. The addition written above could just as easily have been done this way...


```
a = 2
b = 3
print( a + b )
```

While that's nice it's not very useful by itself. Most of the time rather than just printing the result of a calculation, we will want Python to store the result into another variable so our program can use it elsewhere. To assign the result of the above calculation into a variable might look like this...

```
a = 2
b = 3
result = a + b
```

This brings me to a critical point that I have seen countless beginner programmers struggle with. It is vitally important to appreciate the structure of the above lines. The third line could be paraphrased as "assign the variable `result` to the calculation of `a + b`". Many beginner programmers mix this up and write a calculation, an equal sign, and then the name of the variable they want it stored in... but this is the wrong way around! **The variable receiving the answer must always be on the left of the assignment operator, and the calculation must always be on the right of the assignment operator.** Please do not mix these up!

Some more examples showing the use of numeric variables with different calculations...

```
a = 100
b = 6
c = a + b          # addition          ... c == 106
d = a - b          # subtraction       ... d == 94
e = a * b          # multiplication    ... e == 400
f = a / b          # division          ... f == 16.66667
g = a // b         # integer division   ... g == 16 (how many times does 6 go into 100)
h = a % b          # modulus remainder ... h == 4 (ie: remainder of 100 divided by 6)
i = a ** b         # exponent          ... i == 1000000000000 (ie: 10^6)
print(c,d,e,f,g,h,i)
```

Integer division and modulus

Note the introduction of *integer division* and the *modulus remainder* above. These might not be things you have used much in math class but they can come in incredibly handy for programming. A couple of illustrations will help demonstrate this.

Question: How many weeks and days are there in 100 days? Answer...

```
total_days = 100
weeks = total_days // 7
left_over_days = total_days % 7
print(weeks)
print(left_over_days)
```

Question: How many hours is 1000 minutes? Answer...

```
total_minutes = 1000
hours = total_minutes // 60
left_over_minutes = total_minutes % 60
```

As per the summary given above, the integer division tells us how many times the one number goes into the next. The number 23 has 4 5s for instance, and needs another 3 to reach 23. So `23 // 5` will return 4, and `23 % 5` will return 3.

One trick that modulus is very frequently used for is to calculate if a number is odd or even. `number % 2` will always equal 0 if the number is even and 1 if it is odd.

Rules for your variable names

You will find yourself creating a lot of variables in your programs, so it is important to establish a few ground rules that Python has before going any further:

- Variable names must start with an alpha character or underscore, but may then contain numeric characters. You can not have spaces in a variable name!

```
first number = 10      # This is not valid, spaces not allowed!
first_number = 10      # Much better!
```

- Variable names should be meaningful. Establish good habits early. It should be obvious from the name of the variable what it's purpose is. The names I've used in this section of `a`, `b`, `c` are **not** good variable names!

```
# One way of finding the area of a triangle with unclear variable names
b = 10
h = 15
a = 0.5 * b * h
# A better way to name your variables
base = 10
height = 15
area = 0.5 * base * height
```

- Python's preferred practice is to `separate_words_with_underscores`. You may have seen other programming languages using `camelCase`. Python won't complain if you do that, but stylistically the underscore method is what Python programmers prefer. If you choose to use `camelCase` that's ok, just be consistent.

```
one_two = 12           # The Python style
oneTwo = 12            # Will work, but is not considered Python's style
```

- Variable names are case sensitive. `Variable` is not the same as `variable`. This is why it is important to pick a style and stick to it, so you don't get yourself confused later on wondering, "how did I write that variable name?"

```
var = 10
print(var) # This will work
print(Var) # This will not work - Upper casing means this is treated as a different name
```

Converting between integer and float

Sometimes you will need to convert a float to an integer, or integer to float. Python has some built in functions (or commands) that allow us to do this.

To convert a float to integer, use the `int()` command to truncate, or `round()` to round.

```
a = 13.6
b = int(a)
c = round(a)
print(a,b,c)
```

To convert an integer to float, use the `float()` command.

```
a = 13
b = float(a)
print(a,b)
```

Generating random numbers

When working with numeric variables, it is a frequent desire to have our program generate a random number. Python provides a few different functions for us to do this. The first thing we need to do is to instruct Python to load the library of code containing the random functions into our program. We do this by putting the line `import random` at the top of our program.

The random number generator you will likely use most is `randint()` which requires two parameters. (A parameter is the name given to a value we provide to a command in the parentheses). The parameters specify the bottom and top of the range of numbers we want Python to choose from. For instance, to generate a random number between 1 and 100 inclusive...

```
import random

num = random.randint(1, 100)
print(num)
```

There are other random number functions but this is the one you will use most for now.

You can find the Python documentation on the others, should you need it, at <https://docs.python.org/3/library/random.html>.

Math library functions

Before wrapping up our introduction to numbers and variables with Python, there are a range of handy built in functions for common mathematical tasks that we'll take a brief look at. These are particularly useful for geometry and trigonometry. If you haven't learnt what **sin**, **cosine** and **tangent** are in mathematics yet, you should jump over this section.

The mathematical functions built in that you will have most common use for are...

```

answer = math.pi           #  $\pi = 3.141592$ 
answer = math.e             # the natural number,  $e = 2.718281$ 
answer = math.sqrt(100)    # Square root
answer = math.gcd(104,64)   # Greatest common divisor
answer = math.log(1024,2)   # Log of base 2
answer = math.hypot(6,8)    # Hypotenuse of triangle with sides 6, 8
answer = math.cos( angle )  # Cosine of angle (radians)
answer = math.sin( angle )  # Sine of angle (radians)
answer = math.tan( angle )  # Tangent of angle (radians)
answer = math.acos( adj/hypot ) # Arc-cosine in radians
answer = math.asin( opp/hypot ) # Arc-sine in radians
answer = math.atan( opp/adj ) # Arc-tan in radians
answer = math.degrees( rad ) # Convert radians to degrees
answer = math.radians( deg ) # Convert degrees to radians
answer = abs( val )         # Absolute value

```

In order for any of the above to work, however, we must instruct Python to import the **math** library into our program. We do this by adding the instruction **import math** to the top of our program.

A full example to calculate the hypotenuse of a triangle where the adjacent side is 20, and the angle is 45 degrees would look like...

```

import math

adjacent = 20.0
angle = 45.0          # 45 degrees
hypotenuse = adjacent / math.cos( math.radians( angle ) )
print(hypotenuse)     # prints 28.284...

```

- Note all the trigonometry functions work in radians instead of degrees. If you haven't learnt about radians in mathematics yet, simply copy the above method and use the **math.radians()** function to convert any degree angle into a radian angle.

Problems

The following questions assume you will use variables as the inputs into the problem, so the problems can recalculate solutions by changing the value assigned to the variable. You should also print the given information in your answer.

Hint: A lot of these problems will require you to practice using the integer division and modulus operations since these are most likely to be new ideas to you.

1. For any given number, extract the 10s digit. For example for 1234, print 3.
2. Area of a right angled triangle calculator. Given values for base and height, print the area.
3. For any two digit number, swap the position of the digits. For instance, 42 becomes 24.
4. For any three digit number, print the sum of the three digits. For instance 567 becomes 18 (5+6+8)
5. For any given year, print the century that year belongs to. Remember that 1999 and 2000 were the 20th century, whereas 2001 was the beginning of the 21st century.
6. Given a number representing the number of seconds since midnight, print the time in 24hour clock format. For example 70500 seconds should print 19 and 35.
7. The nation of Australia has the following denominations of dollar coins and notes: \$1, \$2, \$5, \$10, \$20, \$50, and \$100. Use Python arithmetic to calculate how many of each note is required to give change of (a) \$1724, (b) \$238, (c) \$43.

Problems that require the Math library functions

8. Area of a non-right angled triangle calculator. Given values for length a, length b and angle in degrees C, return the area of the triangle (remember you will have to convert degrees to radians first).
9. For any given values for a, b and c, will provide the solutions to the quadratic formula (you may assume both solutions are required). Be careful with your order of precedence. Here is an example solution set for testing: If $y=2x^2-4x-10$ then the solutions are 3.44949 and -1.44949.

3. Strings

A **string** is computer programmer speak for **text** (think of a string of text characters). We can store text data into variables just like numbers. A text string can be used to store names, addresses, email addresses, websites, whatever you need.

In Python, to create a string variable, we enclose the value in a set of quotation characters to denote the start and end of the text we want included. It doesn't matter if you use single or double quotes, provided the quote character you use to start the string is the same one you use to end the string.

A couple of examples of strings...

```
name = "Mr Baumgarten"      # Using double quotes
website = 'pbaumgarten.com' # Using single quotes
```

Python even lets you create a string that has multiple lines by using triple double quote characters like this...

```
poem = """Computer science is abbreviated as "comp sci."
Comp sci is so hard it gives you a sigh."""
```

(1)

Because Python lets you choose from three different ways to enclose your text, it means if you need one of the quotation characters to appear in your string, that's ok, you use different quotes to open and close your string. Notice how the poem above includes the double quote characters at the end of the first line.

Another way of inserting multiple lines is to use add a `\n` into your text. This is a special two character sequence that Python will recognise as code for *insert new line*. For example,

```
text = "I want this to appear on\ntwo lines."
print(text)
```

Will output...

I want this to appear on
two lines.

Input

Once we understand the idea of text strings as variables, we can now write programs that will ask the user to provide text as input. In Python, this is done through the `input()` command.

```
name = input("What is your name?")
print( "hello" )
print( name )
```

Which would generate...

```
What is your name?Mr Baumgarten
hello
Mr Baumgarten
```

Two things to notice here. There was no space after the question mark. Python will print what you supply to the input command and nothing more, so if you want a space to appear you should add it. Secondly, we probably want the name to appear on the same line as the greeting. Fortunately Python makes it incredibly easy join to strings together. We just can use the `+` operator...

```
name = input("What is your name? ")
print( "hello " + name )
```

Another way of doing this is to use an "f-string". This allows us to embed variable values into a template string. To create an "f-string", you place the letter `f` in front of the opening quote character, and then any variables enclosed within curly braces will be replaced with the variable content. F-strings were added to Python in version 3.6 and are very useful. They contain features that allow us to control the output better than just using the `+` operator. We'll get into the detail of some of those features later, but suffice to say the f-string will be used a lot in this text.

```
name = input("What is your name? ")
print( f"hello {name}" )
```

It is quite common that we may want to input numbers into our program as well. The `input()` command, however, will also provide a text string as it has no way of knowing that they user will necessarily type a number. If you, as the programmer, are aware the content will be a number though, you can convert a text string to an integer or float using the same `int()` and `float()` functions we looked at last chapter.

```
text = input("Please type a number: ")
number = int(text)
double = 2 * number
print( f"Double of {number} is {double}." )
```


We can simplify this if we don't need to keep the string `text` variable for reference later, by sending the output of `input()` straight into the `int()` function. The result might look like this...

```
number = int(input("Please type a number: "))
double = 2 * number
print( f"Double of {number} is {double}." )
```

Remember the exact same works for floats as well...

```
number = float(input("Please type a number: "))
double = 2 * number
print( f"Double of {number} is {double}." )
```

F-strings

If we are going to want to print numbers, it may be necessary to stylise them sometimes. Consider what happens with the following if the input was **1**.

```
number = float(input("Please type a number: "))
answer = number / 3
print( f"One third of {number} is {answer}." )
```

The output was

```
Please type a number: 1
One third of 1.0 is 0.3333333333333333.
```

While we got a highly accurate number with all those decimals, it might be more than we want to print. The great thing about f-strings is they allow us to control things like this.

To print a float with only two decimal places, for instance, would look like...

```
number = float(input("Please type a number: "))
answer = number / 3
print( f"One third of {number} is {answer:.2f}." )
```

Another situation is you may want to print several lines of numbers that are of different size, and to make it look nice you want them to line up. Consider the following...

```
base = 2
exponent = 20
answer = base ** exponent
print(f"Base:      {base}")
print(f"Exponent:  {exponent}")
print(f"Answer:     {answer}")
```

You went to all the effort to add spaces to make your values line up, only to get this print out...

```
Base:      2
Exponent: 20
Answer:   1048576
```

We all know that numbers look neater when they are right aligned. F-strings allows us to indicate the spacing (width) our numbers should take. To specify, for instance, that all the numbers should be 7 characters wide...

```
base = 2
exponent = 20
answer = base ** exponent
print(f"Base:      {base:7}")
print(f"Exponent:  {exponent:7}")
print(f"Answer:     {answer:7}")
```

Will now give an output of...

```
Base:      2
Exponent:  20
Answer:    1048576
```

One last thing I'll mention at this point is you can combine to two features together as well. Consider the following...

```
base = 2
exponent = 20
answer = base ** exponent
thirds = answer / 3
print(f"Base:      {base:7}")
print(f"Exponent:  {exponent:7}")
print(f"Answer:    {answer:7}")
print(f"Thirids:   {thirds:7}")
```

Which outputs...

```
Base:      2
Exponent:  20
Answer:    1048576
Thirids:   349525.3333333333
```

To set the overall width, while insisting on two decimal places, we could change it to look like this (remember to include the decimal point as requiring a character space)

```
base = 2
exponent = 20
answer = base ** exponent
thirds = answer / 3
print(f"Base:      {base:10.2f}")
print(f"Exponent:  {exponent:10.2f}")
print(f"Answer:    {answer:10.2f}")
print(f"Thirids:   {thirds:10.2f}")
```

Ten spaces for each number, with one decimal point character and 2 decimal places would look like...

```
Base:      2.00
Exponent:  20.00
Answer:    1048576.00
Thirids:   349525.33
```

There are still more f-string features that will discover along the way of this text as well.

String functionality

There are a number of other things we may wish to do with text strings other than just print them or join them together using f-strings.

One very common task is to extract just part of a string.

To get an individual character, you use a square bracket notation on the end of the variable name with the position number inside the brackets. Be aware that the first character in a string has position `0`, the second character has position `1`, etc.

```
name = "Douglas Adams"
first = name[0]
print( f"The first letter of '{name}' is '{first}'." )
```

We can also extract a range of characters, or a sub-string if you will, of the original text by using a colon to denote the first position and the end position. Be aware that starting position is inclusive and the ending position is exclusive (not included).

```
name = "Douglas Adams"
print( name[0:7] )      # Douglas
print( name[8:13] )     # Adams
```

Perhaps a clearer way to demonstrate the difference with the inclusive and exclusive use is as follows. By using a range of `[0:3]`, position `0` is included in the output, position `3` is not included in the output.

```
text = "0123456789"
print( text[0:3] )      # prints 012
```

If we want a range of everything from the beginning, or everything up to the end, we can leave off the beginning or ending number respectively as follows...

```
name = "Douglas Adams"
print( name[:7] )       # Douglas
print( name[8:] )       # Adams
```

When Python recognises our variable is a string, it provides a variety of other functions for us to use, accessibly by adding a dot after the variable name. The most common examples include...

```
original = "To infinity and beyond!"
altered = original.lower()      ## == "to infinity and beyond!"
altered = original.upper()      ## == "TO INFINITY AND BEYOND!"
altered = original.title()      ## == "To Infinity And Beyond!"
altered = original.swapcase()   ## == "tO INFINITY AND BEYOND!"
altered = original.ljust(30)    ## == "To infinity and beyond!      "
altered = original.rjust(30)    ## == "          To infinity and beyond!"
altered = original.replace(" ", "--") ## == "To--infinity--and--beyond!"
```

If we want to perform mathematical calculations based on the string content, we have some useful functions such as...

```
text = "So long and thanks for all the fish!"
num = len(text)          ## get length of string ... num == 36
num = text.count(" ")    ## count spaces in string ... num == 7
num = text.index("t")     ## position of first 'o' in the string ... num == 12
num = text.rindex("t")    ## position of last 'o' in the string ... num == 27
```

This is handy because we can use the result from one of these functions as the input to a substring extraction. If we have a variable with a persons full name, we can use the `.index()` function to locate where the space character is and use it to separate the name into two parts...

```
name = "Alan Turing"
space = name.index(" ")
given_name = name[:space]
family_name = name[space+1:]
```

Finally if we want to test the result of a string, to see if it matches a particular pattern, we can use the "is" functions shown below. These will be particular handy when we learn about `if` statements and they will be revisited later.

```
result = text.isnumeric()  ## does it contain only numbers?
result = text.isalpha()    ## does it contain only letters?
result = text.islower()    ## is it all lowercase?
result = text.isupper()    ## is it all uppercase?
result = text.istitle()     ## is it all title case?
result = text.isspace()    ## is it all spaces?
```

Problems

1. For any string that consists of exactly two words with one space separating them, swap the two words around. For example: Given the string `Hello world!`, have the program print `world! Hello`.
2. Given a sentence input, return how many words are in the sentence. For example, `The quick brown fox jumps over the lazy dog.` is 9 words.
3. Given a string input of a date in format, `dd/mm/yyyy`, print an output advising the current day, month and year number. For instance, `24/01/2020` should output `it is day 24 of month 1 in the year 2020`.
4. Given a string, return a new string made of 3 copies of the last 2 chars of the original string. Assume the input string length will be at least 2 characters. For example, the string "Hello" should be result in "lololo".
5. Given a string, return the string made of its first two chars, so the String "Hello" yields "He". If the string is shorter than length 2, return whatever there is, so "X" yields "X", and the empty string "" yields the empty string "".
6. Given a string, return a version without the first and last char, so "Hello" yields "ell". The string length will be at least 2.
7. Given 2 strings, return their concatenation, except omit the first char of each. The strings will be at least length 1. For example, strings "Hello" and "There" should result in "ellohere".
8. How would you print the following? `All "good" men should come to the aid of their country.` (ie: how to print the double quote character)
9. Write code that will produce the following printout using only a single `print()` function call.

```
Hello
Hello again
```

References

1 = Computer Science poem from https://www.poetrysoup.com/poem/comp_sci_and_sci-fi_---_a_nonsensical_poem_876229

4. Making decisions

So far we have been loading values into variables, performing simple calculations on them, and then printing the result. This is well and good, but the true power of programming comes from the idea of selective execution. In other words, we can pose a question of our program, and on the basis of the answer, it will decide whether or not to run a particular section of code. This means programs can behave differently in different scenarios.

The most commonly used method to specify whether code to run under certain conditions is through an `if` statement. An `if` statement poses a question, and if the answer to the question is `True`, it will execute the given code that immediately follows. If the answer is `False`, it will jump over the code that follows.

A simple example that we'll dissect...

```
n = int(input("Please type a number: "))
if n < 10:
    print(f"The number {n} is less than 10.")
print("Thank you, have a nice day!")
```

The second line is where the magic happens: `if n < 10:`. We are performing a *less than* comparison between two values, the number stored within the variable `n` and the value `10`. We end the comparison by placing the colon `:` symbol which is mandatory.

The line following the `if` statement **must be indented**. The indentation is how Python knows which code should be linked to the `if` statement. If the `if` statement failed the comparison (such as a number larger than 10 was input), Python will skip over the indented code and resume running at the first line it sees that was in line with the `if` statement, in this case the "Thank you" message.

