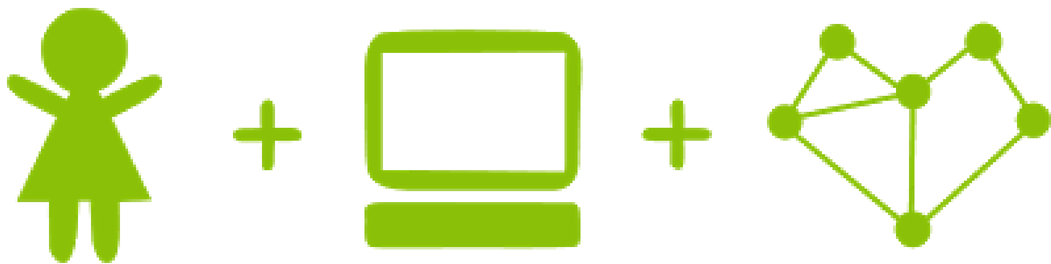


Python Reference Guide

Girls' Programming Network

Table of Contents

Table of Contents **1**

Idle **3**

Installing Idle 3

Opening Idle 5

Making a file 5

Variables **7**

Strings 8

Printing **10**

Using Python as a Calculator **12**

Input **13**

int() 13

If Statements **15**

Lists **19**

Loops **21**

For loops 21

While loops 22

Dictionaries **24**

Using Files **26**

Other python functions **28**

Random() 28

Range() 29

Understanding Errors **30**

Name Error 30

Type Error 30

Value Error 31

Index Error 31

Syntax Error 31

Key Error 32

Idle

Installing Idle

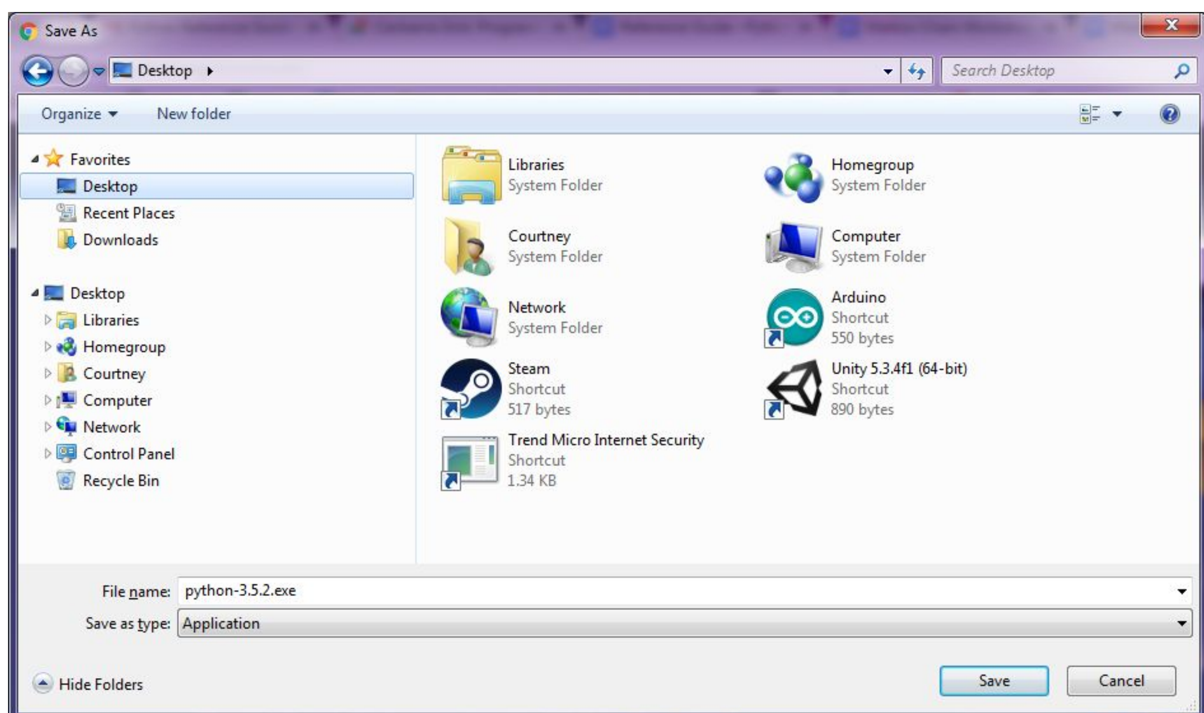
The Girls' Programming Network uses Idle to program in Python. The instructions in this guide are for Windows. You will need administrator permissions.

You can find the download link for Python on here: https://www.python.org/downloads/

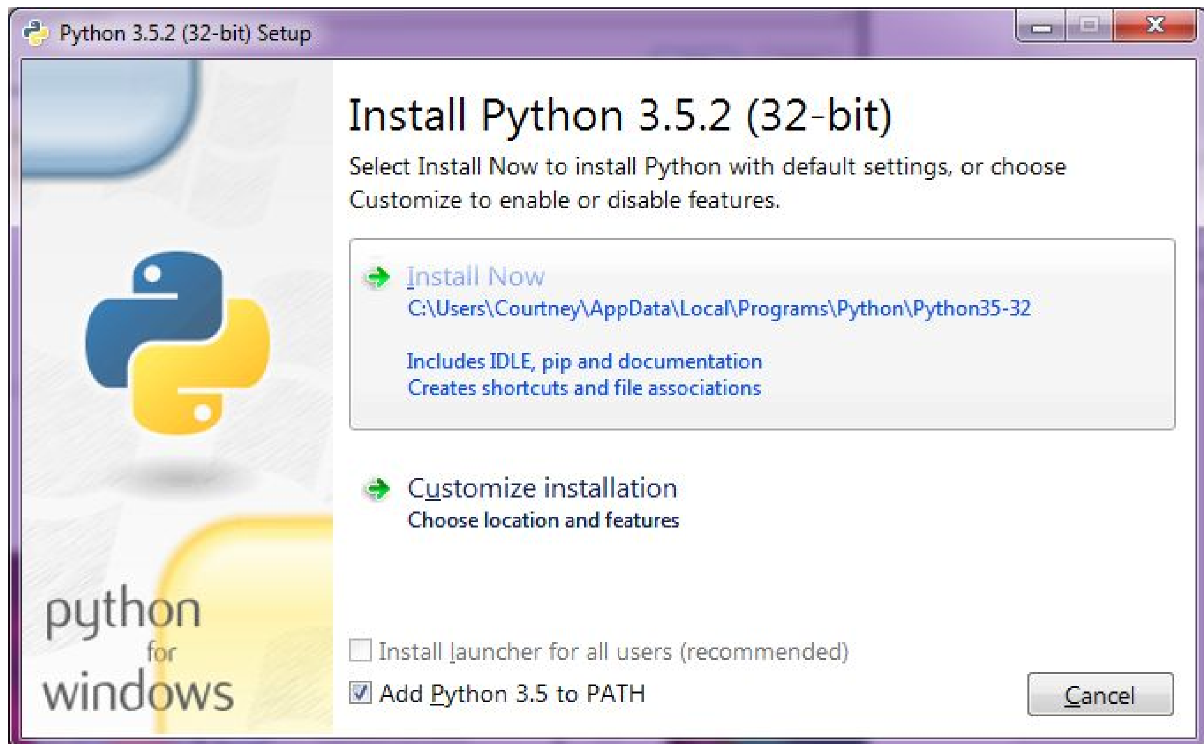
Click on the download button located near the top of the page.