Do be careful with your indentation. It must be consistent. If the level of spacing varies it will confuse Python and you will get an error. Most Python aware editors will try to help you with your indenting and will automatically indent for you when required. To illustrate the following examples will generate errors...

```
n = int(input("Please type a number: "))
if n < 10:
    print(f"The number was {n}.")
    print(f"It was less than 10.")      # Error - change in indentation
print("Thank you, have a nice day!")
```

```
n = int(input("Please type a number: "))
if n < 10:
    print(f"The number was {n}.")
    print(f"It was less than 10.")      # Error - change in indentation
print("Thank you, have a nice day!")
```

Usual practice is to use indentation of 4 spaces. (Side note: There is a debate within Python coders as to whether spaces or tabs are better... Most editors will let you use either and actually convert your tabs to spaces. Just be consistent.)

Indentation is used by a number of features within the Python language. It is always how code gets linked to a certain line like is happening here with the `if` statement. The key is you should always increase your indentation after any line that ends with the colon `:` symbol. You reset your indentation when you want to stop applying the feature that particular line invoked.

You can (and will) indent multiple layers deep. An example looks like...

```
n = int(input("Please type a number: "))
if n > 0:
    print(f"The number is positive")
    if n < 10:
        print(f"The number is less than 10")
    if n > 100:
        print(f"The number is greater than 100")
if n < 0:
    print(f"The number is negative")
print("Thank you, have a nice day!")
```

One operation I haven't shown yet is to test if two values equal each other. The above examples use *greater than* and *less than* symbols only. This is because *equality* deserves a special mention as instinctively you might think it is just the equal sign, `=`, but it's not that simple. Run the following code. The program will not behave as you might expect...

```
n = int(input("Please type a number: "))
if n = 0:
    print("n is zero")
print("Weird eh?")
```

Python should generate an error but many languages will actually execute this code and always print the *n is zero* output. The problem is programming languages need a way to differentiate between using the equal sign `=` for *assignment* and for *comparison*. In other words: are we asking the computer to assign the value `0` to the variable `n`, or are we asking the computer if the value `n` matches the value `0`? Python gets around this by using two different symbols, one for each meaning. Assigning the value is done through a single equal sign `=`, whereas comparing values is done through a double equal sign `==`. It is a small but critical difference.

Given that, the correct way of writing the above would be...

```
n = int(input("Please type a number: "))
if n == 0:
    print("n is zero")
print("That's better")
```

If statements have quite a few other features for us to look at. For starters, we can link multiple options together. The most commonly known method is by using the `else:` key word. If the comparison asked in the `if` statement was not true, then the `else:` allows us to provide alternative code to execute.


```
n = int(input("Please type a number: "))
if n > 0:
    print("n is positive")
else:
    print("n is negative")
print("Bye")
```

While useful this doesn't actually serve all purposes. Sometimes we might have other scenarios we need to test for. A careful consideration of the above problem tells us there is a problem with the code because zero is neither positive or negative, so let's add a special case for it...

```
n = int(input("Please type a number: "))
if n > 0:
    print("n is positive")
elif n < 0:
    print("n is negative")
else:
    print("n is zero")
print("Bye")
```

The `elif` statement is an abbreviation of the phrase *else if*. We can have as many `elif` statements as we want. An example using the built in computer calendar might be...

```
from datetime import datetime
weekday = datetime.now().weekday()
if weekday == 0:
    print("Today is Monday")
elif weekday == 1:
    print("Today is Tuesday")
elif weekday == 2:
    print("Today is Wednesday")
elif weekday == 3:
    print("Today is Thursday")
elif weekday == 4:
    print("Today is Friday")
elif weekday == 5:
    print("Today is Saturday")
else:
    print("Today is Sunday")
```

Another example using the clock...

```
from datetime import datetime

name = input("What is your name? ")
hour = datetime.now().hour
if hour < 12:
    greeting = "Good morning"
elif hour < 17:
    greeting = "Good afternoon"
elif hour < 20:
    greeting = "Good evening"
else:
    greeting = "Good night"
print(f"{greeting}, {name}!")
```

(don't worry, there is a lesson discussing the Python date and time system later)

One interesting thing to observe with chaining multiple `elif` statements together is that if the time is 9am, the above won't print **Good morning**, **Good afternoon**, and **Good evening** even though 9 is less than 12, is less than 17 and is also less than 20. This is because as soon as a chain linked `if/elif/else` statement finds a positive match, it then skips the rest of the comparisons. So it will execute the **Good morning** code and skip the rest, jumping to the final `print` statement.

The final feature to draw your attention to with `if` statements is that we can use combine multiple queries together into one.

```
from datetime import datetime
weekday = datetime.now().weekday()
hour = datetime.now().hour
if weekday < 5 and 8 < hour < 16:
    print("I guess you're at school")
else:
    print("Relaxation time!")
```

The above is saying if the weekday is less than 5 (ie: Monday to Friday) and the hour is greater than 8am and also less than 4pm, then run the indented code indicating we're at school, otherwise school it is either a weekday but not school hours, or it must be the weekend, so either way we can relax.

We can join multiple comparison queries together using the keywords of `and`, `or` and `not`. To illustrate the use of `or`, we could modify the above with...

```
from datetime import datetime
weekday = datetime.now().weekday()
hour = datetime.now().hour
if weekday < 5:                                # Monday to Friday
    if 8 < hour < 16:                            # Hour is greater than 8am and less than 4pm...
        print("I guess you're at school")
    elif hour == 8 or hour == 16:                # Hour is 8am or 4pm...
        print("Travel time")
    elif hour < 7:                                # Before 7am
        print("Get ready for school")
    else:                                          # Must be 5pm or later
        print("Relax")
else:
    print("Enjoy the weekend")
```

Finally, you will recall at the end of the lesson on strings we briefly looked at a few functions that would pattern match the content of a string for us. We can use those in `if` statements to selectively execute code on the basis of a string. For instance, when we have been inputting text and converting it to a number, that code would fail if the user didn't type a number. We can now query the string content first before doing the conversion. An example follows...

```
text = input("Type a number: ")
if text.isnumeric():                            # using a string query function
    n = int(text)
    if n > 0 and n < 100:
        print(f"{n} is between 0 and 100")
    else:
        print(f"n is outside of bounds")
else:
    print("You didn't type a number")
```

Problems

1. Suppose you are creating a climate control system. Create a program that will ask the user for the current temperature and will respond as follows:
 - If the temperature is between 21 and 25, say that it is "perfect"
 - If the temperature is below 21, say that it is "activating heating"
 - If the temperature is above 25, say that it is "activating air conditioning"
2. Create a program that allows the user to input the three sides of a triangle, and then print to advise if the triangle is a Pythagorean Triple or not (ie: $a^2 + b^2 == c^2$).
3. Write a program to check a triangle is equilateral, isosceles or scalene. An equilateral triangle is a triangle in which all three sides are equal. A scalene triangle is a triangle that has three unequal sides. An isosceles triangle is a triangle with (at least) two equal sides.
4. At the end of the lesson was code that demonstrated how to check if the content of an input string is numeric. The code provided works for integers but not for floats, as the presence of the dot character for the decimals means the `isnumeric()` function returns False. Look back at the *String functionality* section of the lesson on Strings and solve how you can create a check for text input that can be safely converted to a float.
5. Create a program for a professor can input a percentage mark for a test, and the program prints the grade letter the student achieved. The professor uses 80% for an A grade, 65% for a B grade, 50% for a C grade, 35% for a D grade, and below that for an F grade.

5. Repetition

In addition to making decisions via the `if` statement, the next most commonly used and seen tool in the programming world is the idea of the loop. There are two common types of loops: the `while` loop and the `for` loop.

Generally you would use a `while` loop when you don't know in advance the number of times you wish to iterate over your code, you can use the `for` loop when it is clearly known in advance the number of loops you wish to have.

In Python, a `while` loop is syntactically very similar to an `if` statement. The difference being that the indented code will be executed repetitively until the comparison stops being `True`. A simple example...

```
import random

print("I am a dice. How many rolls will it take for me to get a 6?")
n = random.randint(1,6)
while n != 6:
    print(n)
    n = random.randint(1,6)
print("Bye!")
```

Another simple example... This will count the number of times you enter the number 2.

```
twos = 0
text = input("Type a number (or blank line to stop): ")
while not text == "":
    num = int(text)
    if num == 2:
        twos = twos + 1
    text = input("Type a number (or blank line to stop): ")
print(f"You entered two {twos} times.")
```

You can use `and` and `or` keywords to combine multiple comparisons together just like `if` statements as well.

```
print("I will keep going while you enter numbers between 0 and 100")
n = int(input("Enter a number: "))
while n >= 0 and n <= 100:
    print(f"The number {n} is between 0 and 100")
    n = int(input("Enter a number: "))
print("Good bye")
```

Problems (while loops)

Having **while** loops combined with **if** statements allow us a range of interesting programming opportunities. See how you go with this problem set!

1. Write a counting program, that continues until the user enters the number 0 to stop. Every time a number is entered, add it to a total, and increase a counter that keeps track of the number of numbers. Once the user enters 0 to terminate, use the running total and counter to determine the average value of the numbers entered.
2. Write a program that loops continually until the user enters an empty string. For each text string entered, count the number of vowels in the text (a, e, i, o or u). Keep a running total of all vowels. Once the user enters an empty string, print the total number of all vowels.
3. Write a counting program, that continues until the user enters the number 0 to stop. Count the number of positive numbers that are divisible by 7. Print the count after the user has entered 0.
4. Create a simple number guessing game. The program needs to work as follows:
 - The computer picks a random number and stores it as a secret number
 - Ask the user to guess the number
 - If the guess is higher than the secret number, print the message "too high"
 - If the guess is lower than the secret number, print the message "too low"
 - If the guess is correct, print the message "you are correct!"
 - To use a while loop to keep the game going until the correct guess has been made
 - Bonus points: Can you keep count of the number of guesses it takes the player to get it correct?

You may need to check the lesson on numbers to remind yourself how to generate a random number.

For loops (over a range)

As already mentioned, **while** loops are not the only types of loops that exist within Python, we also have the **for** loop. As stated previously, typically a **while** loop does not know in advance how many times it will execute (hence it checks the condition every time), generally a **for** loop does know the number of iterations that will occur in advance.

Here is a simple example

```
print("I can count to 100!")
for n in range(100):
    print(n+1)
print("Bye!")
```

The above code will loop through the indented code with values of **n** of 0, 1, 2, 3.. through to 99. The iteration starts at 0, goes up by 1 each loop, and runs while the value is less than the number given to the range function.

For loops do not have to increment by increases of 1 each time. This **for** loop will run from 0 until the value 50, increasing by 5 each iteration...

```
for n in range(0, 50, 5):
    print(n)
print("Bye!")
```

When we provide three values to the **range()** function, their meanings will be interpreted as **range(from, up_to, increment_by)**.

This even works for counting down! To count from 100 down to 1, we could use **range(100, 0, -1)**.

Problems (for loops)

1. Write a counting program, that asks the user to enter a *from* number and a *to* number, which will then count from the first until the second number.
2. Write a program which iterates the integers from 1 to 50. For multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".
3. Write a program which will find all such numbers which are divisible by 7 but are not a multiple of 5, between 2000 and 3200 (both included).
4. Write a program that asks the user for "size of arrow to create" and then construct the following pattern. The following example represents a size of 5...

```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

5. The fibonacci sequence is created by summing the two previous numbers together. The first 10 numbers in the sequence are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55. Use a `for()` loop to create a program that will calculate the n-th number of the sequence. For instance, if asked for the 8th number, it should provide the answer of 21.

6. Lists

A list (known as an array in most other languages) allows us to store a list/set/collection of values all assigned to one variable identifier. They are very useful when we have a collection of values that are similar in nature and that will be processed in the same manner.

For example, supposed we are keeping record of test scores obtained by a group of students. Without a list we could use something like the following:

```
score1 = 59
score2 = 92
score3 = 85
score4 = 61
score5 = 78
```

Supposed we want to calculate the highest, lowest and average score? That would look like...

```
highest = score1          # Initially set highest to the first value
if score2 > highest:
    highest = score2
if score3 > highest:
    highest = score3
if score4 > highest:
    highest = score4
if score5 > highest:
    highest = score5
lowest = score1           # Initially set lowest to the first value
if score2 < lowest:
    lowest = score2
if score3 < lowest:
    lowest = score3
if score4 < lowest:
    lowest = score4
if score5 < lowest:
    lowest = score5
average = (score1 + score2 + score3 + score4 + score5) / 5
print(f"The highest score was {highest}, the lowest was {lowest} and the average was {average}")
```

You can see that the whole process will quickly get very tedious. There will be a lot of copy-and-pasting-and-renaming of code going on. Imagine if we needed to scale this up to 100 students for an entire year group? Unmanageable and error prone!

Enter the list!

The equivalent task using lists might look like

```
scores = [59, 92, 85, 61, 78]
highest = scores[0]      # Initially set highest to the first value
lowest = scores[0]       # Initially set lowest to the first value
total = 0                 # Running total for calculating the average later
for value in scores:     # Iterate through each `value` within `scores`
    if value > highest:
        highest = value
    if value < lowest:
        lowest = value
    total = total + value
average = total / len(scores)
print(f"The highest score was {highest}, the lowest was {lowest} and the average was {average}")
```

Our scores array can easily contain 1000s of records and we would not have to change a single line of the calculations code! Arrays can be extremely useful!

Just like other variables, lists (or arrays), don't have to be just numeric. We can use strings or other data types in lists as well.

```
primes = [1, 2, 3, 5, 7, 11, 13, 17, 19, 23]
vowels = ["A", "E", "I", "O", "U"]
starwars = ["Luke", "Han", "Leah", "Obi-wan", "Yoda", "Rey", "Finn"]
empty_list = []
```

To create a list in Python is as simple as using the square bracket notation as illustrated above. An empty set of square brackets will create an empty list that we can add to later. Items in the list are separated by commas.

Looping over a list

There are a few different ways to loop over a list.

If we have a list called `scores` such as above, we can access the number of items in the list by using `len(scores)`. We can also access each individual element using an index values such as `scores[0]` which would obtain the first value. As such, we can loop over a list using a while loop in this way...

```
scores = [59, 92, 85, 61, 78]
n = 0
while n < len(scores):
    print(f"Item {n} is { scores[n] }")
```

We can also use the `range()` function of a for-loop to re-write the same code in this way...

```
scores = [59, 92, 85, 61, 78]
for n in range(len(scores)):
    print(f"Item {n} is { scores[n] }")
```

Most of the time, however, we can use a special type of for-loop that is designed to automatically iterate over a list of values. The syntax for that would look like this...

```
scores = [59, 92, 85, 61, 78]
for item in scores:
    print(f"Item is { item }")
```

The only disadvantage of the third method is you don't have access to the index (position) numbers of each item. If you do need that number the second method is probably best.

Querying a list

This is a very powerful feature that I use a lot. You can query if an item exists inside a list. A simple query to see if a given value is present in the list.

```
starwars = ["Luke", "Han", "Leah", "Obi-wan", "Yoda", "Rey", "Finn"]
if "Luke" in starwars:
    print("Luke is in starwars")
else:
    print("Luke is not in starwars")
```

Question: Combined with a while loop receiving user input, how could you build a list ensuring you don't have any duplicates with this method?

Manipulating a list

There are two main ways to add items into a list. The `.append()` function will add a value to the end of the list, where as the `.insert()` function can be used to add a value at any given position marker within the list.

```
starwars = ["Luke", "Han", "Leah", "Obi-wan", "Yoda", "Rey", "Finn"]
starwars.append("BB-8") # Append to end of the list
starwars.insert(0, "R2-D2") # Insert to position 0
```

The starwars list will now look like...

```
['R2-D2', 'Luke', 'Han', 'Leah', 'Obi-wan', 'Yoda', 'Rey', 'Finn', 'BB-8']
```

We can combining lists together using the concatenation `+` operator...

```
starwars = ["Luke", "Han", "Leah", "Obi-wan", "Yoda", "Rey", "Finn"]
darkside = ['Vader', 'Palpatine']
characters = starwars + darkside
```

We can remove items by value using the `.remove()` function, or by position location using the `.pop()` function as demonstrated...

```
starwars.remove("Han") # Remove by item value
starwars.pop(0) # Remove by item position
```

Finally we can overwrite an existing item by referencing it's index location in an assignment operation...

```
conflicted = ['Anakin', 'Kylo']
conflicted[0] = "Darth Vader" # He turned!
```

Sublists

Lists have many of the same features of strings (which is really just a list of characters) to query them and get sub-parts from. The notation that works to select part of a string works exactly the same with elements of a list.

What is the value of `partial` after each line?

```
starwars = ["Luke", "Han", "Leah", "Obi-wan", "Yoda", "Rey", "Finn"]
partial = starwars[3:]
partial = starwars[:3]
partial = starwars[-1:]
```

Other list functions

Python has a few handy functions that allow us to quickly process commonly sought information from lists...

```
numbers = [36, 9, 13, 71, 58, 95, 22]
result = min(numbers)      # Get minimum value from the list
result = max(numbers)      # Get maximum value from the list
result = len(numbers)      # Get the length (number of items) of the list
result = sum(numbers)      # Get the sum of all values in the list
numbers.sort()             # Sorts the current list
numbers.reverse()          # Reverses the current list
position = numbers.index(13) # What position does the value 13 appear at?
occurrences = numbers.count(9) # How many occurrences of the value 9 is there?
```

The `sort()` function will also sort strings into alphabetical order.

Be aware that if your strings contain numeric data you may get strange results. Numerically the following should stay in their current order, but when sorted according to their string value you will get `[' 2', '03', '1']`. Spaces precede numbers, and then the `0` is being compared to the `1`, so it comes first.

```
data = ["1", " 2", "03"]
data.sort()
print(data)
```

You will have similar issues with `min()` and `max()` as well.

Splitting and joining lists

One useful trick is to be able to split a string into a list. A meaningless example would look like...

```
saying = "May the force be with you"
words = saying.split(" ")
print(words)           # ['May', 'the', 'force', 'be', 'with', 'you']
```

A more useful example would be to process a date and turn it into three integers for the day, month and year...

```
mydate = input("Enter a date as dd/mm/yyyy: ")
parts = mydate.split("/")
day = int( parts[0] )
month = int( parts[1] )
year = int( parts[2] )
```

A simple practical example of this would be an age calculator like this...

```
from datetime import datetime
text = input("Your date of birth as dd/mm/yyyy:")
parts = text.split("/")
day = int( parts[0] )
month = int( parts[1] )
year = int( parts[2] )
birthday = datetime(year, month, day)
today = datetime.now()
age = today.year - birthday.year
if birthday.month > today.month:
    age = age - 1
elif birthday.month == today.month and birthday.day > today.day:
    age = age - 1
print(f"You are {age} years old")
```

You can also do the reverse by taking a list and join all the elements together to form one string. With this method, you provide a string indicating character(s) you want to use to join the string together with. To just have a space between each item in the list would look like this...

```
parts = ["I", "am", "Groot"]
groot = " ".join(parts)
print(groot)           # 'I am Groot'
```

Another example of this is if you intend on writing the string to a file, you may want to join a list of items together separated by the new line character. Such an example would be...

```
lines = ["Roses are red", "violets are blue", "if you've installed facebook", "they're  
spying on you"]
poem = "\n".join(parts)
print(groot)
```

Will print the following...

```
Roses are red
violets are blue
if you've installed facebook
they're spying on you
```

(1)

Problems

1. Write a program that will start with an empty list, and using a `while` loop, continually prompt the user to enter a new number which will be added to the list. Stop once they enter the number 0. You will use the solution to this to create your lists for several of the later problems in this set.
2. Using your code from question 1: Write a program where the user enters a list, and once fully entered, will print the total of all the items in a list added together.
3. Use the code from question 1: Copy and modify it for creating a list of strings, where the loop finishes when an empty string is entered
4. Using your code from question 3: Have the user enter a list of strings. Count the number of strings where the string length is 2 or more and the first and last character are same.
5. Using your code from question 3: Have the user enter a list of strings. Remove any duplicates that exist in the list (hint: use the `.count()` function).
6. Using your code from question 3: Write a function that takes two lists and returns True if they have at least one common member.
7. Write a program to print a specified list after removing the 0th, 4th and 5th elements. Sample List :
`['Red', 'Green', 'White', 'Black', 'Pink', 'Yellow']` Expected Output : `['Green', 'White', 'Black']`
8. Write a program to print the numbers of a specified list after removing even values from it.
9. Write a program to select an item randomly from a list, which is then removed from the original list so it can't be re-drawn (just like a deck of cards scenario)
10. (challenge question) Write a program for computing primes upto 1000. Hint: Google for the Sieve of Eratosthenes

References

1 - <https://community.spiceworks.com/topic/2146585-computer-poems-funny-limerick-haiku-and-roses-are-red-verses>

7. Computational thinking

It's time to take a break from some of the mechanics of programming for a while to discuss the thinking skills involved in solving programming problems. These thinking skills are commonly known as **computational thinking**. If you search the topic online, you will probably come across this four concepts:

1. Decomposition - Can I divide this into sub-problems?
2. Pattern recognition - Can I find repeating patterns?
3. Abstraction - Can I generalise this to make an overall rule?
4. Algorithm design - Can I design the programming steps for any of this?

To illustrate what these concepts are, it is best to walk through an example. To get started, these are the steps you are going to follow:

1. Start with a small, human solvable version of the problem.
2. Look at your problem, looking to identify where you can use any of the 4 computational thinking prompts outlined above until you devise a solution.
3. Test your proposed solution on larger, real-world versions of the problem.
4. Adapt and repeat until your program is complete.

A good example is to consider how to create a program that will sort a list of numbers (such as the following) into ascending order.

16 23 63 36 67 60 42 15 24 85 90 6 18 46 32 17 61 88 47 64 62 19 80 18 24 60 47 52 48 21 12 70 95 20 35 84 48 66 24 75 38 55 539 24 77 34 4 12 35 45 33 5 92 32 89 93 40 21 65 35 63 82 92 66 92 52 44 36 41 51 27 32 32 47 26 92 98 31 2 11 90 64 99 68 55 73 24 35 94 4 80 44 48 98 2 67 24 25 1 39 18 93 49 90 6 81 100 53 29 78 479 74 63 11 44 21 100 40 51 39 62 12 39 77 27 73 20 27 48 115 40 22 43 78 62 56 58 12 69 79 10 43 49 48 82 31 74 96 56 89

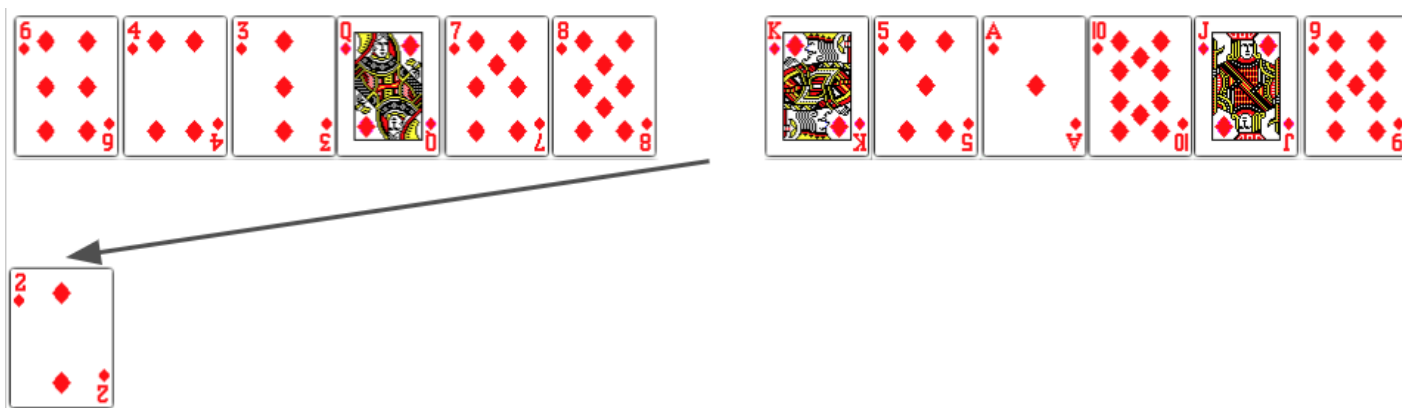
So, by applying step 1, let's create a small, human solvable version of the problem. This is important because an enormous infinitely large set of numbers is too overwhelming to think about. We do know how to sort playing cards though, so let's start with that.



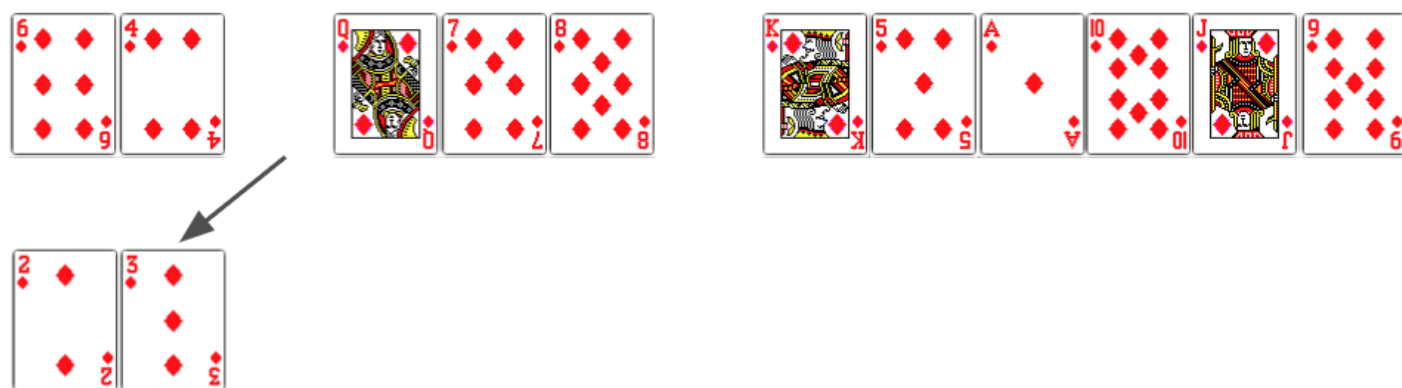
If you had to write a set of instructions for someone who had never sorted cards before, what would you write? I suggest you attempt to write a set of instructions yourself before proceeding any further.

This is my attempt at documenting the human process of sorting cards... see how it compares to what you came up with...

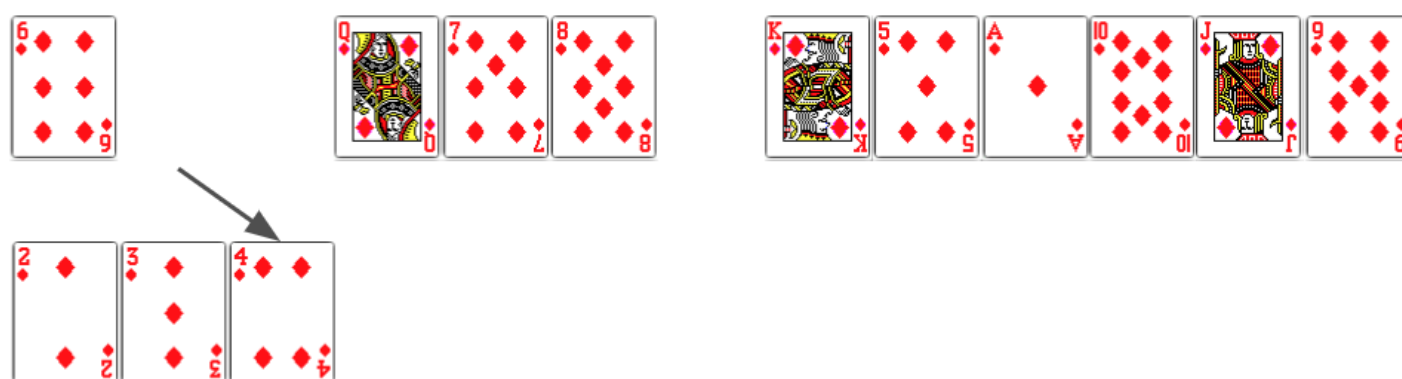
First action...



Second action...



Third action...



And so on...

So what did I actually do? I could write my instructions like this...

```
1. Find the 2, move it to the sorted set
2. Find the 3, move it to the sorted set
3. Find the 4, move it to the sorted set
...etc
```

The concept of **abstraction** requires removal of unnecessary complexity so we can create a general model or rule. This sequence of steps is currently very specific to our sample problem. Remember, the ultimate goal is to sort any set of numbers not just a set of cards, so the procedure needs to be rewritten in generic terms. For instance, refer to the first card or last card, rather than the "2 of diamonds". After all, what happens when we want to sort 1'000'000 numbers of different sizes? We want a rule we can use in all scenarios if possible.

With this in mind, I rewrite my instructions to this...

```
1. Search through the values, find the lowest, move it to the sorted set
2. Search through the values, find the lowest, move it to the sorted set
3. Search through the values, find the lowest, move it to the sorted set
...etc
```

While this is an improvement, there is still more we need to do. Looking at our four concepts of computational thinking again, the one that should jump out at you is the idea of **pattern recognition**. There is clearly a repeating pattern in our set of instructions, so let's fix that. When using a repetition construct in programming, we generally need to ensure we are clear about the terminating scenario (when do we start and/or stop the repetition?), so let's make sure to specify that.

```
while unsorted values remain:
    search through the values, find the lowest, move it to the sorted set
```

If I'm observant, I can easily spot I've actually got three steps happening at once as well... so let's simplify this so that there is one clear step per line.

```
while unsorted values remain:
    search through the values
    find the lowest
    move it to the sorted set
```

We have now taken our original "overwhelming problem" and broken it down into three smaller sub-problems. This is **decomposition** at work! I can now try to solve each of these separately, continuing to decompose into smaller and smaller chunks until I end up with pieces that I know how to convert into programming code.

The process of actually documenting this into a set of accurate instructions is **algorithm design**.

It's time to start attempting to solve this with code. I will take my recipe of instructions and turn them into Python code comments. I will then use those comments to help me design my program.

```
# The problem
values = [6, 4, 3, 12, 7, 8, 2, 13, 5, 14, 10, 11, 9]

# while unsorted values remain:
#   search through the values
#   find the lowest
#   move it to the sorted set
```

When I look at this, I realise I know how to do the loop, so I don't need to decompose that any further. It's a while loop that will run while the number of items in the `values` list is greater than zero...

```
# The problem
values = [6, 4, 3, 12, 7, 8, 2, 13, 5, 14, 10, 11, 9]

# while unsorted values remain:
while len(values) > 0:
    # search through the values
    # find the lowest
    # move it to the sorted set
```

Now I think about it a little further, I realise what I don't have is a place to store my sorted set. I realise that's just another list, so I will add an empty list to my program. I also know that to add a value to a list uses the append function, so I'll add that straight away too.

```
# The problem
values = [6, 4, 3, 12, 7, 8, 2, 13, 5, 14, 10, 11, 9]
result = []

# while unsorted values remain:
while len(values) > 0:
    # search through the values
    # find the lowest
    # move it to the sorted set
    result.append( lowest )
```

Notice that I'm not writing my code in a linear, top-down fashion? It's ok to jump around your code, add lines above where you are working if you realise you need something later. Designing a program beyond the absolutely trivial is never a top-down process. Don't think you have to know everything you have to write at the top of your program before you continue on. Start with what you know and move around as required.

So how to search through the values? I know a `for` loop will let me look at every value if I do something like...

```
for number in values:
    print(number)
```

And I know that `if` statements will let me compare values. So, it occurs to me that I could add a `for` loop to inspect every value, and if that particular number is lower than any I've seen so far, I could treat it as my lowest. Then as the program keeps looping, if it sees a new lower number, that can become the lowest, and so forth.

```
# The problem
values = [6, 4, 3, 12, 7, 8, 2, 13, 5, 14, 10, 11, 9]
result = []

# while unsorted values remain:
while len(values) > 0:
    # search through the values
    lowest = ?????
    for number in values:
        if number < lowest:
            lowest = number
    # find the lowest
    # move it to the sorted set
    result.append( lowest )
```

Ok, I know that a variable has to be created before I can run the loop, but what should I set it to? If I give was to put `lowest = 0`, then it would be lower than all my numbers already so it wouldn't work. What if I did `lowest = 9999`? The problem is I want a general rule that could work for any set of numbers, so what if the lowest happens to be bigger than that?

The solution is to just start with the first value from the list. This is a bit of programmers instinct that will come with practice.

```
# The problem
values = [6, 4, 3, 12, 7, 8, 2, 13, 5, 14, 10, 11, 9]
result = []

# while unsorted values remain:
while len(values) > 0:
    # search through the values, find the lowest
    lowest = values[0]
    for number in values:
        if number < lowest:
            lowest = number
    # move it to the sorted set
    result.append( lowest )
```

I notice that my `for` loop and `if` statement section take care of two jobs, so I've rearranged my comments to match.

There's one last problem however. I haven't actually "moved" a number into the sorted set, I "append" or "add" a number to the sorted set. Moving is actually a two step process. I need to remove it from the original list and then add it to the new list. It's ok if you didn't spot this mistake until you ran your program... that's the whole point of running tests on your code, to help spot what you are still missing.

Let's add that final step...

```
# The problem
values = [6, 4, 3, 12, 7, 8, 2, 13, 5, 14, 10, 11, 9]
result = []

# while unsorted values remain:
while len(values) > 0:
    # search through the values, find the lowest
    lowest = values[0]
    for number in values:
        if number < lowest:
            lowest = number
    # move it to the sorted set
    values.remove( lowest )
    result.append( lowest )

print(result)
```

And, that's it. Our program is done.

I will preface that this not a perfect sort algorithm at all. It is quite an inefficient algorithm. That said, it largely gets the job done and will suffice for our purposes of illustrating the computational thinking process.

Some final thoughts when facing a daunting programming problem:

1. Just start. A blank screen can be scary, so put something, anything, on screen. It doesn't matter if you end up deleting it all later, just start coding!
2. Don't start at the start. As discussed earlier, jump around your code. Start writing the bit you can figure out and go from there.
3. Start with something you know. This might be the user interface, perhaps some print statements or the input commands.
4. Don't be afraid to Google. When you do search online prioritse results from sites that are reputable for programmers such as stackoverflow.com.
5. Test and print a lot. You won't get it all in one hit. Add a thousand `print()` statements into your code to see what different variables are doing, or when different lines run.

Good luck and remember to have fun! Programming can be frustrating at times, but it is extremely rewarding as well when you persevere with a problem long enough to finally figure it out.

Problems

Mastering the art and skill of programming requires good problem solving skills. The goal of this lesson in computational thinking is to help you develop those skills. Now you read through a walk-through of the process, it is time to put theory into practice. Once you complete these problems you'll feel accomplished and ready to take on more complicated projects.

Caesars cipher

Julius Caesar created one of the first own encryption algorithms. It used a substitution method where each letter was replaced by another a fixed number of letters across from the original.

The amount each letter shifts is known as the cipher key value. A cipher key with a value of 1 would be the letter "a" would shift to become the letter "b" and so forth. Note in this instance the letter "z" would wrap back to the start to become the letter "a".

Program an implementation of the Caesar cipher that inputs (a) the plain text and (b) a cipher key value, and then outputs the correct cipher text. You should maintain upper/lower casing and ignore punctuation or digits.

Example input	Cipher key	Example output
attack	3	dwwdfm
defend the east wall of the castle	1	efgfoe uif fbtu xbmm pg uif dbtumf
captain my captain	5	hfuyfns rd hfuyfns

- Your program must include appropriate prompts for the entry of data.
- Error messages and other output need to be set out clearly.
- All variables, constants and other identifiers must have meaningful names.

Vigenère cipher

Another early cipher, Vigenère used the letters of a key-word to determine how many positions an each letter shifted.

The premise behind the cipher is that it can be encoded or decoded using a key word or phrase. For example, let's say you wanted to encode the message **HELLO** using the keyword **ONE**. In this case, the keyword is shorter than the plain text, so you start repeating the keyword again until you have enough characters. Eg:

```
Plaintext: HELLO
Keyword:   ONEON
```

The first character in the phrase is **H** and the first character in the keyword is **O**. To find the first character in your encoded phrase simply shift the plain text character by the place value of the relevant key word letter value (a=0, b=1, c=2, d=3 etc). In this example, because O is the 15th letter of the alphabet, the **H** in **HELLO** is shifted 14 places to become **V**. The step would be repeated for every subsequent character in the plaintext.