Remember, we use Python 3 throughout this workbook, so you will need to download Python 3.X.X.

This will download Python-3.X.X.exe. Save it somewhere handy on your computer - like the desktop!



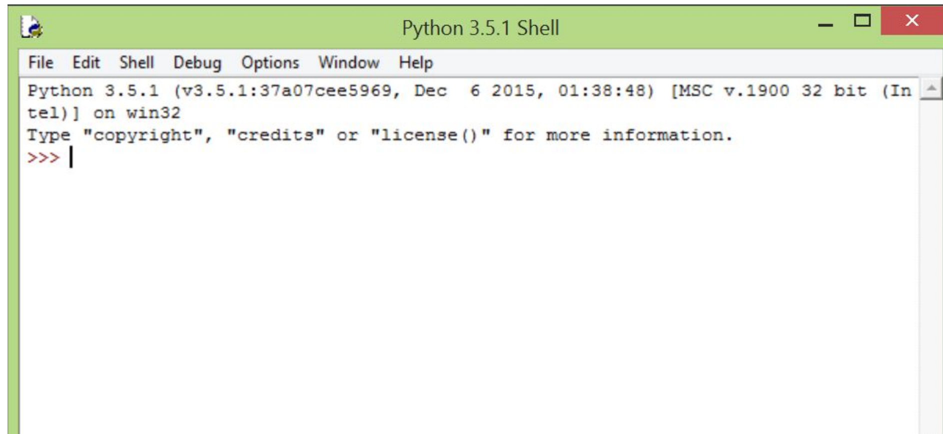
You will then need to double click on the downloaded Python file and follow the instructions. This will install Python. Make sure you select Python 3.5 to PATH.



Once it's finished installing, you can go to the start menu and click on Idle to run it.

Opening Idle

In the Start menu, search for the program IDLE and click on it. The program should look something like this:

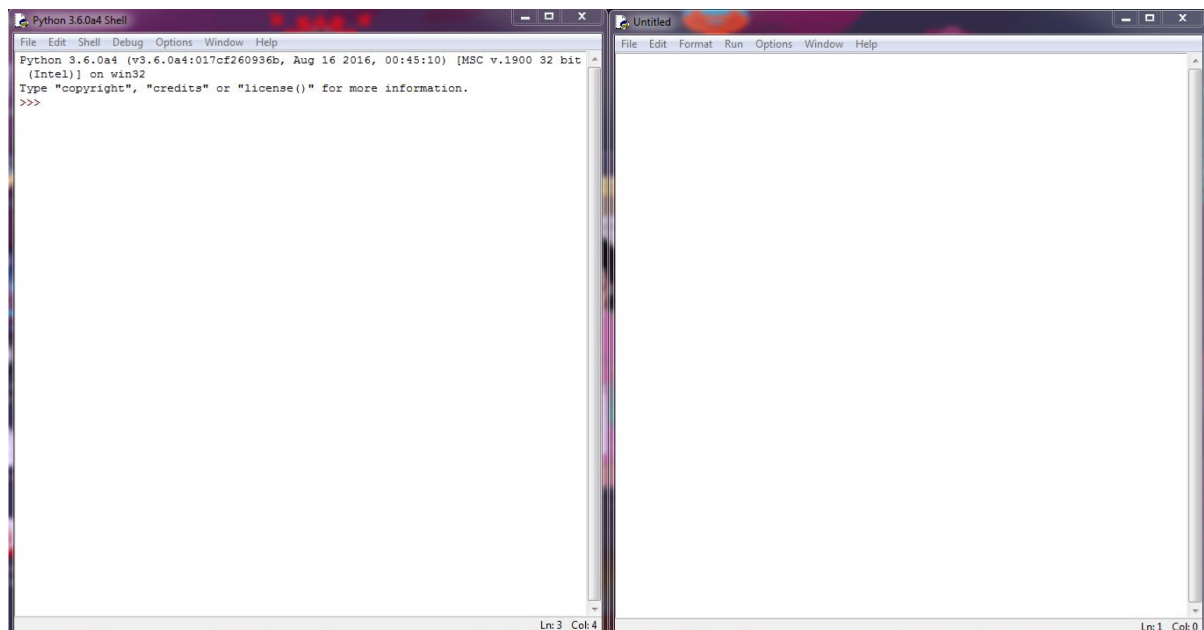


This window is called the ***shell***. Any commands you type here will run immediately. This is convenient for playing around with commands, or using Python as a calculator.

Making a file

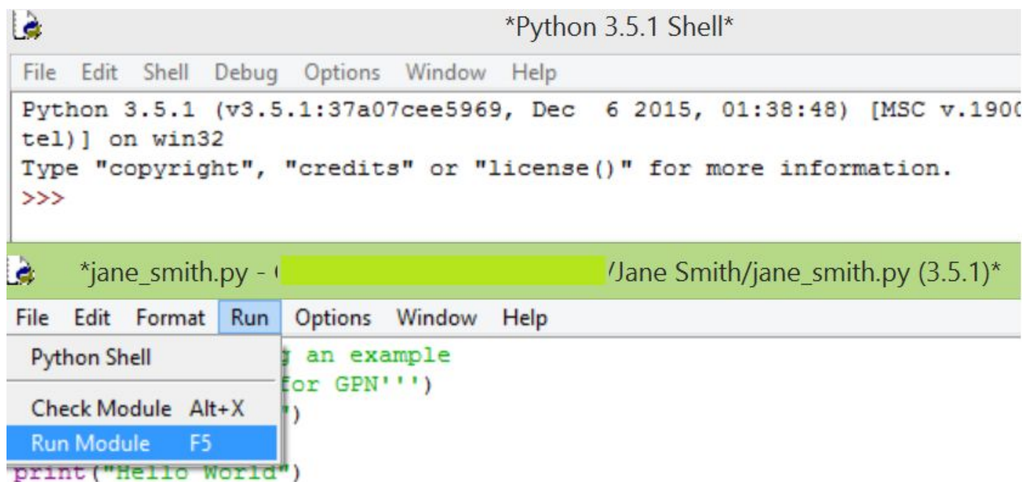
If you want to work on a lot of code and don't want to rewrite it every time you run it, it's much better to save it to a file.

To create a Python file, click on 'File' in the top left, and then 'New File'. You should now have a second window which is blank.

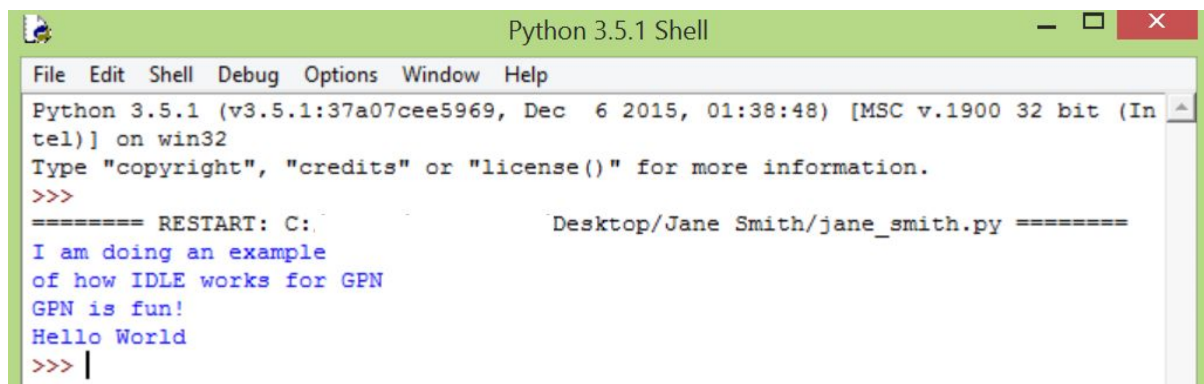


In the second window, click '***File***', and then '***Save As***'. Save the file somewhere you'll remember and name it something handy like **"**python_examples.py". Remember to add .py to the end so the computer knows it is a Python file.

After you have typed your code in this file you can run the file by selecting 'Run Module' from the 'Run' menu.



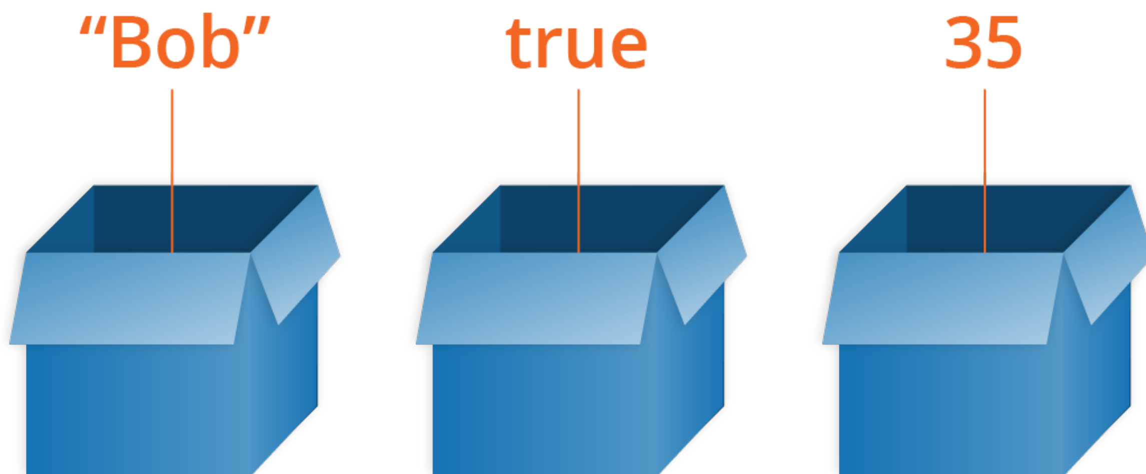
The code will then run, and any output or error messages will be printed in the shell.



Variables

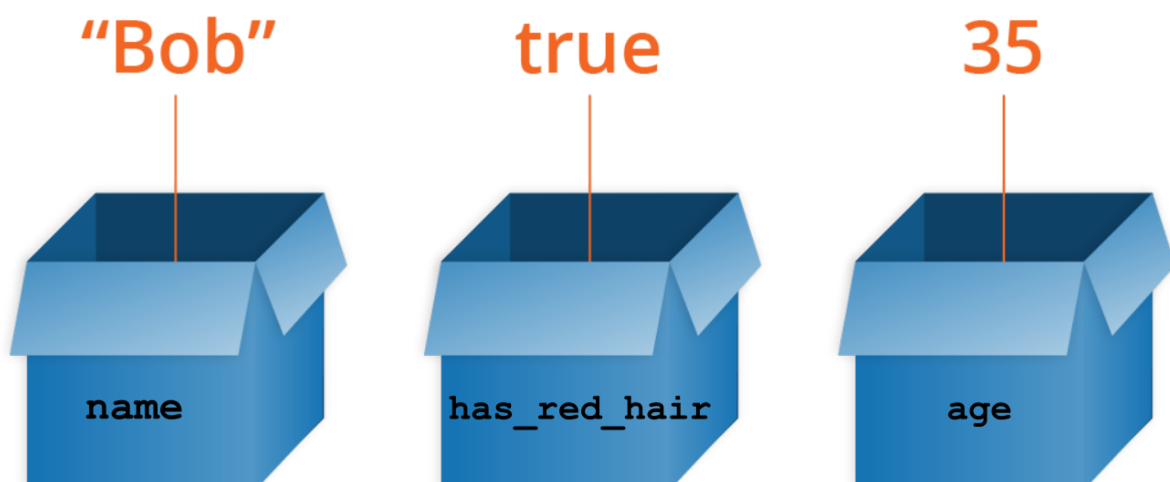
In programming, we like to store information that can be reused later on. We store this information in ***variables***.

A variable is like a box that we can store values in. For example:



We’ve assigned the boxes above different values. We’ve stored the values `"Bob"`, `true` and `35`!