```
Plaintext: HELLO
Keyword:   ONEON
Ciphertext: VRPZB
```

Example input clear text	Key word	Example output
defend the east wall of the castle	help	kiqtuh ewl ilha alas sq ioi npzxwt
My name is Bond. James Bond.	secret	Ec prqx aw Dfrw. Beovw Ugrf.

- Your program must include appropriate prompts for the entry of data.
- Error messages and other output need to be set out clearly.
- All variables, constants and other identifiers must have meaningful names.

Rail fence cipher

To quote from the wikipedia article on the Rail Fence cipher:

In the rail fence cipher, the plain text is written downwards and diagonally on successive "rails" of an imaginary fence, then moving up when the bottom rail is reached. When the top rail is reached, the message is written downwards again until the whole plaintext is written out. The message is then read off in rows. For example, if 3 "rails" and the message 'WE ARE DISCOVERED. FLEE AT ONCE' is used, the cipherer writes out:

```
W . . . E . . . C . . . R . . . L . . . T . . . E
. E . R . D . S . O . E . E . F . E . A . O . C .
. . A . . . I . . . V . . . D . . . E . . . N . .
```

Then reads off to get the ciphertext:


```
WECRLTEERDSOEEFEAOCAIVDEN
```

Note that this particular example does NOT use spaces separating the words. The decipherer will need to add them based on context. If spaces are shown in the ciphertext, then they must be included in the count of letters to determine the width of the solution grid.

Keys can also be used in this cipher eg In this example shown above have Key=3 which means there is three rails, or three lines of text.

More info: https://en.wikipedia.org/wiki/Rail_fence_cipher

Tic-tac-toe

This exercise requires you to read input in from a user, perform some logic on  it, and spit out some new information back to the user. You need to understand how to keep repeating a loop until a goal is met. You also need to understand how to make the program make decisions, as this will be a game of player verses computer.

Here are the project goals:

1. Assume the user (player) is an X and the computer is a O
2. Ask the player where they want to place their X
3. The computer places an O
4. Output the 9 tiles showing where each player went
5. Ask the player where they want to place their X
6. Repeat until a winner is determined

Sound pretty simple? Go and make it! Then play it. Try and break it. Put an X where an O is.

What happens when there is a "Cat" game (no winner)? Here's what I am suggesting for the final product:

- Your program must include appropriate prompts for the entry of data.
- Error messages and other output need to be set out clearly.
- All variables, constants and other identifiers must have meaningful names.

Credit card number validation

The Luhn algorithm or Luhn formula, also known as the “modulus 10” or “mod 10” algorithm, is a simple checksum formula used to validate a variety of identification numbers, such as credit card numbers, IMEI numbers, National Provider Identifier numbers (wikipedia).

The Luhn test is used by some credit card companies to distinguish valid credit card numbers from what could be a random selection of digits.

Those companies using credit card numbers that can be validated by the Luhn test have numbers that pass the following test:

- Double every second digit starting from the first, ie: the 1st, 3rd, 5th, 7th ...
- If any value is now greater than 9, sum their individual digits together. (*For example if you had originally doubled 7, this would give a new value of 14, so you would sum 1+4 to result 5.*) **except the last value**
- Sum all the new values together.
- If the modulus ten of your sum total is zero, you have passed the Luhn algorithm test.

Worked example...

Step																
Card number	4	9	1	6	8	3	2	4	7	1	4	0	6	2	0	8
Double the odd placed digits	8		2		16		4		14		8		12		0	
Sum the digits if >9 except the last					7				5				3			
Final value for each digit	8	9	2	6	7	3	4	4	5	1	8	0	3	2	0	8

Sum of $8+9+2+6+7+3+4+4+5+1+8+0+3+2+0+8 = 70$ $70 \% 10 == 0$... therefore passed!

Some fake credit card numbers you can use for testing purposes...

VISA	MasterCard	American Express (AMEX)
4916832471406208	5408608073972181	349916382888946
4539515831865208	5448131672611698	379279126081887
4556019822708469278	5345203118153280	372209733301573

- Your program must include appropriate prompts for the entry of data.
- Error messages and other output need to be set out clearly.
- All variables, constants and other identifiers must have meaningful names.

Change calculator

Given an amount of money (expressed as an integer as the total number of cents, one dollar being equal to 100 cents) and the list of denominations of coins (similarly expressed as cents), create and return a list of coins that add up to amount using the greedy approach where you use as many of the highest denomination coins when possible before moving on to the next lower denomination. The list of coin denominations is guaranteed to be given in sorted order, as should your result also be.

amount	coins	Expected result
64	[50, 25, 10, 5, 1]	[50, 10, 1, 1, 1, 1]
123	[100, 25, 10, 5, 1]	[100, 10, 10, 1, 1, 1]
100	[42, 17, 11, 6, 1]	[42, 42, 11, 1, 1, 1, 1, 1]

Extension

Can you adapt your algorithm so that requires produces the least number of coins. This requires more thought than may be immediately obvious.

For example, if country X had coins [50, 25, 20, 10, 1] and you needed to return 40c in change. If you always fulfil the highest coin value first you will return 7 coins being 25c, 10c, 1c, 1c, 1c, 1c, 1c; where as the least number of coins is only 2 coins being 20c, 20c.

Tip: This is a type of computing problem known as "greedy algorithms" if you'd like to research into it.

Closest cities

The closest pair problem is a "classic" algorithm for beginning coders. (The closest pair of points problem or closest pair problem is a problem of computational geometry: given n points in metric space, find a pair of points with the smallest distance between them). It has real life significance in many applications. Consider: where are the closest shops? where is the closest hospital?

Recommended prior understanding:

- Concept of latitude and longitude coordinate system
- Basic trigonometry

Given two sets of coordinates, use the haversine formula to determine the distance between them.

The haversine formula determines the great-circle distance between two points on a sphere given their longitudes and latitudes. Where

- d = the distance between the two points (along the surface of the sphere),
- r = the radius of the sphere,
- ϕ_1 = latitude of point 1
- ϕ_2 = latitude of point 2,
- λ_1 = longitude of point 1,
- λ_2 = longitude of point 2.

Then, the distance between those two points is

NOTE

- You may assume the radius of the earth is 6373 kilometers.
- You will likely need to import your Math library.
- Remember you'll need to convert your degree based coordinates to radians using the appropriate function in your Math library.

8. Functions

Functions are blocks of code that you assign a name to. You can use that name to easily run that code again whenever you need.

Functions are very useful for separating common tasks out from your main code. It allows you to avoid repeating yourself all the time which makes your code easier to maintain. Tasks like reading from a file, saving to a file, etc are all ideally suited to being chopped off into a separate function.

To start with, we can take the mathematical function that calculates the area of a circle.

```
def area_of_circle( radius ):  
    PI = 3.1415  
    a = PI * radius ** 2  
    return a  
  
print( area_of_circle( 1.0 ) )  
print( area_of_circle( 5.0 ) )  
print( area_of_circle( 8.0 ) )
```

Here we can see that the same piece of code is being used to solve three different versions of the same problem. We don't need to re-write the calculations each time, we can just use the function we defined.

The theory of the function is that it should behave in a consistent, predictable manner. For a certain set of inputs and circumstances, the function should always return the same result. The idea is that as a programmer you can treat a function as a black box - you don't need to know or care how it calculates its result, you should be able to rely on it to behave as expected and trust the result it returns. The skill as a programmer is to be able to (a) use functions others have created and (b) create your own functions for use in your code and by others. Mastering the art of thinking in this compartmentalised way will go a long way to building your prowess as a programmer.

The syntax of a Python function is outlined as follows:

- The **def** keyword denotes you are defining a function.
- You must have the parenthesis after the function name. Inside these parenthesis you will optionally list the input parameters for your function. If no parameters are required, just leave the parenthesis empty.
- Indentation is used to indicate which code belongs to the function. Return to the previous level of indentation when the function is complete.
- The **return** statement at the end of the function is optional. If there is no result to pass back to the code that called it, you can skip it. Be aware that as soon as Python encounters a **return** statement it will exit the function, even if it hasn't reached the end of the function code.

Separating functions into different files

I recommend putting functions you will frequently reuse in different files. This will make it super easy to import these into your other projects later, enabling easy re-use of code.

For example, if I created two separate Python files, `circles.py` and `main.py`, I could put my circle functions into a separate file as follows...

File: `circles.py`

```
def area(radius):
    PI = 3.1415
    a = PI * radius ** 2
    return a

def circumference(radius):
    PI = 3.1415
    circ = 2 * PI * radius
    return circ
```

File: `main.py`

```
import circles          # Import the circles file

r = int(input("What is the radius of your circle?"))

answer1 = circles.area(r)
answer2 = circles.circumference(r)

print(f"The area of your circle is: {answer1}")
print(f"The circumference of your circle is: {answer2}")
```

For this to work, you can't have spaces or hyphens in your file name as you can't use these characters in the import statement.

Optional and default parameters

It is possible to provide default values for function parameters so they don't necessarily have to be included.

The following provides a default value for the **language** parameter so if it isn't included, it will be set to **English**.

```
def greetings( name, language="English" ):
    if language == "English":
        print(f"Hello, {name}")
    elif language == "Française":
        print(f"Bonjour, {name}")
    elif language == "Español":
        print(f"Hola, {name}")
    # you get the idea...
```

If you use a **default** value of **None**, then you can test to see if it was provided by a simple **if** statement with the variable as the following demonstrates.

```
def greetings( name, language=None ):
    # Providing a language is optional in this example
    if language:
        print(f"Hello, {name}")
    else:
        print("I don't know how to greet you?!")

greetings("Jane", "Française")
greetings("John")
```

If you wish to use multiple optional parameters, you can specify the names of the parameters you do include. This is used a lot once you start coding for GUIs or other graphics systems like games, because you can have a lot of optional settings.

```
def add_text( text_content, x, y, font="Arial", size=10, color="#000000" ):
    # Write the text on screen.....
    # Requires text_content, x, y
    # The other parameters are optional
```

With the above scenario, all of the following would be valid function calls...

```
add_text( "Hello world!", 20, 50, "Comic Sans", 24, "#ff0000" )
add_text( "Hello world!", 20, 50, font="Times" )
add_text( "Hello world!", 20, 50, size=42 )
add_text( "Hello world!", 20, 50, color="#ffffff" )
add_text( "Hello world!", 20, 50, font="Times", color="#ffffff" )
add_text( text_content="Hello world!", x=20, y=50 )
```

Functions for user input validation

Functions can be a handy way to require the user to comply with our wishes to enter information in a particular manner. By the time we write the checking/validation code and the loop, user input checks can run to several lines, and it would be quite common within a simple program to want to validate the same style of input several times.

```
def input_yn( prompt ):  
    valid = False  
    response = ""  
    while not valid:  
        response = input( prompt )  
        if response == "y" or response == "n":  
            valid = True  
        else:  
            print("Only a 'y' or 'n' character are accepted, please try again.")  
    return response
```

Problems

1. Create a function `area_right_angled_triangle(base, height)` that returns the calculated area.
2. Create a function `area_non_right_angled_triangle(base, height, angle)` that returns the calculated area (remember you will need to convert the angle to radians before using it with the sine function).
3. Create a function `input_int(prompt)` intended to validate user input. It should display the prompt to the user via the normal `input()` function, but then verify that the user response was actually an integer, and keep requesting the input until an integer is provided. When satisfied, return the integer to the calling code. Hint: Remind yourself how the `.isnumeric()` function of strings works.
4. Create a function `input_float(prompt)` intended to validate user input. It should behave the same as question 3 except verify the user input is a valid float number.
5. Create a function `input_date(prompt)` that validates the user input as a valid date in the `dd/mm/yyyy` format. Bonus points if you ensure that the `dd`, `mm` and `yyyy` values make sense (ie: day should be between 1 and 31). Hint: use `.split()`.

9. Files & folders

Gaining the functionality to read and write to files opens a whole new world of possibilities for your programs. Suddenly your program can remember things between sessions. You can also load material created by others, and save information for other programs to use.

A file is simply a stream of bytes or characters written to the disk on your computer. There are lots of different ways of reading and writing to files in Python depending on what you wish to do. I'll briefly cover the basics here.

The most common way of opening a file for reading or writing purposes in Python is to use the `with` block and the `open()` function. A simple example that will save something to a file is...

```
name = input("What is your name? ")
with open("stuff.txt", "w") as f:    # Open file stuff.txt for writing
    f.write(name)                  # Write this string to the file
```

You could then read that file using this code...

```
with open("stuff.txt", "r") as f:    # Open file stuff.txt for writing
    name = f.read()
    print("Hello {name}. I remembered your name from last time")
```

The file remains open for reading or writing purposes while you remain indented in the `with` block. Once you unindent, the file will be closed.

The `open()` function takes two parameters:

- The first is the name of the file you wish to open
- The second is the mode with which to open the file. Most commonly you will use `"r"` for reading, `"w"` for writing, or `"a"` for append or adding.

The `.read()` function will read the entire content of the file in one go, and return it as a string. As most files consist of multiple lines, your next move is to probably going to be to split the file into a list of strings, one for each line. That can be easily done using the `.splitlines()` function as the following example shows...

```
with open("stuff.txt", "r") as f:    # Open file stuff.txt for writing
    lines = f.read().splitlines()
    print(f"There were { len(lines) } in the file...")
    for line in lines:
        print(line)
```

The `.write()` function does not insert new line characters for you, so if you are wanting to write multiple lines to a file you should add the `"\n"` character yourself. To illustrate, I'll do the reverse of the above: take a list of strings and turn it into a file where each string is one line...

```
content = ['Leah', 'Obi-wan', 'Yoda', 'Rey', 'Finn', 'bb-8']
save = "\n".join(content)           # Convert list to a joined string
with open("people.txt", "w") as f:  # Open file people.txt for writing
    f.write(save)                   # Write this string to the file
```

The catch with the `"w"` mode is it will **overwrite/replace** the content of your file. If, instead, you are wanting to add to an existing file use the `"a"` mode in your file `open()` function. A simple example is...

```
print("Input an empty line to quit...")
with open("stuff to remember.txt", "a") as f:
    text = input("What would you like me to remember? ")
    while not text == "":
        f.write(text+"\n")
        text = input("What would you like me to remember? ")
```

CSV files

One really handy feature is if you want to read from CSV files that were created from Microsoft Excel or Google Sheets. These would let you access data according to field names. Imagine a contacts directory like the following spreadsheet...

	A	B	C	D
1	first_name	last_name	email	phone
2	Gerta	Giannazzo	ggiannazzo0@webs.com	+852-555-1235
3	Ede	Bruton	ebruton1@gizmodo.com	+852-555-1236
4	Rowena	Shivell	rshivell2@hp.com	+852-555-1237
5	Luce	Back	lback3@deliciousdays.com	+852-555-1238
6	Amabelle	Ricarde	aricarde4@list-manage.com	+852-555-1239

The `csv` library allows us to turn the column headings (provided on the first row) into field names that we can then reference in Python.

```
import csv

with open("contact details.csv", "r") as f:
    records = list(csv.DictReader(f, delimiter=','))

for row in records:
    first = row['given_name']
    family = row['family_name']
    email = row['email']
    print(f"The person {first} {family} has an email address {email}")
```

To have a go using CSV data, google for "name and address generator csv". I have also placed one on my Github @ <https://github.com/paulbaumgarten/data-sets>.

There's obviously a lot more you can do with CSV files, they are quite useful. We'll revisit them more later.

Operating system utility functions

There is a range of operating system functionality that you will commonly use with working with files. Rather than going into too much detail, I'll just provide examples of some of the most useful functions. For more information, check the official Python docs at <https://docs.python.org/3/library/os.html>

```
# Import the Operating System module
import os

# Check if a file or folder exists of a given name
if os.path.exists("filename.txt"):
    print("The item exists")

# Check if a file exists
if os.path.isfile("filename.txt"):
    print("The item is a file")

# Check if a folder exists
if os.path.isdir("documents"):
    print("The item is a folder/directory")

# Delete a file - USE WITH CAUTION
os.remove("file.txt")

# Rename a file
os.rename("oldfile.txt", "newfile.txt")

# Create a folder - Will error if it already exists
os.mkdir("folder")

# Remove a folder - Will error if it is not empty
os.rmdir("folder")

# Get current logged in user
username = os.getlogin()

# Get the users home folder
home = os.path.expanduser("~")

# Get the users desktop folder (Note: For Windows requires minimum of Windows 7)
desktop = os.path.expanduser("~/Desktop")

# Get a list of all files and sub folders within a folder (won't give contents of sub
# folders though)
files = os.listdir( "/folder/name" )

# An example...
files = os.listdir( folder )
for item in files:
    full_item = os.path.join( folder, item)
    if os.path.isdir( full_item ):
        print(f"found folder {full_item}")
    elif os.path.isfile( full_item ):
        print(f"found file {full_item}")

# Get a list of all files within a folder, including files inside sub folders
files = []
for folder, _, folder_files in os.walk(folder):
    for item in folder_files:
        files.append( os.path.join( folder, item ))

# Get a list of all folders within a folder, including any sub folders
folders = []
for folder, sub_folders, _ in os.walk(folder):
    for item in sub_folders:
        folders.append( os.path.join( folder, item ))
```

Problems

1. Read a text file into a string, print it to the screen.
2. Read a text file into a list of strings, and print out the number of lines in the file.
3. Ask the user for the name of a file they'd like to create. Ask the user to type an input, and then save that as the content of the file.
4. Ask the user for the name of a file they'd like to create. Using a while loop, keep ask the user to type an input and only stop when they enter an empty input. Save all the lines entered as the content of the file.
5. Read a list of strings from a text file. Tell the user how many lines there are and ask them to enter a line number indicating one they would like to read. Print just the content of that line to the user.
6. Read a list of strings from a text file. Tell the user how many lines there are and ask them to enter a line number indicating one they would like to change. Prompt the user for the new content of the relevant line. Write to the file the new list of strings.

To-do app

Write a simple to-do app. Each line represents an task. The first character should be `-` if the task still needs doing, and `x` once it has been marked as complete. When the program starts, it should present the user with 3 options as follows.

```
Welcome to simple-to-dos!

* Enter the task number to mark it as complete,
* Hit enter on an empty line to quit, or
* write some text to add a new task.

The tasks currently pending are:

1 Cook dinner
2 Clean bedroom

Your input >>
```

An example of the text file follows.

```
x Write blog post
- Cook dinner
- Clean bedroom
x Make weekly tiktok
x Subscribe to Mr B TV
```


Closest cities

Return to the *Closest cities* problem in the *Computational thinking* lesson. Now that we know how files work, we are going to add some additional functionality to our solution.

Start by downloading the `cities.csv` file from <https://github.com/paulbaumgarten/data-sets>. This is a set of latitude and longitude coordinates for major cities.

Modify your program to:

- Load all the coordinates from the CSV file into your program
- Prompt the user for two city names
- Find the distance between the two given cities

Hang person

By the end of this task set you should have a simple hang-person game working.

Get the list of words to use in your game from <https://github.com/paulbaumgarten/data-sets>, it is a file called `hangperson-words.txt`.

Task 1

Create a function that can be used to hiding the letters not yet guessed of a word. The function should receive the `special_word` and `letters_guessed` as parameters, and then return the modified version that hides letters not guessed.

Example special word	Example letters guessed	Example return value
"secret"	["a", "b", "c", "d", "e"]	"_e__e_"
"elephant"	["a", "e", "s", "n", "t"]	"e_e__ant"

Task 2

Your hang-person game needs a number of "images" to draw, depicting how close the player is to game over. This can be done through multi line text strings. For instance, the final "image" could look like...

```
+---+
|
0
/ \
/ \
=====
```

You should create a function that, based on the number provided through a parameter, will draw your hang-person at various stages of the game. You can decide how many chances to give your player.

Task 3

Create a function that

- Load the words text file into a list/array
- Use the random number generator to randomly select one item from the list as the secret word
- Returns the secret word

Task 4

It's time to use the functions you created in tasks 1, 2 and 3 to make your functioning hang-person game.

10. Exceptions

Study the following code and ask, what will Python do if the user inputs `0`? Test your hypothesis.

```
denominator = float(input("Please enter a number: "))
result = 100.0 // denominator
print(f"100 divided by {denominator} is {result}")
```

Exceptions are the error events that occur when your program is running that Python can not recover from on its own. They will result you your program crashing out.

We can avoid program crashes by writing code designed to catch and deal with the exception. If we are writing a program that may result in an exception, this allows us to preempt it by Python with an alternative to take. This is referred to as catching the exception.

Catching an exception

Firstly, we can have a generic "catch every exception" response with a "try and except" set of blocks.

```
try:
    denominator = float(input("Please enter a number: "))
    result = 100.0 // denominator
    print(f"100 divided by {denominator} is {result}")
except:
    print("I can't do that!")
```

You should be aware that while possible, this failsafe catch-all of `except:` is considered poor practice and should be avoided at all costs. This is because it hides bugs in your code, after all any trivial syntax error in your code will also create an exception!

The following exsample will illustrate the problem...

```
try:
    denominator = float(input("Please enter a number: "))
    result = 100.0 // denoninator
    print(f"100 divided by {denominator} is {result}")
except:
    print("I can't do that!")
```

This code looks the same as before but has one critical different. I have inserted a deliberate bug... intentionally made hard to find... can you spot it? Hint: There is a spelling error on a variable name.

This bug means no matter what the user enters when they run the program, it will fail and produce the **I can't do that!** message. This would become very frustrating for you - the programmer - as you won't understand why your code is not working.

A better solution is to specify which errors we are anticipating and to catch those specifically.

```
try:
    denominator = float(input("Please enter a number: "))
    result = 100.0 // denominator
    print(f"100 divided by {denominator} is {result}")
except ValueError:
    print("That wasn't a number")
except ZeroDivisionError:
    print("I can't divide by zero")
```

The above code looks runs the exception handlers for two different types of exception, the `ValueError` and the `ZeroDivisionError`. Any other exception will still cause the program to crash.

The trick to this is to know what exception name is required. To find it you can either run the code in such a way as to generate the error (and thus read it from the error statement that Python generates), or check the Python Exceptions documentation at the link for the description of each official exception.

- <https://docs.python.org/3/library/exceptions.html>

Raising exceptions

Occasionally you might have a valid cause to want to Python to have an exception event.

We can use the **raise** statement to do this.

Raising a generic exception like shown here however is considered bad practice and should be avoided. For the same reason as catching generic exceptions... they hide bugs.

I generally only ever use this approach when I am developing a program and want to quickly test if a particular section works as intended or not, and so would want the program to fail. Deliberately causing your program to fail when in real-world use is very bad practice and lazy programming. You should program your way around it with code that responds to errors rather than causing them.

```
print("I am about to fail")
x = 10
if x > 5:
    raise Exception("Not allowed to have a number greater than 5")
print("You will never see me print")
```

Getting into the detail of raising exceptions is beyond the scope of what I normally teach, so I won't go into further detail here. If you are interested in more on this, I suggest getting the relevant Python documentation at..

- <https://docs.python.org/3/tutorial/errors.html#raising-exceptions>

Problems

Do not use the generic exception handler in your responses to these problems.

Q1. The length of the hypotenuse in a right angled triangle is given by the formula

$$c = \sqrt{a^2 + b^2}$$

Rearranged, the length of one side, where **c** is the hypotenuse and **a** and **b** are the two sides adjacent to the right angle, we could use...

$$a = \sqrt{c^2 - b^2}$$

Now, in reality a triangle will not end up with a length **b** longer than the hypotenuse **c**, but if the user incorrectly types lengths so that is the case, the above would result in square rooting a negative number which is not possible.

Find out what the exception name is that would be generated, and write a program to calculate the length of side **a** that will be protected against this exception.

Q2. If you have learnt the quadratic formula in mathematics, create a quadratic formula calculator that uses exceptions...

- to ensure the **a**, **b** and **c** entered by the user are numbers
- to detect if **a** is zero (would result in a division by zero)
- to detect if there are no real solutions (square root of a negative number exception)

Q3. Question 5 of the Files problem set reads *"Read a list of strings from a text file. Tell the user how many lines there are and ask them to enter a line number indicating one they would like to read. Print just the content of that line to the user"*.

- Add exception handling in case the file does not exist.
- Add exception handling in case the user asks for a line number beyond the limits of the list (ie: if there are only 5 lines and the user asks for line 6).

11. Dates & times

Dates and times are complicated. Very complicated. You may think they are simple but that's just because you've been using them all your life and you are used to them.

Ponder these questions:

- What was the date 1000 hours ago? Well... firstly we need to know the current date. 1000 hours is approximately 41 days, so how many days was in the previous month? We may need to know how many days was in the month preceeding that as well! If either of those months is February, is it a leap year? What timezone are we in? Are we affected by daylight saving? Did daylight saving time start or end during that period?
- If a person is 15 years old, how many days old are they? Again, you need to know how many leap years there were, and many other little details.

Not only all are there all the obvious complications but the calendar system is actually rarely static. Countries change tweak their calendars all the time. If you want to be able to accurately determine anything involving dates and times it is highly recommended that you use a pre-existing system to do the work for you. These are constantly updated and can handle things like changes in timezones, daylight savings, leap years, leap seconds, and all the other complications that exist.

In Python, this is the `datetime` module.

To create a datetime object in Python, we can either initialise it based on the current clock, or programmatically provide a date and/or time for it. Here are four different ways of creating a datetime object.

```
from datetime import datetime

# Create a datetime using current computer date & time
now = datetime.now()

# Create a datetime with year=2019, month=12, day=25
christmas = datetime( 2019, 12, 25 )
print(christmas)

# Create a datetime with year=1969, month=7, day=20, hour=20, minute=17, seconds=40
apollo11 = datetime( 1969, 7, 20, 20, 17, 40 )
print(apollo11)

# Create a datetime from a formatted string
# - See section below about formatting the string
text = input("What is your birthday (write it as dd/mm/yyyy) ?")
birthdate = datetime.strptime(text, "%d/%m/%Y")
print(birthdate)
```

Once we have two datetime objects we can do interesting things with them. For instance, how many days have lapsed between the two?

```
from datetime import datetime, timedelta

apollo_11 = datetime( 1969, 7, 20, 20, 17, 40 )
now = datetime.now()
diff = now - apollo_11      # diff is a `timedelta` object
print(f"Apollo 11 landed {diff.days} days ago!")
```

Or, what is the date some time off into the future?

```
from datetime import datetime, timedelta

now = datetime.now()
ten_thousand = now + timedelta( days=10000 )
print(f"10'000 days from today is { ten_thousand.strftime('%d %B, %Y') }")
```

Want to know how many days old you are?

```
from datetime import datetime, timedelta

text = input("What is your birthday (write it as dd/mm/yyyy) ?")
birthdate = datetime.strptime(text, "%d/%m/%Y")
now = datetime.now()
diff = now - birthdate
print(f"You are {diff.days} days old!")
```

Or what day of the week a particular date is/was?

```
from datetime import datetime

day_names = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]
text = input("Write a date (write it as dd/mm/yyyy) ?")
somedate = datetime.strptime(text, "%d/%m/%Y")
day_of_week = day_names[ somedate.weekday() ]
print(f"{text} is a {day_of_week}")
```

One thing several of the examples above have used is the `strptime()` or `strftime()` functions and they've contained weird strings like `"%d/%m/%Y"`. What is this all about? Again, this is because dates are complicated. Every one around the world seems to have their own different way for writing them! So because there is no one, universally agreed method, when we need Python to convert a string to a datetime or a datetime to a string, we have to specify what style of date to use. These strings specify the layout of dates and times so Python can do the conversion. The `strptime()` function will convert a string into a datetime, and the `strftime()` function converts from a datetime to a string.

The codes used within the style string follow. You can use these codes to completely customise the appearance of dates and times as you like.

Date based codes

- %a - Weekday abbreviated (eg: Sun)
- %A - Weekday full name (eg: Sunday)
- %d - Day number in month (zero padded eg: 02)
- %b - Month name abbreviated (eg: Jan)
- %B - Month full name (eg: January)
- %m - Month number (zero padded eg: 01)
- %y - Year without century (zero padded)
- %Y - Year with century (zero padded)

Time based codes

- %I - Hour 12 hour clock (zero padded)
- %H - Hour 24 hour clock (zero padded)
- %M - Minute (zero padded)
- %S - Second (zero padded)
- %p - AM or PM

Examples

```
from datetime import datetime

now = datetime.now()
print( now.strftime("%d/%m/%Y") )      # 26/01/2020
print( now.strftime("%d/%m/%y") )      # 26/01/20
print( now.strftime("%d %b %y") )      # 26 Jan 20
print( now.strftime("%A, %d %B, %Y") )  # Sunday, 26 January, 2020
print( now.strftime("%I:%M %p") )      # 10:28 PM
print( now.strftime("%H:%M") )          # 22:28
print( now.strftime("%H:%M:%S") )      # 22:28:38
print( now.strftime("%I:%M %p, %A, %d %B, %Y") ) # 10:28 PM, Sunday, 26 January, 2020
```

Get parts of dates/times

To retrieve the integer values of part of a date or time...

```
from datetime import datetime

apollo_11 = datetime( 1969, 7, 20, 20, 17, 40 )
y = apollo_11.year      # 1969
m = apollo_11.month     # 7 (July)
d = apollo_11.day       # 20
hr = apollo_11.hour     # 20 (8:00pm in 24 hr time)
mi = apollo_11.minute   # 17
se = apollo_11.second   # 40
wkd = apollo_11.weekday() # 6 (0=Monday so 6 is Sunday)
```

Replace parts of a date

You can use the `.replace()` function to take an existing date, and change one part of it. For instance, the following takes 20th June 1980 and turns it into 20th June 2019. Not very useful you may think initially, but you could use it for something like taking someones date of birth, and then replace the year with the current year to find out the day of the week their birthday will be this year, as the second example shows.

Example 1

```
from datetime import datetime
date_1 = datetime(1980, 6, 20)
date_2 = date_1.replace( year = 2019 ) # Replace the year
print(date_2) # 20/06/2019
```

Example 2

```
from datetime import datetime

day_names = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]
typed = input("What is your birthday (write it as dd/mm/yyyy) ?")
birthday = datetime.strptime(typed, "%d/%m/%Y")
now = datetime.now()
birthday_this_year = birthday.replace( year = now.year ) # Replace the year
day_number = birthday_this_year.weekday()
print(f"Your birthday this year is a {day_names[ day_number ]}")
```

Using timestamps

Some times you may find it necessary to use a *timestamp*. This is when you use an integer to represent a date or time value. Computers use timestamps for the internal process of storing date and time information. Historically this is stored as the number of seconds since the computing epoch, deemed as 01/01/1970 00:00 UTC.

```
from datetime import datetime

# Create a timestamp based on current date/time
timestamp = datetime.now().timestamp()

# Create a datetime from a timestamp
timestamp = 1563958625 # Number of seconds since 01/01/1970 00:00 UTC
july24_2019 = datetime.fromtimestamp(timestamp)
print(july24_2019)
```

Because `datetime.now().timestamp()` will always tell you the number of seconds since 01/01/1970 00:00 UTC, you could use it to track the passage of time within your program. For a completely useless example...

```
from datetime import datetime

started = datetime.now().timestamp()
for i in range(1000000):
    print( i )
finished = datetime.now().timestamp()
print(f"That took { finished-started } seconds")
```

Problems

1. Write a function that, given a string in date format, will calculate and return your age in years (obviously it should check if the day & month has past in the current year and adjust accordingly). Example `getAge("05/05/2010")` returns 9.
2. Write a function that, given a string in date format, will calculate and return the number of days until the date. `get_days_until("01/01/2021")` returns 312.
3. Write a function that, given a string in date format, will calculate and return the day of week as a string for that date. `get_day_of_week("01/01/2010")` returns "Tuesday".
4. Write a function accepting two dates that will return the number of days between the two dates. Example function call being `get_days_between("04/06/2018", "02/08/2016")`

12. Dictionaries

A Python dictionary is similar to a list in that it allows us to store multiple values against the one variable name. The difference is that instead of using an index number to reference each individual element, we can assign a name to each element. Other languages may refer to this as using a hash table, or a key-value pair.

```
# Create an empty dictionary
person = { }      # Curly braces instead of the square brackets used for lists
# Set values to your dictionary
person["given_name"] = "Paul"
person["family_name"] = "Baumgarten"
person["website"] = "https://pbaumgarten.com"
print(person)
```

To loop through all the elements of the dictionary with a for loop, we use the `.items()` function...

```
for key,value in person.items():
    print(f"field {key} has value {value}")
```

We can remove an individual element from a dictionary

```
del person["website"]
```

Or add new items at any time...

```
person["youtube"] = "https://youtube.com/pbaumgarten/"
```

Notice that the only time the curly braces was required was when declaring the dictionary for the first time. It is square brackets for everything after that.

Similar to lists, you can use the `if ... in` syntax to query if something exists in a dictionary. This becomes useful if you are loading your dictionary from a file. If you are not entirely sure if certain information has been provided, this allows you to check before assuming/using it which would generate an exception if it didn't exist.

```
person = {}
person["given_name"] = "Paul"
person["family_name"] = "Baumgarten"
person["website"] = "https://pbaumgarten.com"

if "given_name" in person:      # To check if a field name exists
    print("Yep")
if "Paul" in person.values():   # To check if a value exists
    print("yep")
```

Dictionaries have a lot of uses within Python. We will briefly look at three: CSV files, JSON files and website requests.

CSV files

One use case that I find happens a lot is to have lists of dictionaries. For instance, if you are using a dictionary to record several fields of information about a person, as shown above, it's not that big a leap to then have a list of dictionaries in order to store information on multiple people.

CSV stands for "Comma Separated Values" and is a widely used file format for exchanging data between systems. One very useful aspect to the use of CSV files is that you can use Microsoft Excel or Google Sheets to create them. This means you can use a spreadsheet to create a large data set, perhaps a contacts list, or a products list for a shopping system, or financial data, or anything really... and then export it to CSV as an easy way of bringing the data into your Python program.

Take another look back at the brief introduction to CSV files that I included when we looked at reading and writing files.

	A	B	C	D
1	first_name	last_name	email	phone
2	Gerta	Giannazzo	ggiannazzo0@webs.com	+852-555-1235
3	Ede	Bruton	ebruton1@gizmodo.com	+852-555-1236
4	Rowena	Shivell	rshivell2@hp.com	+852-555-1237
5	Luce	Back	lback3@deliciousdays.com	+852-555-1238
6	Amabelle	Ricarde	aricarde4@list-manage.com	+852-555-1239

```
import csv

with open("contact_details.csv", "r") as f:
    records = list(csv.DictReader(f, delimiter=','))

for row in records:
    first = row['given_name']
    family = row['family_name']
    email = row['email']
    print(f"The person {first} {family} has an email address {email}")
```

You may now recognise that the `row` is simply a dictionary variable. The `DictReader` object will automatically take the first row of the CSV file and use it as your field names.

In the same way you can read a CSV file into a list of dictionaries, you can also write back to one to keep any changes you make.

```
import csv

with open("contact_details 2.csv", "w") as f:
    # Either manually specify your field list
    fields = ["given_name", "family_name", "email"]
    # Or automatically generate a field list from a provided dictionary
    fields = list(records[0].keys()) # Use the dictionary of the first item in the list
    writer = csv.DictWriter(f, fieldnames=fields)
    writer.writeheader() # Will write the first row with the field names
    for row in records:
        writer.writerow(row)
```

JSON files

Another commonly used file format for exchanging data between applications is JSON. It derives from Javascript, hence the acronym Java Script Object Notation.

In any case, it is very commonly used as a format for uploading or downloading data from the web.

In terms of a data format, a JSON file is just a combination of lists and dictionaries, where one can contain another, which can contain another and so on.

One nice thing about using JSON is you can save or read from them in just a couple of lines.

```
import json

with open("contact_details.json", "r") as f:
    records = json.parse( f.read() )

for row in records:
    first = row['given_name']
    family = row['family_name']
    email = row['email']
    print(f"The person {first} {family} has an email address {email}")

with open("contact_details 2.json", "w") as f:
    f.write( json.dumps( records, indent=3 ) )
```


Website requests

This brings us on to our next use of dictionaries and JSON files, which is to actually import data from an online web based resource.

At it's most basic it can look as simple as this...

Note: The first time you want to run this, you may have to install the **requests** library as it is not included by default. See Appendix 1: How to install a library.

```
import json
import requests

url = "https://raw.githubusercontent.com/paulbaumgarten/data-sets/master/contact%20details.json"
records = requests.get(url).json()

for row in records:
    first = row['given_name']
    family = row['family_name']
    email = row['email']
    print(f"The person {first} {family} has an email address {email}")
```

The nice thing about interacting with online services, is it allows your software to tap into software systems that others have setup.

Here is a simple example. Provide it an address and it will give you the latitude and longitude coordinates.

```
"""
To manage demand on their servers, free unauthenticated use of this system is intentionally
slowed down. response times on the free ports are normally throttled to no more than 1
request per second for all un-authenticated users combined. If you want to use this service
and get faster responses, you'll have to create a paid account.
"""

import requests
import json

location = input("Type an address (including state and country): ")
url = "https://geocode.xyz/"+location+"?json=1"
headers = { 'Content-Type': 'application/json' }
response = requests.get(url, headers=headers)
info = response.json()
print("City:      "+info["standard"]["city"])
print("Country:   "+info["standard"]["countryname"])
print("Latitude:  "+info["latt"])
print("Longitude: "+info["longt"])
```

The trick to using these online services is you need to study their documentation to learn the structure of the JSON data their system will provide to you. This is known as their API documentation. Simply printing out the raw information received and going through it manually is also very helpful. Be aware that many online service providers will also require an account of some kind (to prevent against misuse of their systems) and may possibly impose a cover charge.

Here's a fun app to lighten your mood with photos of kittens and puppies!

```
import requests
import json
from PIL import Image
from io import BytesIO

headers = { "Content-Type": 'application/json' }
print("Hello, I am here to cheer you up!")
choice = input("Type cat, dog or blank to quit: ")

while choice in ["cat", "dog"]:
    if choice == "cat":
        # Random cat photo service to provide the URL of where to find a cat photo
        url = "https://aws.random.cat/meow"
        response = requests.get(url, headers=headers)
        picture_url = response.json()["file"]

    elif choice == "dog":
        # Random dog photo service to provide the URL of where to find a dog photo
        url = "https://shibe.online/api/shibes?count=1"
        response = requests.get(url, headers=headers)
        picture_url = response.json()[0]

    # Once we have a url containing a photo, download the actual photo and display it
    picture_response = requests.get(picture_url)
    if picture_response.status_code == 200:
        img = Image.open(BytesIO(picture_response.content))
        img.show()
    else:
        print("Error downloading the picture :-)")

    # That was fun, let's go again!
    choice = input("Type cat, dog or blank to quit: ")
```

If you want to know the weather for your city (or any city)... you can use the weather service provided by <https://openweathermap.org/>. Note: You will have to sign up for a free API KEY for this one to work.

```
import json
import requests
import os
from pprint import pprint
API_KEY = "-----insert-your-api-key-here-----"

# Get the user choice
city = input("What city do you want the weather for?")
# Setup the request
url = f"https://api.openweathermap.org/data/2.5/weather?q={city}&APPID={API_KEY}"
headers = {}
headers['Content-Type'] = 'application/json'
headers['Authorization'] = API_KEY
# Query the online service
response = requests.get(url, headers=headers)
data = response.json()
# View the raw response if you wish
pprint(data)
# Process the response
description = data["weather"][0]["description"]
kelvin = data["main"]["temp"]
celsius = round(kelvin-273.15)
message = f"The weather in {city} is {description} and the temperature is {celsius}"
# Print the weather information
print(message)
```

Problems

One large problem split into multiple parts...

1. Create a function that will prompt the user to input a persons name, email address and date of birth. The function should return a dictionary item containing those three pieces of information as strings.
2. Create a loop that will continually execute the function from part 1, adding the dictionary items to a `people` list each time.
3. Each time a new person is added to the `people` list, save the contents of people to a CSV file called `people.csv` and a JSON file called `people.json`.
4. Create a separate program that will read in either the `people.csv` or the `people.json` file, and will print out the list of people contained within it, and tell you the age of each person.

For example, by the end of part 3, your CSV file might look like this...

	A	B	C
1	name	email	dob
2	John B Goodenough	john@gmail.com	25/7/22
3	Jim Peebles	jim@gmail.com	25/4/35
4	Esther Duflo	esther@gmail.com	25/10/72

and your JSON file might look like this...