It’s a good idea to name your variable something obvious. That way, when you use your variable later on in your code (or when someone else reads your code) it’s easy to tell what information the variable stores. Every variable must have a unique name so Python knows exactly what you want.



Now every time we go to use one of these variables, it’s easy to tell what information it stores!

Here is how you would create these variables in Python:

```
1  name = "Bob"
2  has_red_hair = True
3  age = 35
```

In this example, `name`, `has_red_hair` and `age` are the variable names.

Notice that text is always surrounded by quotation marks, but numbers aren't. There's also a special type of variable here, which can only store a value of `True` or `False`.

There are many different types of variables. Some examples are:

- a variable that contains a whole number, like 5, is called an `integer`, or `int` for short;
- a variable that contains a number with a decimal point, like 2.5, is called a `double`;
- a variable that contains text is called a `string`; and
- a variable that contains true or false is called a `boolean`.

Python almost always automatically knows what type of variable you are using so we don't need to specify the type.

Try this at home!

- How do you create a string with no letters in it?
- What happens when you create a variable, assign some text to it, and then assign a number to it?
- How would we add two number variables together?
- How would we add one string variable to the end of another string variable? (hint: it's like adding two numbers together!)
- What happens if we try and add a string variable to a number variable?

Strings

Strings are a variable type that you can use to save letters, words and sentences. Python will interpret everything you write in quotation marks as a string. Here are some examples of variables that contain strings:

```
1  traffic_light = "red"
2  some_other_words = "The traffic light"
3  action = "stop"
```

The cool thing about strings is that you can combine them to form a sentence by adding multiple words. ***You are not limited to the variables we created before***, and you can add new words by using the quotations marks.

```
4  sentence1 = some_other_words + " is " + traffic_light + "."
5  sentence2 = "We should " + action + "!"
6  both_sentences = sentence1 + " " + sentence2
7  print(both_sentences)
```

This will print:

```
The traffic light is red.
We should stop!
```

As you can see from this example, you need to add all missing parts to your sentence so that it makes sense. In this case we inserted spaces around the `is` word with the string `" is "` and a full stop at the end with the string `"."`.

Try this at home!

- What happens if you just add all three variables?
- What happens if try to add a number to a string?

Printing

When programming, it's really handy to be able to pass information back to to the user. We do this by printing to the screen using one of Python's handy ***built-in functions***, `print()`.

For example, we can do the following:

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

The above code prints a sentence:

```
Hello World!
```

You can put anything between the quote marks! Let's try another example.

```
2   print("This book is a python reference guide!")
```

We can also print variables to the screen using the print function.

```
1   some_number = 5
2   some_text = "a bit of text"
3   some_boolean = True
4   print(some_number)
5   print(some_text)
6   print(some_boolean)
```

The above code will print out the following to the screen:

```
5
a bit of text
True
```

Notice how we didn't use quote marks in the print statement? That's because Python already knows about the variables and can use them.

Python's print function is clever enough to accept text in quotation marks and a variable at the same time.

```
7   print("Whats is stored in the variable some_number? ", some_number)
```

Or with even more text:

```
8   print("Whats is stored in the variable some_number? ", some_number, "!
    I like that number!")
```

The first line would print

```
Whats is stored in the variable some_number? 5
```

The second line would print:

```
Whats is stored in the variable some_number? 5!  I like that number!
```

Those lines are getting longer and longer. It's getting harder to read on the screen! It might be nicer to split the text across multiple lines:

```
1   print("This is a really long line,")
2   print("but we can split it across multiple lines.")
```

This will print out:

```
This is a really long line,
but we can split it across multiple lines
```

Here is another example that will print the same as the code above:

```
3   print("This is a really long line, \nbut we can split it across
    multiple lines.")
```

This uses a special character, `\n`, to achieve the linebreak. Each letter we typed in quotes is printed except for the special character `\n`. The `\n` is one of many special characters you can use to format your print statements.

It's up to you which way you prefer.

Try this at home!

- What happens when you put the sentence "Welcome to the Girls' Programming Network!" in a print statement?
- What happens when you forget the quote marks?
- What happens when you forget the brackets?

Using Python as a Calculator

So now that we know how to store integers and doubles in variables, we can get Python to manipulate them. Python uses ***operators*** to perform mathematical equations on integers and doubles.

Operator	What it does	Example
+	Adds two numbers together	2 + 2 = 4
-	Subtracts the right number from the left number	3 - 2 = 1
*	Multiplies two numbers together	3 * 2 = 6
/	Divides the left number by the right number	6 / 2 = 3
%	Divides the left number by the right number and gives the remainder	5 % 2 = 1
**	Performs exponential (power) calculations on number (ie 5 to the power of 2)	5 ** 2 = 25
//	Performs division and returns a whole number with no remainder	5 // 2 = 2

An example is:

```
1   addition = 6 + 6
2   subtraction = 8 - 5
3   division = 8 / 4
4   print("6 + 6 = ", addition)
5   print("8 - 5 = ", subtraction)
6   print("8 / 4 = ", division)
```

This will print:

```
6 + 6 = 12
8 - 5 = 3
8 / 4 = 2
```

While the examples above use numbers, you can use variables instead! Why not give it a try?

Input

It's a little boring running the same program over and over again using the values in variables that we created. We can make our program more interesting by getting input from the user.

For example, if we wanted to ask the user for their name, we could use the following code:

```
1    user_name = input("What is your name? ")
```

What we did here was ask the user for their name using the built-in function `input()`. We then stored what the user told us in a variable called `user_name`.

We can then confirm that we stored the user's information correctly with a print statement:

```
2    print("Your name is", user_name)
```

Try this at home!

- What happens when we ask for a number from the user, store it in a variable and then try to add another number to it?
- What happens if you try a print statement with the variable listed before the text?

int()

An ***Integer*** or ***int*** is a number. When we get a number from the user using the `input()` function, Python automatically stores it as a string. We can see its value when we run this code:

```
1    user_age = input("What is your age? ")
2    print("Your age is", user_age)
```

But what happens when we try to run the following code?

```
1    user_age = input("What is your age? ")
2    next_year = user_age + 1
3    print("Next birthday, you will turn", next_birthday)
```

We get an error! Python doesn't know that you actually received an integer as input, so gets confused when you try and add an integer to a string.

So what we need to do is explicitly tell Python the input we got is actually a number, or more specifically, an integer. We can convert strings to integers using the `int()` function.

```python
user_age = input("What is your age? ")
user_age = int(user_age)
next_year = user_age + 1
print("Next birthday, you will turn", next_year)
```

Now python is happily adding 1 to `user_age`!

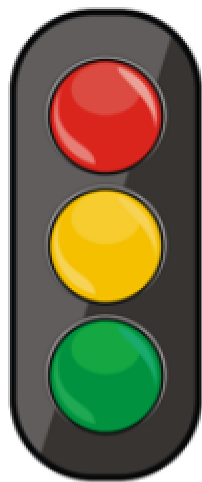
Try this at home!

- What happens if you try converting the value `"hello"` to an integer?
- What happens if you try converting the value 3.1415 to an integer?

If Statements

One of the most important parts of computer programs is making decisions about what to do next. These decisions might depend on what a user does, what is in a file or database, or something else. To do this, we need ***if statements***. An if statement checks if a ***condition*** is true: if so, it does something; if not, it does something else.

We do this all the time in real life. For example:

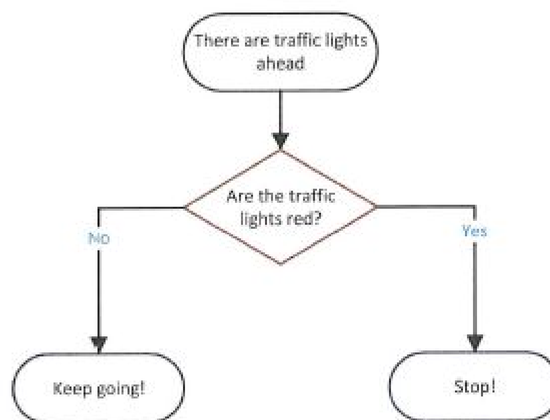


If light is red, then stop

Otherwise, if light is yellow, then slow down

Otherwise, if light is green, then go

We can represent what happens at the traffic lights in a flow chart:



We could represent this in Python as:

```
1  traffic_light = "red"
2  if traffic_light == "red":
3      print("Stop at the lights!")
4  else:
5      print("You can go!")
```

There are a few things going on here, so let's break it down. First, we made a variable called `traffic_light` and assigned it the value `"red"` in line 1. That's nothing new.

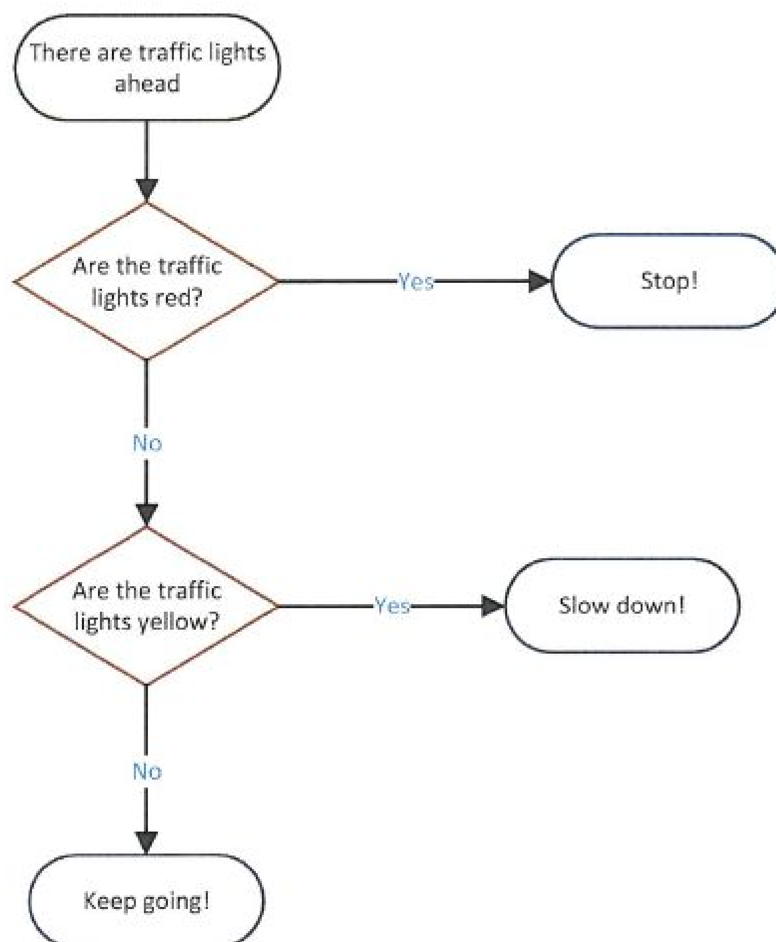
Then, we start the if statement on line 2. If statements describe conditions that are either met or not met. In line 2 the condition we are checking is if the variable `traffic_light` is equal to red.

The `==` is the equality operator. Don't confuse this with `=`, which is the assignment operator! The equality operator compares if two things are equal.

If the condition on Line 2 is met, (which it has been in this example), then we run the code that is on line 3. If the condition has not been met, that is they are not equal, then the block of code starting on line 5 is run.

In Python, the if condition always ends with the colon symbol. The indentation on lines 3 and 5 is important as it shows where the next block of code starts, if the condition on the line before has been met.

But our example above is a little more complex! What about the yellow light?