```
[
  {
    "name": "John B Goodenough",
    "email": "john@gmail.com",
    "dob": "25/07/1922"
  },
  {
    "name": "Jim Peebles",
    "email": "jim@gmail.com",
    "dob": "25/04/1935"
  },
  {
    "name": "Esther Duflo",
    "email": "esther@gmail.com",
    "dob": "25/10/1972"
  }
]
```

By the end of part 4, your example output might look like this...

```
John B Goodenough is 97 years old
Jim Peebles is 84 years old
Esther Duflo is 47 years old
```

Bonus points if you can tell me what those 3 people have in common 😊

13. Classes & objects

Preface: There is a lot of technical theory associated with classes and objects that I will not be going into in this section. If you study Computer Science in high school you will likely come across the theory then. This will be a brief introduction with the aim of not introducing more theory than beyond the general concept. This book is aimed to be as practical as possible.

Remember back to the lesson on Computational Thinking, abstraction is a technical term for creating a generalised model or rule. Classes and objects are a new tool for creating abstractions that we will now consider. We have used several different tools already: Variables themselves are an abstract idea. Functions and dictionaries are two quite significant abstractions we have learnt from recent lessons.

One way of thinking of a function is that it allows us to create a self-enclosed block of code that can be called with data inputs and will return a result value.

Classes and objects solve one problem that can come from using functions alone. Functions don't remember anything from the last time you ran it. Every time you run a function, you have to provide all the information it needs to complete its task. Classes and objects solve this: They can remember data between uses.

You have been using objects for a while now without me drawing your attention to it. Any time we've used a **variable** that was followed by a dot and a function of some kind, we were most likely using an object. Some examples from our previous lessons...

```
with open("somefile.txt", "r") as f:
    content = f.read()
```

The variable `f` is an object that retains a reference to the file we opened. We can then use the `.read()` and `.write()` functions that come with the object to perform operations on the file, without having to manually specify the filename every time.

```
d = datetime.now()
print( d.year )
print( d.month )
print( d.day )
print( d.strftime("%d/%m/%Y") )
```

In this example, `d` is an object that retains a reference to the current date and time. We can then run multiple lines of code to extract different parts of that, or convert it, or modify it, all without respecifying what the date and time is we want it to be using.

```
s = "Apparently I'm just an object to you"
print( s.upper() )
print( s.split(" ") )
print( s.count(" ") )
```

Even the string variable we have been using extensively is more correctly an object. As the above example shows, after creating a string, `s`, there is then a range of functions we can use that come with the variable. Each time we want to use the function, we don't have to respecify what the value of the string is that we want to convert to upper case, or to split, or to count. Python has remembered what the information was.

So hopefully the above illustrates that objects can be quite useful. We will use objects in some of the later lessons. To create graphical user interfaces (GUIs), objects are used a lot to represent items such as a screen window, a button or a text box. Similarly to create projects using databases, networks, photographic images, even computer games, all depend on the idea of objects.

The title of this lesson is "Classes and objects". The preceding few paragraphs I have written about objects. So, what is a class? To simplify somewhat, a class is the template code from which individual objects can be created later. You create code defining a class, in the same way you would create code to define a function. Once complete, you can then create objects based on that class, in the same way you would run a function based on it's previously written definition in code.

To use a real world analogy, the class represents the blueprints of a home, where as the objects are the individual homes actually constructed from the blueprints.

Enough theory, hopefully you've got the general idea now, let's get coding! How do we create a class in Python?

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

    def get_name(self):
        return self.name
```

So the above is a very simple class that doesn't really do much of use, except for allowing us to examine the syntax of what is happening.

As with other Python constructs we use indentation to signify the code that belongs to the thing we are defining, in this case a class. As we saw from the examples before, classes contain functions and so this class has two functions in it, `__init__()` and `get_name()`.

What the heck is `__init__()`??? This is a special function. The `init` stands for *initialisation*. It is the function that will... wait for it... initialise (!) the objects we make based on our class. Python will automatically look for this function when we create an object, and will run it for us.

The other question that you are probably thinking is, what is up with all the `self` stuff everywhere. This is our classes and objects remember things between their executions. `self.` is a prefix for, this thing about myself. So, the `self.name =` is to create a variable attached to ourself called name. Note the `self` variable must be the first parameter of every function within a class. You do not need to supply this variable when using objects based on the class however as Python silently provides it.

Let's look at creating some objects based on this class to see how it works...

```
alan = Person("Alan Turing")
grace = Person("Grace Hopper")

print( grace.get_name() )
print( alan.get_name() )
```

This code creates four distinct objects, all of which are based on the class `Person`, and have all the functionality of the code we created for `Person`, but each of them remember their own information separately from each other. When we run the function `.get_name()`, it is returning the name that was assigned to the `self` of the individual object instance.

Let's build a slightly more useful example...

```
from datetime import datetime

class Person:
    def __init__(self, name, birthdate_string):
        self.name = name
        self.birthdate = datetime.strptime(birthdate_string, "%d/%m/%Y")

    def get_name(self):
        return self.name

    def get_age(self):
        today = datetime.now()
        age = today.year - self.birthdate.year
        if today.month < self.birthdate.month:
            age = age - 1
        elif today.month == self.birthdate.month and today.day < self.birthdate.day:
            age = age - 1
        return age
```

This time, our `__init__()` function has the silent `self` parameter, and two parameters we must supply, `name` and `birthdate_string`. Let's set up some code that will create these objects...

```
alan = Person("Alan Turing", "29/06/1912")
grace = Person("Grace Hopper", "09/12/1906")
margaret = Person("Margaret Hamilton", "17/08/1936")
vint = Person("Vint Cerf", "23/06/1943")
```

This time, if we run the function `.get_age()` that is automatically attached to these objects, we will see that Python has remembered their birthdate for us, and will execute the function by referencing the information we provided at initialisation. As things like dates of birth do not change, it doesn't make sense to have to provide it each time, so an object can just remember it and look it up as required.

```
print( alan.get_age() )  
print( grace.get_age() )  
print( margaret.get_age() )  
print( vint.get_age() )
```

Rather than creating a problem set specific to classes and objects, the next lesson will walk you through a detailed problem that is based on classes. Please ensure you complete that.

14. Classes & objects (part 2)

Back in the lesson on files, there was a challenge question in the problem set to create a "simple" task manager.

This lesson will center around creating an updated version of that problem, this time using objects. Additionally, rather than just giving you all the code, I am going to walk you through my mental process of designing this solution to help with your computational thinking and problem solving of a task of this complexity. At no point do I give you the entire code base, so you can't mindlessly copy it out. You have to read the surrounding text to determine what each section is and where it belongs.

The first question I asked myself is what information do we want to store about individual tasks in our 'to-do' app? To avoid getting too complicated, let's stick to three items: Task description, task status, and task due date. With that decided, I'll create the start of a `Task` class...

```
from datetime import datetime

class Task:
    def __init__(self, description, due_date):
        self.description = description
        self.status = "Pending"
        self.due = datetime.strptime(due_date, "%d/%m/%Y")
```

Notice I didn't require `status` as a parameter because it doesn't really make sense to need it. Obviously if we are creating a new task, it hasn't been completed yet, so we can just initialise it to a default value of `Pending`.

Now, what functions do we want to create around the description? Probably only two: One to get the description, and we might as well make one for updating it. We might not need the second one, but it won't hurt...

```
def get_description(self):
    return self.description

def set_description(self, new_description):
    self.description = new_description
```

What functions might be required regarding the status? We will definately want to be able to query the status (thinking... "get all tasks that are not complete") and to mark the task as complete.

```
def is_pending(self):
    if self.status == "Pending":
        return True
    else:
        return False

def set_complete(self):
    self.status = "Complete"
```

Finally, functions for the due date. Again, I am simply trying to anticipate what my application may require. I may have to modify these later.

```
def is_overdue(self):
    today = datetime.now()
    if today > self.due:
        return True
    else:
        return False

def get_days_remaining(self):
    today = datetime.now()
    diff = self.due - today    # *** see note below
    return diff.days
```

note: It was at this point I realised I would need the `timedelta` module. Remembering back to our lesson on Computational Thinking where I suggested coding is not a linear or top-down process. Now I realise I needed this module, I have modified my import section accordingly to `from datetime import datetime, timedelta`.

Before building a program to use our `Task` class, there is one last function we'll add. Quite commonly, we'll want to print the content of our variables and objects. When testing and debugging programs, I tend to add a lot of extra calls to `print()` to keep track of how my program is behaving. At the moment, however, if we were to just print a task, we wouldn't get a response that is very helpful. For example...

```
task = Task("Happy Chinese new year", "25/01/2020")
print(task)
```

Gives me the output of...

```
<__main__.Task object at 0x10b796a90>
```

This print output is simply telling me the memory location that the `Task` object has been stored at within the computer. Probably not very helpful.

Just like the special built in `__init__()` function, there is another special function that, if we create it, will be used by Python when we use `print()` on the object. That special function is called `__repr__()`. So, let's create one that will give us a simple human readable summary of the task...

```
def __repr__(self):
    if self.is_pending():
        days = self.get_days_remaining()
        if self.is_overdue():
            return f"Task '{self.description}' is overdue by {abs(days)} days."
        else:
            return f"Task '{self.description}' is due in {days} days."
    else:
        return f"Task '{self.description}' is complete."
```

Now if I create a couple of dummy tasks and print them...

```
a = Task("Summer Holidays", "24/06/2020")
b = Task("Chinese new year", "25/01/2020")
print(a)
print(b)
```

I will get more meaningful output...

```
Task 'Summer Holidays' is due in 146 days.
Task 'Chinese new year' is overdue by 5 days.
```

Ok, I'm now ready to start creating the main part of my program that will use objects based on our `Task` class. Given I am going to want my task manager to have more than one task at a time, I'm going to have a list of tasks. And I want this list to print when the program starts and whenever the menu appears, so I'll create a function for that so I can print them all with one command.

```
from datetime import datetime, timedelta

# ... insert all my class Task code here ...

def print_tasks( tasks, pending_only=True ):
    # sorting based on https://stackoverflow.com/a/403426/10971929
    tasks.sort(key=lambda x: x.get_days_remaining())
    for task in tasks:
        if pending_only and task.is_pending():
            print( task ) # This can use my new __repr__() function to print a friendly
summary.
            elif not pending_only:
                print( task )

# Let's test this...
tasks = [] # empty list to store all my tasks
print("Test run 1")
tasks.append( Task("Summer Holidays", "24/06/2020" ))
tasks.append( Task("Chinese new year", "25/01/2020" ))
print_tasks(tasks)
print("Test run 2")
tasks[1].set_complete()
print_tasks(tasks)
print("Test run 3")
print_tasks(tasks, pending_only = False)
```

I thought I should sort the output by number of days remaining according to the due date while I was at it. Notice I had to do a quick google to remind myself how to do that. I've included the url of the reference I used to (a) be ethical and provide attribution to a resource I used and (b) so I know what webpage to refer back to if I have issues with that section of code. Please get into good habits and include attribution comments to your code.

Anyway, testing this function gave me the following output, so I'm happy that it seems to work...

```
Test run 1
Task 'Chinese new year' is overdue by 5 days.
Task 'Summer Holidays' is due in 146 days.
Test run 2
Task 'Summer Holidays' is due in 146 days.
Test run 3
Task 'Chinese new year' is complete.
Task 'Summer Holidays' is due in 146 days.
```

I highly encourage you to develop your programs using this same process... code a bit, test a bit... code the next bit, test it, and so forth. Don't try to write everything in one go before you test, it will just make things impossibly difficult for you.

Ok, for the next section, let's make the code that allows us to create tasks and set them as complete. While at it, I might create a simple user interface menu inside a while loop.

While coding this I also made a slight change to my `print_tasks()` function so it would provide me a task number as well.

```
def print_tasks( tasks, pending_only=True):
    tasks.sort(key=lambda x: x.get_days_remaining())
    for task_num in range(len(tasks)):
        if pending_only and tasks[task_num].is_pending():
            print(f"{task_num:3}: { tasks[ task_num ] }")
        elif not pending_only:
            print(f"{task_num:3}: { tasks[ task_num ] }")

tasks = []          # empty list to store all my tasks
action = ""
while action != "exit":
    print("")
    print("*** WELCOME TO TASK MANAGER ***")
    print_tasks( tasks )
    print("")
    print("Your options...\nnew = Create new task\nclose = Close existing task\nexit = Quit\nthe program")
    action = input("> ")
    if action == "new":
        desc = input("New task description: ")
        due = input("New task due date (as dd/mm/yyyy): ")
        new_task = Task( desc, due )
        tasks.append( new_task )
    if action == "close":
        close_num = int(input("Enter task number to close: "))
        tasks[close_num].set_complete()
```

Executing the code worked as follows...

```
*** WELCOME TO TASK MANAGER ***

Your options...
new = Create new task
close = Close existing task
exit = Quit the program
> new
New task description: Summer holidays
New task due date (as dd/mm/yyyy): 24/06/2020

*** WELCOME TO TASK MANAGER ***
  0: Task 'Summer holidays' is due in 146 days.

Your options...
new = Create new task
close = Close existing task
exit = Quit the program
> new
New task description: Chinese new year
New task due date (as dd/mm/yyyy): 25/01/2020

*** WELCOME TO TASK MANAGER ***
  0: Task 'Chinese new year' is overdue by 5 days.
  1: Task 'Summer holidays' is due in 146 days.

Your options...
new = Create new task
close = Close existing task
exit = Quit the program
> new
New task description: Starwars day
New task due date (as dd/mm/yyyy): 04/05/2020

*** WELCOME TO TASK MANAGER ***
  0: Task 'Chinese new year' is overdue by 5 days.
  1: Task 'Starwars day' is due in 95 days.
  2: Task 'Summer holidays' is due in 146 days.

Your options...
new = Create new task
close = Close existing task
exit = Quit the program
> close
  0: Task 'Chinese new year' is overdue by 5 days.
  1: Task 'Starwars day' is due in 95 days.
  2: Task 'Summer holidays' is due in 146 days.
Enter task number to close: 0

*** WELCOME TO TASK MANAGER ***
  1: Task 'Starwars day' is due in 95 days.
  2: Task 'Summer holidays' is due in 146 days.

Your options...
new = Create new task
close = Close existing task
exit = Quit the program
> exit
```

Wrapping up, it would be very useful to have this program save our tasks to a file when we quit, and then reload them when we restart, so let's add that functionality. To do this task I am going to introduce a new feature to you which is the **Pickle** module of Python. The Python docs describe Pickle as a module that "implements binary protocols for serializing and de-serializing a Python object structure." What does this mean? Basically it converts the variable contents of your objects into a sequence of bytes that can be saved to, or loaded from a file.

To add this functionality, I have added two new import statements so my entire import section now looks like this...

```
from datetime import datetime, timedelta
import pickle
import os
```

And then between my `tasks = []` and `action = ""`, I have inserted the following to check if my tasks file exists and to load it if it does...

```
tasks = []          # empty list to store all my tasks
tasks_filename = "tasks.dat"
# If existing tasks have been saved to a file, load that information
if os.path.exists(tasks_filename):
    with open(tasks_filename, "rb") as f:
        tasks = pickle.load(f)
action = ""
```

Finally, at the end of my program, prior to quitting, I have added the following to use pickle to save my tasks list to a file...

```
# Save our tasks to a file before closing
with open(tasks_filename, "wb") as f:
    pickle.dump(tasks, f)
```

Presto! A fairly basic but functional task manager! For a beginner project, this is quite complex. You are using classes and objects, Pickle file storage, the datetime library and all sorts of data manipulation. If you got this working, give yourself a high five and a pat on the back!

Just before wrapping up, I should make one thing very clear. Hopefully you noticed that what is missing is anything in the way of error or exception handling. If a user does not input a date in the `dd/mm/yyyy` format, or they mis-type a task number when seeking to close a task (or just type a task number outside of range), or there is a problem reading or writing the pickle file to disk - there is no exception handling to deal with that. This walk through is not a "complete" project, it is only a starting point. It is now up to you to finish it off.

Some thoughts for additions this project would benefit from:

- Obviously exception handling is a must
- You could add a **priority** variable to each task, such as 1=Urgent, 2=High, 3=Normal, 4=Low or similar. This could then modify the order of task printing.
- Perhaps a reminder function?

15. Networking

This is only intended as a brief introduction into networks because it is something I frequently have students ask for. Networking can very quickly get quite complex but this guide is intended to be practical and allow for quickly getting basic functionality achieved rather than providing a deep theoretical introduction. For instance, none of what I discuss here dives into the essential and enormous topic of network security. This is very much a surface level introduction with a couple of example activities.

Networked programs tend to come in two main types: peer to peer, and client server. We will focus on the client server model for now. The general work flow requires two different programs: the client and the server. **Client** programs (like your web browser), initiate a network connection with a server. The **server** program is just running continually in the background waiting for a client to request something of it. Once it receives a request, the server (a) decides if it will oblige the request (does security allow it?), and then if so, completes the requested task and sends the response back to the waiting client program.

In our internet based world, most computers are networked using the **Internet Protocol**. This provides an IP address to each network connecting device (your wifi card for instance). The client program needs to know (or have a way of finding) the IP address of the server so it can address its request. These internet addresses are like your street address. It is a unique address by which your computer can be located on the internet. Your IP address might look like **192.168.0.1** if you are running IP version 4, or **2001:0db8:85a3:0000:0000:8a2e:0370:7334** if you are running IP version 6.

There is an additional complication in that one computer may be hosting several different server programs at once. Each of these need to be able to distinguish themselves as well, and so the idea of a network **port** is also used. If the IP address is the equivalent to your street address, the port number is the equivalent to an apartment number within the building at that street address.

Finally, your server and client need to agree on a **protocol** to use. A protocol is basically just a previously agreed "language" that the two computers will use to communicate with each other by. Obviously your two computers need to be able to make sense of what the other is saying (remember that the most basic level, network communications are a simple stream of 0s and 1s).

With that basic terminology, we can get a couple of sample networking demos up and running.

Example: Text messages

Initially, rather than dealing with socket level programming where we have to create our own protocol, it will be a lot simpler to use an existing protocol, perhaps the most widely used protocol, HTTP - the language of the web.

To get started, we'll create a client/server program that will demonstrate the overall idea. We will create two programs, one for the client, and one for the server. If possible, I encourage you to run the client on one computer, and the server on a second computer where they are both connected to the same wifi network.

The flow of the two programs is as follows...