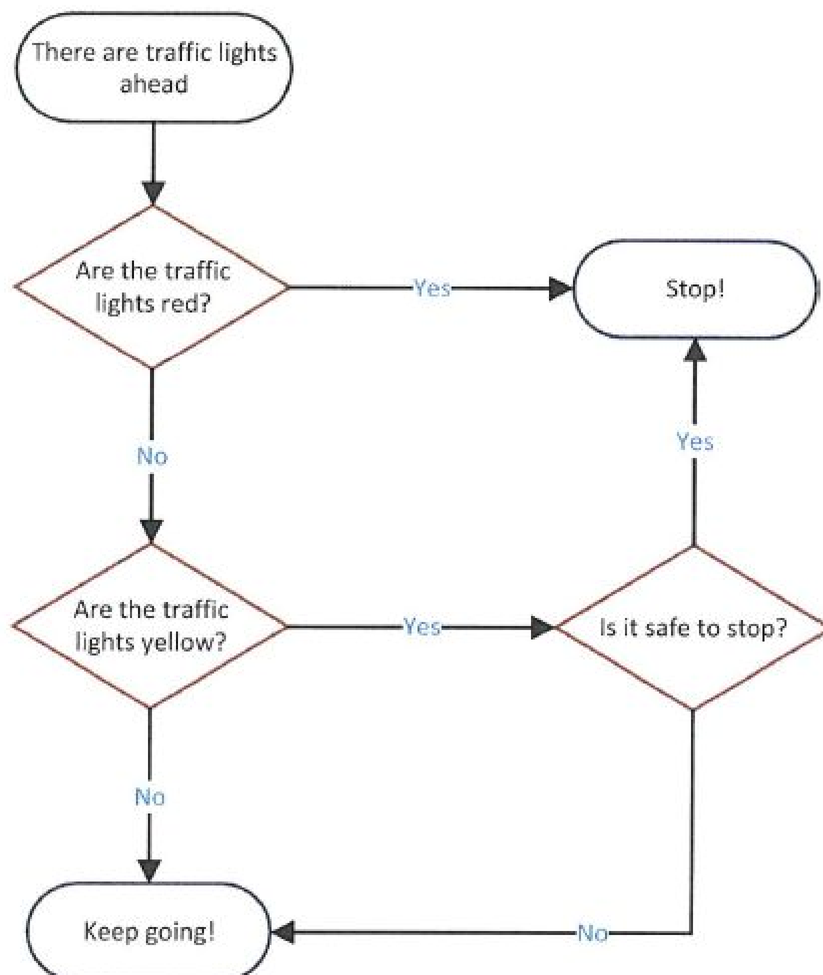


This additional condition means we need to use an `elif`:

```
1   traffic_light = "red"
2   if traffic_light = "red":
3       print("Stop at the light!")
4   elif traffic_light = "yellow":
5       print("Slow down!")
6   else:
7       print("You can go!")
```

In line 4, we add an `elif` block. This is shorthand for “else if” and basically means, if the first condition wasn’t met, then check if this other condition is met. You can have as many `elif` statements or conditions as you want.

Sometimes there are multiple conditions to be met to satisfy the if statement.



```
1   traffic_light = "yellow"
2   safe_to_stop = False
3   if traffic_light == "red":
4       print("Stop at the light!")
5   elif traffic_light == "yellow" and safe_to_stop == False:
6       print("Go through the light!")
7   elif traffic_light == "yellow" and safe_to_stop == True:
8       print("Slow down!")
9   else:
10      print("You can go!")
```

In line 3, we are combining two comparisons using the `and` operator. If both the conditions are met, the if statement has been satisfied and the next line of code (that is indented) is run.

Which line do you think should print in the last example? Try running it and see if you're right!

We can also combine the comparisons using the `or` operator. When there is an `or` operator, if either one of the conditions are met, then the if statement is satisfied and the indented code is run.

Try this at home!

- Can you rewrite the traffic light example to use two if statements?
- Can you write an if statement that uses the or operator?
- What happens if you use two if statements in a row?
- What happens if you change the value of `traffic_light`

Lists

So far we have dealt with variables that are either numbers, or text. Another type of variable is a ***list***. It's what it sounds like: a list of things. These things can be numbers, text or even lists! Yes, you can have a list of lists. You create a list with square brackets and separate each item with a comma, like this:

```
1   dinner_options = ["spaghetti", "tacos", "stir-fry"]
```

There are a lot of functions that you can use to change or access a list's items. To access a certain item from the list, we can use:

```
2   selected_dinner = dinner_options[1]
3   print(selected_dinner)
```

This will print the following:

```
tacos
```

What? It printed the second item in the list? That's because *counting in Python always begins with zero*. This is really important to remember!

But what if we need to add another item to the list we created? We can do this using the `append()` function. `append()` always adds the item to the end of the list.

```
4   dinner_options.append("pizza")
5   print(dinner_options)
```

This will print

```
["spaghetti", "tacos", "stir-fry", "pizza"]
```

We've just been told that tacos is no longer an option for dinner! We can remove tacos from our list using the `remove()` function.

```
6   dinner_options.remove("tacos")
7   print(dinner_options)
```

Now our list looks like this:

```
["spaghetti", "stir-fry", "pizza"]
```

We can even convert a string into a list! We do this using the `split()` function. The `split()` function uses the spaces in sentences to split the string into a list.

```
1   some_text = "I know all about lists"
2   some_list = some_text.split()
3   print(some_list)
```

Our newly-created list looks like this:

```
["I", "know", "all", "about", "lists"]
```

The split() function can be really handy! We can also go in reverse though, and use our list to create a sentence using the join() function.

```
4   space = " "
5   new_text = space.join(some_list)
6   print(new_text)
```

Using the join() is a little more confusing! In the above code, we're creating a new variable, space, that just stores a space. In the second line of code we're creating a string where all the words have a space between them and assigning the string to new_text. We then print our sentence, which looks like this:

```
I know all about lists
```

Try this at home!

- How would you access the fifth item in a list?
- How would you access the first item in a list?
- How would you add an item to the middle of the list?
- How would you remove it the word you just added?
- Using an if statement, can you determine if a word is in a list?
- How can you find out how many items are in a list?
- How can you turn a list into a sentence with the '-' character between words?

Loops

Sometimes we want to do the same thing many times. We could just copy and paste the code lots of times, but that gets pretty boring. Instead, we can use a ***loop*** to do this for us. There are two types of loops in Python: `for` loops and `while` loops. A `for` loop will do something a certain number of times, whereas a `while` loop will do something repeatedly while some condition is met.

For loops

Let's have a look at an example of a `for` loop.

```
1   number_list = [1,2,3,4,5]
2   for i in number_list:
3       print("We're up to number", i)
```

The first line sets how many times the loop will run. The code indented after this line is what will be run repeatedly. In this example, the loop will run 5 times and the variable `i` will give the number of iterations that we're up to. So, what will it print?

```
We're up to number 1
We're up to number 2
We're up to number 3
We're up to number 4
We're up to number 5
```

You can also run `for` loops over lists of words. It works much the same, but instead of the loop variable being a number, it is now an item in the list.

```
1   dinner_options = ["spaghetti", "tacos", "stir-fry"]
2   for option in dinner_options:
3       print(option)
```

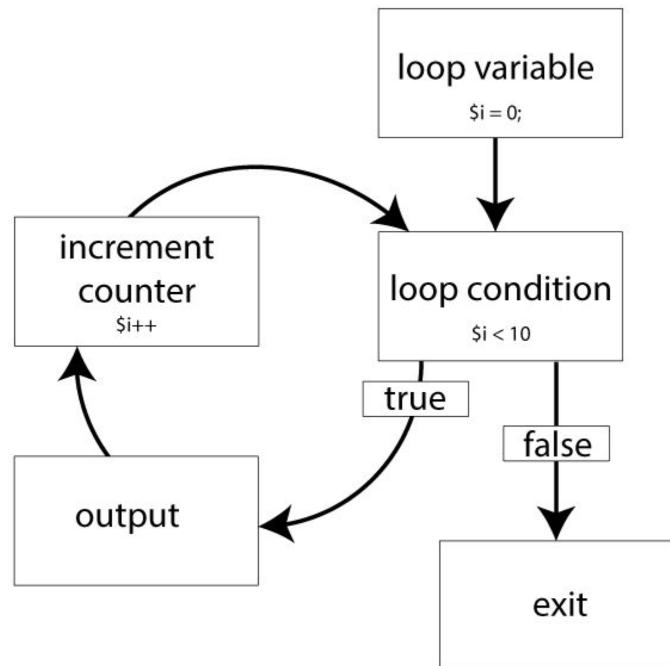
We can do the same thing over strings. Python will automatically split the string into a list:

```
4   dinner = "spaghetti"
5   for letter in dinner:
6       print(letter)
```

What do you think those two examples will print?

While loops

Let's look at an example of a `while` loop. While loops continue until the starting condition is no longer true:



This time, instead of looping a certain number of times, our loop keeps going and going and going until its condition becomes false.

An example of a while loop in Python is:

```
1   keep_going = "yes"
2   while keep_going == "yes":
3       keep_going = input("Keep going? ")
```

This example creates a variable `keep_going` and initially sets it to `"yes"`. Then, the loop checks whether the variable is equal to `"yes"`, which of course it is because we just set it to that, but then, inside the loop it asks the user if it should keep going. If the user says yes, then the loop condition is true and it loops again. If the user says no, then the condition becomes false and the loop ends.

In while loops, we don't need to give python a variable to keep checking. We can just set the while loop to be true:

```
1   while True:
2       print("Still going!")
```

This while loop will never stop running. This is because true will always evaluate to be true! This is known as an infinite while loop. Now we have to force our program to stop! Press “CTRL” and “C” at the same time. This will show an exception but the program will be stopped.

We can stop infinite while loops using break. Break exits the loop entirely:

```
1   traffic_light = "green"
2   while True:
3       if traffic_light == "red":
4           break
5       traffic_light = "red"
6       print("traffic light is now", traffic_light)
```

```
traffic light is now red
```

When the loop was passed once, the `traffic_light` variable will be set to red. The second time around the code will exit the loop due to the if statement and the `break`.

Try this at home!

- Can you put a for loop inside of a while loop? What happens?
- If you put a for loop inside of a for loop, how many times does it run? (hint: use a counter variable!)
- What happens when you put a for loop inside a while loop, and use a break statement within the for loop?

Dictionaries

Dictionaries are really useful when you want to store a lot of information about a certain topic. Dictionaries work by using key-value pairs. You can use the key in the dictionary to find the value - this is called mapping!

We can use this to store a lot of phone numbers:

```
1    phone_numbers = {}
2    phone_numbers = {"Alice":62662675, "Bob":62665812, "Eve":62669224}
```

Let's break down the code above. We first created a variable, `phone_numbers`. We then created the dictionary with `{ }` brackets. This is really important! It's how python knows it's a dictionary.

In the second line, we stored key-value pairs in the dictionary. Our first key is Alice, and the value is Alice's phone number. We then have another key, Bob, and we've stored Bob's phone number.

We can access each of these key-value pairs individually by using the key:

```
3   alice_number = phone_numbers["Alice"]
4   print(alice_number)
```

This will print:

```
62662675
```

Adding to the dictionary is easy. We just need to give python they key and value:

```
5   phone_numbers["Trudy"] = 62669158
6   print(phone_numbers)
```

Now our dictionary looks like this:

```
{'Eve': 62669224, 'Bob': 62665812, 'Trudy': 62669158, 'Alice': 62662675}
```

We can also remove values from dictionaries:

```
7   del phone_numbers["Eve"]
8   print(phone_numbers)
```

`del` is a keyword that is short for the word delete. In the code above, we deleted the key `"Eve"`. Because we deleted the key, the value was also removed and our dictionary now looks like this:

```
{'Bob': 62665812, 'Alice': 62662675, 'Trudy': 62669158}
```

Dictionaries can store more than one kind of information. Let's create a dictionary about a person:

```
1    person_dictionary = {}
2    person_dictionary = {"name": "Sally", "age": 13, "School": "Lyneham
     High School"}
```

So long as there is a key and a value, we can store whatever we like in dictionaries, including lists!

Try this at home!

- Can you create a dictionary where the values are lists?
- In the dictionary of lists you created, can you remove one item from the list?
- Can you loop over the dictionary and print each key value pair out onto a separate line?

Using Files

Sometimes we need to get a lot of input for our program. Rather than typing it all out every time we run our program, it would be much better to save the input to a file and read it instead.

To start, create a text file using a text editor such as notepad and save it on your desktop as `myFile.txt`. Put three lines of text in there, such as:

```
We're learning how to read from files.
So we need some text!
I love learning!
```

Now, we need to tell our program to open the file so it can be read. We use the `open()` function for this.

```
f = open("C:\Desktop\myFile.txt", "r")
```

In this code, we created a variable called `f` which we stored a file object in. This file object was created by the `open()` function. We gave the open function two arguments. The first argument is the `filename`, which is the absolute path to the text file we created. The second argument is what mode to open the file. We gave it the argument `"r"`, which means we opened it in read mode.

We then can read all the data in the file:

```
f.read()
```

This reads in the entire file in one go and stores it in memory. Using a for loop, we can then go over each of the lines:

```
for line in f:
    print(line)
```

When we've finished reading the file, we need to close it. We can do this with `close()` function:

```
f.close()
```

So our entire code looks like this:

```
1   f = open("C:\Desktop\myFile.txt", "r")
2   for line in f:
3       print(line)
4   f.close()
```

Now that we've read from a file, we need to write to it! We can open a file for writing in two different modes - write or append. Using write mode completely overwrites the file, while append mode will add additional text to the end of the file.

```
f = open("C:\Desktop\myFile.txt", "w")
```

Notice how in the code above we used `"w"` instead of `"r"`? This opened the file in write mode. If we want to use append mode, we can use `"a"`.