- The server runs continuously once started, waiting to receive messages.
- The client starts up, connects to the server, sends a message, and waits for a reply.
- The server, when it receives a message, displays it on screen. It then asks the user to enter a reply which is then sent to the client.
- Once the client receives the reply, it displays it on screen and then terminates.
- The server will keep running until you terminate it with Ctrl+C.

The server code...

```
from flask import Flask
from flask import request
from datetime import datetime
import json
import socket

app = Flask(__name__)
messages = []

# Execute this function when a message is received to the /talk address
@app.route('/talk', methods=["POST", "GET"])
def talk():
    global messages
    message = {}
    # Retrieve the variables sent with the request
    message["from"] = request.values["from"]
    message["content"] = request.values["content"]
    message["timestamp"] = datetime.now().timestamp()
    # If we received a message, save it
    if message["content"] != "":
        messages.append(message)
    # Send a maximum of the last 100 messages in reply
    if len(messages) > 100:
        return json.dumps(messages[-100:])
    else:
        return json.dumps(messages)

if __name__ == "__main__":
    ip_addr = socket.gethostbyname(socket.gethostname())
    print(f"The server IP address is {ip_addr}")
    app.run(host='0.0.0.0', port=9999)
```


The client code...

```
from PIL import Image
from io import BytesIO
import requests
from datetime import datetime

SERVER = input("Server address? ")
PORT = 9999
author = input("What's your handle? ")
content = input("What's your message (q = quit) ?")
last_message_received = 0
while content != "q":
    payload = {}
    payload["from"] = author
    payload["content"] = content
    # Initiate request over the network
    response = requests.get(f"http://{SERVER}:{PORT}/talk", params=payload)
    # Convert response to JSON data
    messages = response.json()
    for message in messages:
        if message["timestamp"] > last_message_received:
            time_string = datetime.fromtimestamp(message["timestamp"]).strftime("%H:%M:%S")
            author = message["from"]
            content = message["content"]
            print(f"[{time_string} {author}] {content}")
            last_message_received = message["timestamp"]
    content = input("What's your message (q = quit) ?")
```

Example: Security camera

Here is a slightly different example... What if the server was acting as a security web camera? Every time the client connects, the server will take a photo through it's camera and send it to the client. In real life you'd want some kind of security protocol in place here, rather than anyone in the world being able to access your camera, but as a demonstration it serves the task. Research question: How would you secure it? How do you know your method is "good enough"?

The server code...

```
from flask import Flask
from flask import send_file
from PIL import Image
import io
import ImageTools

camera = ImageTools.Camera()
app = Flask(__name__)

# Execute this function when a message is received to the /photo address
@app.route('/photo')
def photo():
    # based on https://stackoverflow.com/a/51986716/10971929 by (Dan Erez)
    img = camera.take_photo()           # Take a photo
    img_io = io.BytesIO()               # Create an empty byte array
    img.save(img_io, 'JPEG', quality=70) # Save the JPEG photo to the byte
    array                               # Save the JPEG photo to the byte
    img_io.seek(0)                     # Reset the array to it's starting
    index                               # Reset the array to it's starting
    return send_file(img_io, mimetype='image/jpeg') # Send the bytes to the client

if __name__ == "__main__":
    ip_addr = socket.gethostbyname(socket.gethostname())
    print(f"The server IP address is {ip_addr}")
    app.run(host='0.0.0.0', port=9999)
```

The client code...

```
from PIL import Image
from io import BytesIO
import requests

SERVER = input("Server address? ")
PORT = 9999
# Make our request of the server
response = requests.get(f"http://{SERVER}:{PORT}/photo")
# Process the response as a byte array to open an image
img = Image.open(BytesIO(response.content))
img.show()
```

Example: Socket based text messages

Sockets are the technical term to describe the process of programs talking over a network connection. An open socket is effectively the equivalent of an open phone line between your client and server.

Our previous examples used sockets, but the socket layer was hidden from us as we were using the HTTP protocol (effectively our server was running a webserver, you could have opened your server address in a web browser if you wanted).

We will now eliminate the protocol layer and deal with the raw sockets. Remember that protocols provide a language to communicate. We are getting rid of that and just have an open phone line. It is now up to us to ensure the programs on both ends of that phone line are able to understand each other.

In other words, programming with sockets gets more complex.

For a start, network traffic is unreliable. You can never be certain how many bytes will be transmitted in one data packet (a fragment of a message). Secondly, without a "protocol", we have no way of knowing when our program has finished receiving a message! (like when you have a dramatic pause mid sentence?)

This means we need a way of checking if we have received all of the message and to keep receiving new fragments until we have all of it.

There are different ways of doing this. In this example, the first four bytes of the message are being used to indicate the length of the rest of the message. We are doing this by converting an integer to an array of 4 bytes using the built in `.to_bytes()` and `.from_bytes()` commands. Four bytes gives us 2^{32} of message length capacity, or put another way, about 4 gigabytes.

In the real world, you would also need to use threading with this. At the moment, a maximum of 5 requests will be queued for processing at once, and the server will deal with the requests in the queue one at a time. Threading would add the ability for the program to process requests in the queue simultaneously (more complexity). Nevertheless, hopefully this task is a good insight as to what is happening at the byte level of network connections.

If this is something you want to get into more depth with, I recommend starting with the official Python Socket Programming Howto @ <https://docs.python.org/3/howto/sockets.html>

The server code ...

```
import socket

class Server:
    def __init__(self, port=9999):
        self.ip_address = socket.gethostbyname(socket.gethostname())
        self.port = port
        print(f"[Server] IP address is {self.ip_address}")

    def get_message(self, connection):
        # Receive up to 256 bytes and decode it into a string
        message_bytes = connection.recv(256)
        message_length_bytes = message_bytes[0:4] # first 4 bytes = message length
        # Convert those 4 bytes to an integer value
        message_length = int.from_bytes(message_length_bytes, byteorder='big')
        received_length = len(message_bytes)-4 # exclude first 4 bytes
        message_content = message_bytes[4:] # Message content after 1st 4 bytes
        while received_length < message_length:
            # Receive another 256 bytes
            message_bytes = connection.recv(256)
            received_length += len(message_bytes)
            message_content += message_bytes
        if received_length > message_length:
            message_content = message_content[:message_length]
        return message_content.decode()

    def send_message(self, connection, message):
        # Convert response to bytes
        message_bytes = message.encode()
        # Get length of bytes
        message_length = len(message_bytes)
        message_length_bytes = message_length.to_bytes(4, byteorder="big")
        # Add length information to the payload
        payload = message_length_bytes + message_bytes
        # Send
        connection.sendall(payload)

    def generate_response(self, message):
        return f"Hello. You sent me '{message}'. Did I do good? :-)"

    def listen(self):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Create socket object to
        accept connections
        s.bind(("0.0.0.0", self.port)) # Listen to this port number
        s.listen(5) # Queue up to 5 requests
        while True:
            # Note the .accept() method is BLOCKING!
            # That means Python will pause here until an incoming connection request is
            received
            (connection, address) = s.accept() # Create a socket with an
            individual client
            message_received = self.get_message(connection) # Receive the message
            print(f"Received from {address}: {message_received}")
            response = self.generate_response(message_received) # Generate our reply
            self.send_message(connection, response) # Send our reply
            print(f"Replied to {address}: {response}")
            connection.close() # Hang up / close connection

# Start up
server = Server(port=9999)
server.listen()
print("Exiting!")
```

The client code ...

You will notice quite a bit of similarity with the server code. The `get_message()` and `send_message()` functions are actually identical in the server and client, so feel free to copy-and-paste.

```
import socket

class Client:
    def __init__(self, server_addr, port=9999):
        self.server_addr = server_addr
        self.port = port

    def get_message(self, connection):
        # Receive up to 256 bytes and decode it into a string
        message_bytes = connection.recv(256)
        message_length_bytes = message_bytes[0:4] # first 4 bytes = message length
        # Convert those 4 bytes to an integer value
        message_length = int.from_bytes(message_length_bytes, byteorder='big')
        received_length = len(message_bytes)-4 # exclude first 4 bytes
        message_content = message_bytes[4:] # Message content after 1st 4 bytes
        while received_length < message_length:
            # Receive another 256 bytes
            message_bytes = connection.recv(256)
            received_length += len(message_bytes)
            message_content += message_bytes
        if received_length > message_length:
            message_content = message_content[:message_length]
        return message_content.decode()

    def send_message(self, connection, message):
        # Convert response to bytes
        message_bytes = message.encode()
        # Get length of bytes
        message_length = len(message_bytes)
        message_length_bytes = message_length.to_bytes(4, byteorder="big")
        # Add length information to the payload
        payload = message_length_bytes + message_bytes
        # Send
        connection.sendall(payload)

    def send_message_wait_reply(self, message):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Create socket object
        s.connect((self.server_addr, self.port)) # Connect
        self.send_message(s, message)
        print(f"[Client] Message to {self.server_addr}: {message}")
        reply = self.get_message(s)
        print(f"[Client] Reply: {message}")
        s.close() # Hang up connection
        return reply

# Start up
addr = input("Server IP address? ")
client = Client(addr, 9999)
msg = input("Message? ")
while msg != "":
    reply = client.send_message_wait_reply(msg)
    print(f"Received: {reply}")
    msg = input("Message? ")
```


16. GUIs

The graphical user interface system that comes built in with Python, and that works for Windows, Mac or Linux, is called Tkinter.

Tkinter makes extensive use of classes and objects. Every window is an object, all the buttons, text boxes and other "widgets" drawn onto the screen are all objects. This makes sense when you think about the level of abstraction that objects provide. By being able to create multiple objects that all have the same core functions but where each is configured independently, means we can easily create multi window programs with multiple buttons and multiple text boxes that all look different and perform different tasks.

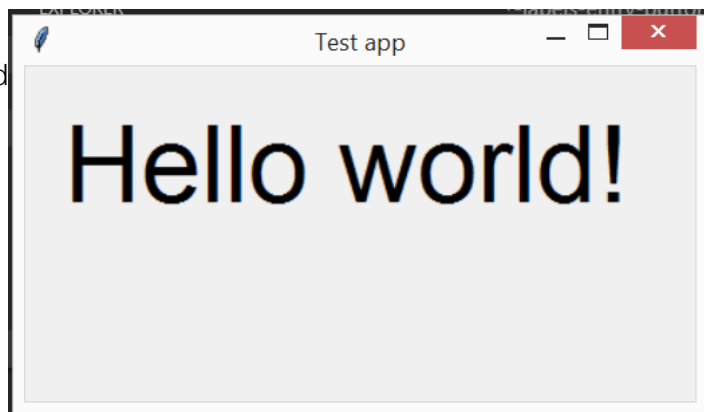
A lot of knowing how to write a Tkinter app comes down to studying the documentation. You need to know how their classes have been defined, what the functions are and what they do, to be able to use them. Lucky for you, I've done a lot of the difficult part for you for the most common scenarios. All you need to do is be able to recognise how my code is doing what it is doing. I've tried to add comments where I think something is not intuitive.

Be aware, once you start digging into the documentation you'll soon discover that Tkinter can get quite complicated. I will also point out that most documentation will urge against using the layout method I've used in these tutorials. There are three main layout methods: **pack**, **grid**, and **place**. The idea of **pack** and **grid** is that Tkinter will automatically rearrange your items on screen to suit different screen sizes and so forth. For proper commercial uses these would be the better way to go. For students who are just learning Tkinter for the first time, however, I tend to find students find the idea of the **place** method to be easier to grasp. With the **place** method you simply manually specify the **x** and **y** pixel coordinates of where you want an item to be drawn on your window. Simple, easy, clear. I recommend starting with this, and once your confidence level has grown, then look at the **pack** and **grid** methods.

A basic app

The following is about the most basic Tkinter app you can get. We are creating a class for our window, called `AppWindow`. The initialisation function, `__init__` will setup the window and draw a text label.

The `tk.Tk()` object forms the basis of every Tkinter application, and forms the `root` of our window. Every window needs to keep track of its parent or you'll quickly run into problems. With a simple one window application, that parent is the root `tk.Tk()`. You'll see this part of the code change a little once we start drawing multiple windows.



```
import tkinter as tk
from tkinter import ttk

# Pre-define some defaults
FONT_LARGE = ("Arial", 48)

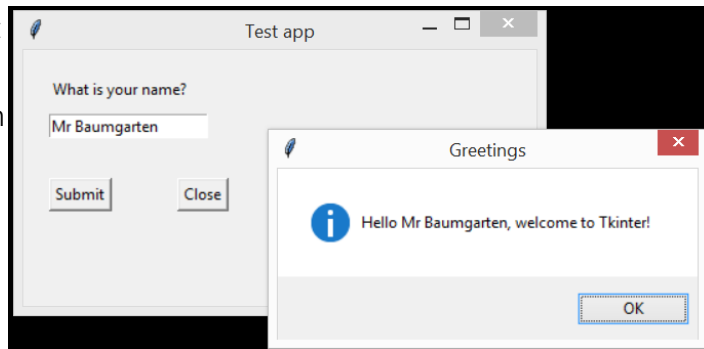
class AppWindow():
    def __init__(self, parent):
        # Create the window
        self.window = parent
        self.window.geometry("400x200")
        self.window.title("Test app")
        self.window.protocol("WM_DELETE_WINDOW", self.window.quit) # Enable the close icon
        # Create a text label and place it in the window
        self.hello_label = tk.Label(self.window, text="Hello world!", font=FONT_LARGE)
        self.hello_label.place(x=20, y=20)

if __name__ == "__main__":
    root = tk.Tk()
    app = AppWindow(root)
    root.mainloop()
```


Labels, entry box, buttons

This time we have extended our program to have a text box and two buttons. When we create the button, the `command=` parameter specifies what function we wish to run when the button is clicked. In the case of the `self.submit_button`, we are telling it to run the `self.greetings` function.

Because the `self.greetings` function is also a part of our `AppWindow` class, it can access all the internal variables belonging to `self`. Because of this, it can access the `self.name_entry` text box and get the string value of the text in that box. The function retrieves that information and then displays it in a pop up window with the `messagebox.showinfo()` function.



```
import tkinter as tk
from tkinter import ttk
from tkinter import messagebox

# Pre-define some defaults
FONT_LARGE = ("Arial", 48)

class AppWindow():
    def __init__(self, parent):
        # Create the window
        self.window = parent
        self.window.geometry("400x200")
        self.window.title("Test app")
        # Create a text label
        self.question_label = tk.Label(self.window, text="What is your name?")
        self.question_label.place(x=20, y=20)
        # Create a text entry box
        self.name_entry = tk.Entry(self.window)
        self.name_entry.place(x=20, y=50)
        self.name_entry.focus() # Put the cursor in the text box
        # Create a button
        self.submit_button = tk.Button(self.window, text="Submit", command=self.greetings)
        self.submit_button.place(x=20, y=100)
        # Create a second button
        self.close_button = tk.Button(self.window, text="Close", command=self.window.quit)
        self.close_button.place(x=120, y=100)

    def greetings(self):
        # This function is executed when the submit button is clicked
        # Retrieve the text from the entry box
        person = self.name_entry.get()
        # Display a message box
        messagebox.showinfo("Greetings", f"Hello {person}, welcome to Tkinter!")

if __name__ == "__main__":
    root = tk.Tk()
    app = AppWindow(root)
    root.mainloop()
```

List box & Simple dialog

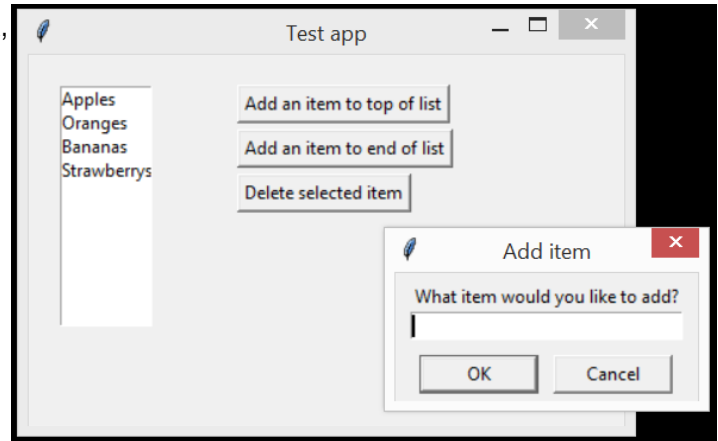
Two new features in this example program: The list box, and the use of a simple dialog to prompt the user requesting information.

Pay careful attention to the comments I have placed in the code to indicate what each section does.

You should code this example up to see how all the different elements interact. Does it work the way in which you are expecting?

Pay particular attention to `self.list.bind()`, this is the line that tells Python what function to execute when the list is clicked.

If you are using an intelligent Python editor, when you enter `simplifiedialog`, it should pop up with other examples of uses for the `simplifiedialog`. We are only using `askstring()`, but what else can it be used for?



```

import tkinter as tk
from tkinter import ttk
from tkinter import messagebox
from tkinter import simpledialog

# Pre-define some defaults
FONT_LARGE = ("Arial", 48)

# List of items to demo listbox
items = ["Apples", "Oranges", "Bananas", "Strawberrys"]

class AppWindow():
    def __init__(self, parent):
        # Create the window
        self.window = parent
        self.window.geometry("400x250")
        self.window.title("Test app")
        # Create a list box
        # -- width is characters, height is lines
        self.list = tk.Listbox(self.window, width=10, height=10)
        for item in items:
            # Add each item to the end of the list
            self.list.insert(tk.END, item)
        self.list.place(x=20, y=20)
        # -- When an item in the list is selected, execute the list_clicked function
        self.list.bind('<<ListboxSelect>>', self.list_clicked)
        # -- Give `selected` a default of -1
        self.selected = -1
        # Create some buttons
        self.add_to_top_button = tk.Button(self.window, text="Add an item to top of list",
command=self.add_to_top_clicked)
        self.add_to_top_button.place(x=140, y=20)
        self.add_to_end_button = tk.Button(self.window, text="Add an item to end of list",
command=self.add_to_end_clicked)
        self.add_to_end_button.place(x=140, y=50)
        self.close_button = tk.Button(self.window, text="Delete selected item",
command=self.delete_selected_clicked)
        self.close_button.place(x=140, y=80)

    def add_to_top_clicked(self):
        answer = simpledialog.askstring("Add item", "What item would you like to add?")
        self.list.insert(0, answer)

    def add_to_end_clicked(self):
        answer = simpledialog.askstring("Add item", "What item would you like to add?")
        self.list.insert(tk.END, answer)

    def delete_selected_clicked(self):
        if self.selected >= 0:          # Check this still isn't -1
            self.list.delete(self.selected)
            self.selected = -1          # Reset back to -1
        else:
            messagebox.showerror("Error", "Can't delete the selected item if you haven't
selected anything!")

    def list_clicked(self, e):
        print(e)
        self.selected = int(self.list.curselection()[0])          # item number selected in list
        item = self.list.get(self.selected)                        # text of selected item
        print(f"You have clicked item {self.selected} which is {item}")

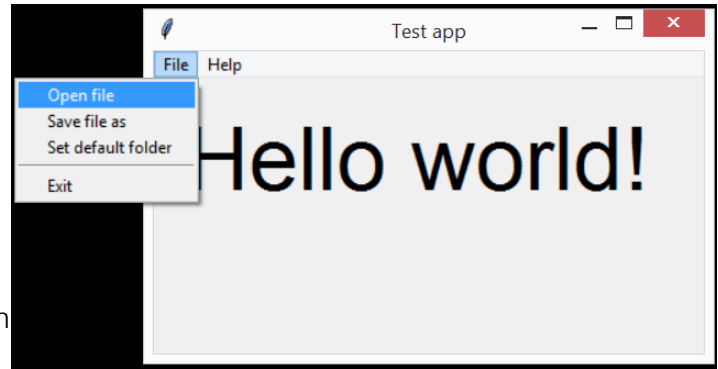
if __name__ == "__main__":
    root = tk.Tk()
    app = AppWindow(root)
    root.mainloop()

```

Menu bar & File dialog

The two features we are adding to this example are the use of the menu bar, and the file dialog popup box.