We then need to write to the file:

```
f.write("Files are awesome!")
```

Once we've finished writing to the file, we need to close it again!

```
f.close()
```

So this time, our entire code looks like this:

```
1    f = open("C:\Desktop\myFile.txt", "w")
2    f.write("Files are awesome!")
3    f.close()
```

If you open the file located on your desktop, you should see that the file now contains the text:

```
Files are awesome!
```

If you want to open an existing file and add more text to it you need to open your file in mode "a" for append instead of "w" for write.

Try this at home!

- Can you give the open() function a relative path to a file?
- Can you use a for loop to write multiple lines to a file?
- What happens if you forget to close a file and then try to open it again?

Other python functions

Functions are hidden code that tell python to do something. We've already seen two really good examples of pythons functions: `print()` and `input()`. Python has a lot of these in built functions that we can use. This section is going to introduce you to two more of them.

A ***Module*** is a collection of functions. It has code that has been written by someone else and that has been packaged so that we can use it. When we want to use a module, we can import the library using the reserved term `import`. We can then use specific functions from that library.

Random()

We can import a library to let us select an item from a list at random. This is called the `random` library.

```
import random
```

We then need to create our list with dinner options.

```
dinner_options = ["spaghetti", "tacos", "stir-fry"]
```

Today we're going to randomly select what's for dinner. To choose a random item from a list, we use the random.choice() function, like this:

```
dinner = random.choice(dinner_options)
```

So, our full dinner selection example looks like this:

```
1   import random
2   dinner_options = ["spaghetti", "tacos", "stir-fry"]
3   dinner = random.choice(dinner_options)
4   print(dinner)
```

Try it in the shell and see what Python has chosen for your dinner! Try it a few times, and see if it picks something different each time.

Range()

The `range()` function is really handy for when you need to set up lists of numbers quickly.

```
1    my_list = list(range(5))
2    print(my_list)
```

This will print

```
[0, 1, 2, 3, 4]
```

Wait, what? Why did it start at 0? This is because the `range()` function always starts at 0.

We can combine the `range()` function with a for loop to decide how many times the for loop will run.

```
3    for i in range(5):
4        print("We're up to number", i)
```

This will print:

```
We're up to number 0
We're up to number 1
We're up to number 2
We're up to number 3
We're up to number 4
```

Isn't that nifty? Now it's really easy to set up a for loop when we know how many times we'd like it to run! We can also use variables for this:

```
1    run_times = 7
2    for i in range(run_times):
3        print("We're up to number ", i)
```

What is the last number that you think will print out?

Understanding Errors

Sometimes when we try to run a Python program, an error will appear. Errors are nothing to be afraid of and are just the Python Shell's way of helping us write code that will run. Let's look at an example:

```
File "<pyshell#1>", line 1, in <module>
```

The first part, `File "<pyshell#1>"`, tells us which file the error is in, or, as in this case, that it was in the Python interactive shell.

The second part, `line 1`, tells us which line in the file the error is on.

The third part, `<module>`, tells us which module the problem is in. We haven't talked about modules in this guide. You can ignore this part of the error for now.

Sometimes Idle will put a red underline exactly under where the problem is, to help you find it.

Name Error

```
>>> print(myname)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print(myage)
NameError: name 'myname' is not defined
```

Variables and strings can be tricky. Remember that strings always have "quotes" either side, and variables don't have quotes but have to be defined before they can be used. In this example, python thinks `myname` is a variable because there are no quotes, but we haven't told Python to store anything inside of it. To make this code snippet correct, we have to make sure `myname` is defined by storing a value inside it:

```
>>> myname = "Elly"
>>> print(myname)
Elly
```

Type Error

```
>>> len(5)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    len(5)
TypeError: object of type 'int' has no len()
```

In this example we are trying to get the length of the integer 5. But some functions only work with specific types of input and `len()` is one of these. Python doesn't know how to tell the length of an integer, so we see this error. However we can get the length of strings. If we really wanted to know how long a number is, we can wrap it in quotes to make it a string

first. Or if we want to know how many characters are in a string, we can do that with `len()`. Some examples:

```
>>> len("5")
1
```

```
>>> len("GPN is Awesome!")
15
```

Value Error

```
>>> int("hello")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    int("hello")
ValueError: invalid literal for int() with base 10: 'hello'
```

Python is telling us that `"hello"` is an invalid value here. The function `int()` expects strings that contain only numbers and no letters. Base 10 is our counting system with 10 numbers, 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. But the letters h, e, l, and o are not in that list of numbers and so are ‘invalid’ values, resulting in a value error.

This error is different from a Type error, where only a certain type is allowed. In this example both ints and strings can be given to the function `int()`, but the value of the string might be invalid.

An example of correct usage of `int()` is:

```
>>> int("125")
125
```

```
>>> int(99)
99
```

Index Error

```
>>> fruits = ["Banana", "Apple", "Pear"]
>>> print(fruits[3])
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    Fruits[3]
IndexError: list index out of range
```

This is a tricky problem. Python is telling us our list index, 3, is out of range, which means there isn’t an element in the list at index 3. Remember that list indexing in Python starts from 0. So for a `list` with 3 elements, the index of the first element will be 0, then 1, then 2 for the last element. So if we want to access the last element in this list, we can do it like this:

```
>>> print(fruits[2])
"Pear"
```

Syntax Error

```
>>> if myname = "Helen":
SyntaxError: invalid syntax
```

In this error the equals sign is highlighted red. A single equals sign is used to store values inside variables. But in this if statement, we want to compare two values. Comparison is done with two equals signs right next to each other, like this:

```
>>> if myname == "Helen":
```

```
>>> if light == "green":
print("You can go!")
SyntaxError: expected an indented block
```

Python is able to know what lines are part of an if statement (or a while or for statement) by the indentation of the lines under it. In the error above, the print statement is at the same indentation as the if statement that it belongs to, meaning Python doesn't recognise the print statement as belonging to the if. There is no indented statement after the if statement, but Python is expecting one. The correct syntax is:

```
>>> if light == "green":
    print("You can go!")
```

Key Error

```
>>> mydictionary = {"Name":"Helen", "Age":"15"}
>>> print(mydictionary["Name"])
Helen
>>> print(mydictionary["age"])
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    Mydictionary["age"]
KeyError: 'age'
```

In this example we have a dictionary with two keys, `"Name"` and `"Age"`, along with corresponding values. We try to access the key `"age"`, but in Python, strings are case-sensitive and `"age"` is not the same as `"Age"` so Python can't give us an answer. The correct way to access Helen's age is:

```
>>> print(mydictionary["Age"])
```