If you observe the creation of the menu bar closely, you'll see that each submenu (File and Help in this case) have their own variable created and linked back to the main `menubar`. Similar to buttons, each menu item has a `command=` parameter indicating the function to execute if that item is selected.



The `filedialog` widget is a simple one line tool that allows you to prompt the user to select file or folder locations for the purposes of opening/saving files etc.

```

import tkinter as tk
from tkinter import ttk
from tkinter import messagebox
from tkinter import filedialog

# Pre-define some defaults
FONT_LARGE = ("Arial", 48)
ALLOWED_FILES = (("JPEG files", "*.jpg"), ("PNG files", "*.png"), ("all files", "*.*"))

class AppWindow():
    def __init__(self, parent):
        # Create the window
        self.window = parent
        self.window.geometry("400x200")
        self.window.title("Test app")
        # Create a text label and place it in the window
        self.hello_label = tk.Label(self.window, text="Hello world!", font=FONT_LARGE)
        self.hello_label.place(x=20, y=20)
        # Create a menu bar
        menubar = tk.Menu(self.window)
        filemenu = tk.Menu(menubar, tearoff=0)
        filemenu.add_command(label="Open file", command=self.file_open)
        filemenu.add_command(label="Save file as", command=self.file_saveas)
        filemenu.add_command(label="Set default folder", command=self.select_folder)
        filemenu.add_separator()
        filemenu.add_command(label="Exit", command=self.window.quit)
        helpmenu = tk.Menu(menubar, tearoff=0)
        helpmenu.add_command(label="About", command=self.about)
        menubar.add_cascade(label="File", menu=filemenu)
        menubar.add_cascade(label="Help", menu=helpmenu)
        self.window.config(menu=menubar)
        # Intialise the default folder location
        self.default_folder = "."

    def file_open(self):
        filename = filedialog.askopenfilename(initialdir=self.default_folder, title="Select
file", filetypes=ALLOWED_FILES)
        print(f"Open file: {filename}")

    def file_saveas(self):
        filename = filedialog.asksaveasfilename(initialdir=self.default_folder,
title="Select file", filetypes=ALLOWED_FILES)
        print(f"Save file as: {filename}")

    def select_folder(self):
        folder = filedialog.askdirectory(initialdir=self.default_folder, title = "Select
folder containing your photos")
        self.default_folder = folder
        print(f"New default folder: {folder}")

    def about(self):
        messagebox.showinfo("About", "Copyright (c) 2019 Paul Baumgarten\nWebsite:
pbaumgarten.com")

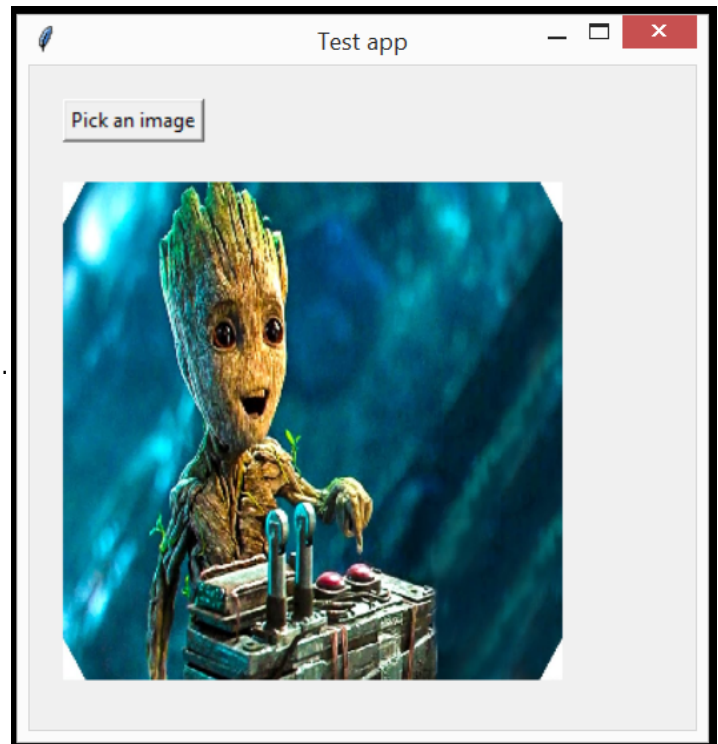
if __name__ == "__main__":
    root = tk.Tk()
    app = AppWindow(root)
    root.mainloop()

```

Images

If you don't have the Python Image Library installed for this, you will need to install it. Check the appendix for instructions on installing libraries.

A fairly simple program where the main purpose is to illustrate how you can get images (such as JPG) drawn onto the screen. It is actually done by creating a text label of the size and location you want the image to be, and then using a `.configure()` function on the label. Notice when the label is first created when using the `.place()` function we aren't just stipulating the `x` and `y` coordinate but also the `width` and `height`.



```
import tkinter as tk
from tkinter import filedialog
from PIL import Image, ImageTk

# Pre-define some defaults
FONT_LARGE = ("Arial", 48)
ALLOWED_FILES = (("JPEG files", "*.jpg"), ("PNG files", "*.png"), ("all files", "*.*"))

class AppWindow():
    def __init__(self, parent):
        # Create the window
        self.window = parent
        self.window.geometry("400x400")
        self.window.title("Test app")
        # Button
        self.pick_file_button = tk.Button(self.window, text="Pick an image",
command=self.show_image)
        self.pick_file_button.place(x=20,y=20)
        # Create a label reserved for displaying image later
        self.image_label = tk.Label(self.window)
        self.image_label.place(x=20,y=70,width=300,height=300)

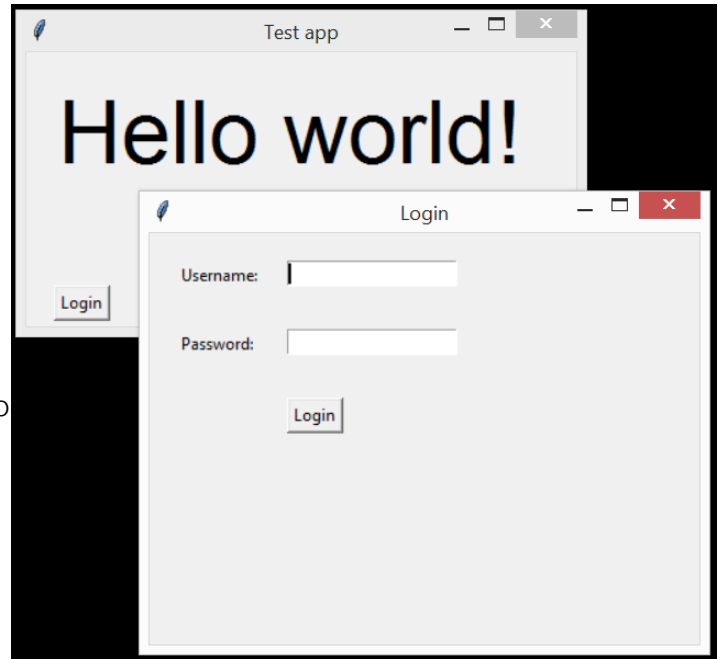
    def show_image(self):
        # Get image selection
        filename = filedialog.askopenfilename(title="Select image", filetypes=ALLOWED_FILES)
        print(f"Opening file: {filename}")
        # Open the image file
        img = Image.open(filename)
        # (optional) resize the image
        img = img.resize((300, 300))
        # Convert the image into tk compatible form, and save to self. Otherwise it will be
        # cleared from memory when this function ends and disappear from view.
        self.tking = ImageTk.PhotoImage(img)
        # Display the image in the label
        self.image_label.configure(image=self.tking)

if __name__ == "__main__":
    root = tk.Tk()
    app = AppWindow(root)
    root.mainloop()
```

Opening a second window

We have now reached the point of creating more than one window at once. With respect to organising your code, it is generally good practice to make each window its own class, so here we have the `AppWindow` for our main window, and `LoginWindow` for the second window that will open up.

Notice that the start of the `LoginWindow.__init__()` code is slightly different to what we have used so far. This is because in order to create a second windows, instead of just saving the parent to window and using that, we need to create a `tk.Toplevel()` object. Use what I have provided as a template from which to make your secondary windows and you should be fine.




```

import tkinter as tk
from tkinter import ttk
import time

# Pre-define some defaults
FONT_LARGE = ("Arial", 48)

class LoginWindow():
    def __init__(self, parent):
        # Create a variable with which we can reference our parent
        self.parent = parent
        # Secondary windows are made using tk.Toplevel() instead of using parent
        self.window = tk.Toplevel()
        self.window.geometry("400x300")
        self.window.title("Login")
        # Labels
        self.username_label = tk.Label(self.window, text="Username:")
        self.username_label.place(x=20,y=20)
        self.password_label = tk.Label(self.window, text="Password:")
        self.password_label.place(x=20,y=70)
        # Entry boxes
        self.username_text = tk.Entry(self.window)
        self.username_text.place(x=100,y=20)
        self.username_text.focus()
        self.password_text = tk.Entry(self.window, show="*")
        self.password_text.place(x=100,y=70)
        # Button
        self.login_button = tk.Button(self.window, text="Login", command=self.login)
        self.login_button.place(x=100,y=120)

    def login(self):
        self.userid = self.username_text.get()
        self.passwd = self.password_text.get()
        print(f"Your username is {self.userid} and password is {self.passwd}")
        # Close the login window
        self.window.destroy()

    def get_info(self):
        return self.userid, self.passwd

class AppWindow():
    def __init__(self, parent):
        # Create the window (linked to the app parent)
        self.window = parent
        self.window.geometry("400x200")
        self.window.title("Test app")
        # Create a text label and place it in the window
        self.hello_label = tk.Label(self.window, text="Hello world!", font=FONT_LARGE)
        self.hello_label.place(x=20, y=20)
        # Create a button
        self.login_button = tk.Button(self.window, text="Login", command=self.login_clicked)
        self.login_button.place(x=20, y=170)

    def login_clicked(self):
        # Create login window
        login_window = LoginWindow()
        # Wait until the login window is closed
        self.window.wait_window(login_window.window)
        print("Finished waiting")
        uid, pwd = login_window.get_info()
        self.hello_label.configure(text=f"Hello {uid}")

if __name__ == "__main__":
    root = tk.Tk()
    app = AppWindow(root)
    root.mainloop()

```

Tabs

The last item to mention in Tkinter is the use of Tabs. As far as Tkinter is concerned, the content area within a tab can almost be treated as it's own semi independent window. Any objects that we wish to place within a tab, the location coordinates are relevant to that tab.



```

import tkinter as tk
from tkinter import ttk
from tkinter import messagebox

class AppWindow():
    def __init__(self, parent):
        # Create the window
        self.window = parent
        self.window.geometry("400x400")
        self.window.title("Test app")
        # Create a text label and place it in the window
        self.hello_label = tk.Label(self.window, text="Hello world!", font=FONT_LARGE)
        self.hello_label.place(x=20, y=20)
        # Create 3 tabs
        self.tab_container = tk.Frame(self.window)
        self.tab_container.place(x=0,y=0,width=400,height=400)
        self.tabs = ttk.Notebook(self.tab_container)
        self.tab_1 = tk.Frame(self.tabs)
        self.tab_2 = tk.Frame(self.tabs)
        self.tab_3 = tk.Frame(self.tabs)
        self.tabs.add(self.tab_1, text="Tab 1")
        self.tabs.add(self.tab_2, text="Tab 2")
        self.tabs.add(self.tab_3, text="Tab 3")
        self.tabs.place(x=0,y=0,height=400,width=400)
        # Define what function to run when current tab is changed
        self.tabs.bind("<NotebookTabChanged>", self.on_tab_selected)
        # Content for tab 1
        self.label1 = tk.Label(self.tab_1, text="I am the content of tab 1")
        self.label1.place(x=20, y=20) # Coordinates are relative to within the tab area
        # Content for tab 2
        self.label2 = tk.Label(self.tab_2, text="I am the content of tab 2")
        self.label2.place(x=20, y=20) # Coordinates are relative to within the tab area
        # Content for tab 3
        self.label3 = tk.Label(self.tab_3, text="I am the content of tab 3")
        self.label3.place(x=20, y=20) # Coordinates are relative to within the tab area
        self.close_button = tk.Button(self.tab_3, text="Close", command=self.close_clicked)
        self.close_button.place(x=20,y=70)

    def on_tab_selected(self, e):
        # Function to execute whenever current tab is changed
        selected_tab = e.widget.select()
        tab_text = e.widget.tab(selected_tab, "text")
        if tab_text == "Tab 1":
            print("You clicked into tab 1")
        if tab_text == "Tab 2":
            print("You clicked into tab 2")
        if tab_text == "Tab 3":
            print("You clicked into tab 3")

    def close_clicked(self):
        result = messagebox.askyesno("Confirm", message="Do you want to quit?")
        if result:
            self.parent.quit()

if __name__ == "__main__":
    root = tk.Tk()
    app = AppWindow(root)
    root.mainloop()

```

Other assorted Tkinter tips

A few little discoveries I've made along the way of building a few projects with Tkinter.

To execute a function (in this case, `self.window.quit()`) when the window close icon is clicked...

```
self.window.protocol("WM_DELETE_WINDOW", self.window.quit)
```

To make the window partially transparent...

```
self.window.attributes('-alpha', 0.8)
```

To force the window to stay visibly on top of all other programs...

```
root.wm_attributes("-topmost", 1)
```

Any other useful little tricks that should be added here? Let me know!

Further reading

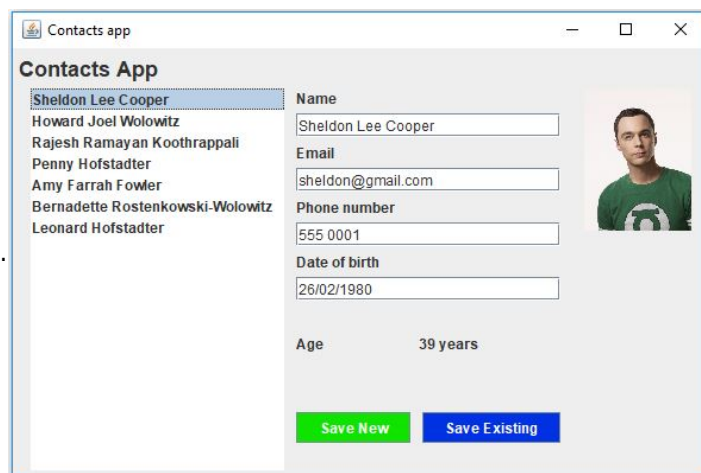
- https://python-textbok.readthedocs.io/en/1.0/Introduction_to_GUI_Programming.html
- <http://www.effbot.org/tkinterbook/grid.htm>
- <https://docs.python.org/3.7/library/tkinter.html>
- https://www.python-course.eu/python_tkinter.php

Problem

This is a simple contacts directory application, that stores each persons name, phone number, email address and date of birth (from which it auto calculates their age). It also checks a folder using `os.listdir()` (see the files & folders lesson) for the presence of a `jpg` file with a name matching the name of the selected person. If the photo exists, it is also displayed on the window.

This is a challenging application but it doesn't use any programming technics we have not covered.

To help get you started, here is some sample JSON data I created for my testing of the project. Obviously feel free to come up with your own.



```
[
  {
    "phoneNumber": "555 0001",
    "name": "Sheldon Lee Cooper",
    "dateOfBirth": "1980-02-26",
    "email": "sheldon@gmail.com"
  }, {
    "phoneNumber": "555 0002",
    "name": "Howard Joel Wolowitz",
    "dateOfBirth": "1981-03-01",
    "email": "howard@gmail.com"
  }, {
    "phoneNumber": "555 0003",
    "name": "Rajesh Ramayan Koothrappali",
    "dateOfBirth": "1981-10-06",
    "email": "raj@gmail.com"
  }, {
    "phoneNumber": "555 0004",
    "name": "Penny Hofstadter",
    "dateOfBirth": "1985-12-02",
    "email": "penny@gmail.com"
  }, {
    "phoneNumber": "555 0005",
    "name": "Amy Farrah Fowler",
    "dateOfBirth": "1979-12-17",
    "email": "amy@gmail.com"
  }, {
    "phoneNumber": "555 0002",
    "name": "Bernadette Rostenkowski-Wolowitz",
    "dateOfBirth": "1984-01-01",
    "email": "bernadette@gmail.com"
  }, {
    "phoneNumber": "555 0006",
    "name": "Leonard Hofstadter",
    "dateOfBirth": "1980-05-17",
    "email": "leonard@gmail.com"
  }
]
```


17. What next?

You are limited only by your imagination now.

May I kindly suggest a visit to pbaumgarten.com/python where I have a range of further tutorials to teach you how to use various libraries that do a range of cool things such as...

- Create your own video games (using Pygame)
- Image editing & manipulation (using PIL and OpenCV)
- Create your own PDF files (using fpdf)
- Read excel spreadsheets (using Pandas)
- Generate graphs & charts (using Matplotlib)
- Read/write to databases (using sqlite3)
- Build a webserver (using Flask)
- Robotics with Raspberry Pi (using easyaspi)
- Robotics with Arduino (using pyfirmata)
- Robotics with Cozmo robots (using pycozmo)
- Face detection (using ImageToolsMadeEasy and OpenCV)
- Introduction to machine learning/artificial intelligence systems

Wishing you all the very best with your programming journey!

Thank you

Paul Baumgarten (2020)

Appendix: Installing Python libraries

For a couple of projects within this book, you need to install libraries that are commonly used with Python but not included as part of the automatic install process.

PIP is the Python package manager that should be installed with Python. We will use it to install the libraries we need. What are these libraries?

- **requests** is a HTTP connection library used for allowing Python programmatically access files and resources from a URL.
- **Pillow** is the Python Imaging Library, it is used to loading and manipulating image files such as JPEG and PNG.
- **Flask** is a simple Python web server library.

On Windows

On Windows, using your keyboard press the **Windows** and **X** keys together, choose **Command Prompt (Admin)**. Click **Yes** when asked if you want to allow the app to make changes to your PC.

```
pip install requests Pillow Flask
```

- If you don't have administrative access to your computer (perhaps it is a school computer), you may need to get your local technical support person to assist. Show them this page and they should know what to do.

On Mac and Linux

On Mac, go to your **Applications** folder, goto **Utilities**, and open the **Terminal** application. Type the following commands...

```
sudo pip3 install requests Pillow Flask  
exit
```