



The Raspberry Pi Foundation Notes

August 2019

Reviewed

These are the transcripts of the 4 day course on Python designed by the Raspberry Pi Foundation and the Open University. I have also added a commentary.

What you will need

To create Python programs you need a **text editor** to write your code and a **Python interpreter** which takes your code and runs it.

An editor, interpreter and other useful tools (such as a file browser) are often bundled together into an **Integrated Development Environment (IDE)**, which makes the process of creating programs much easier.

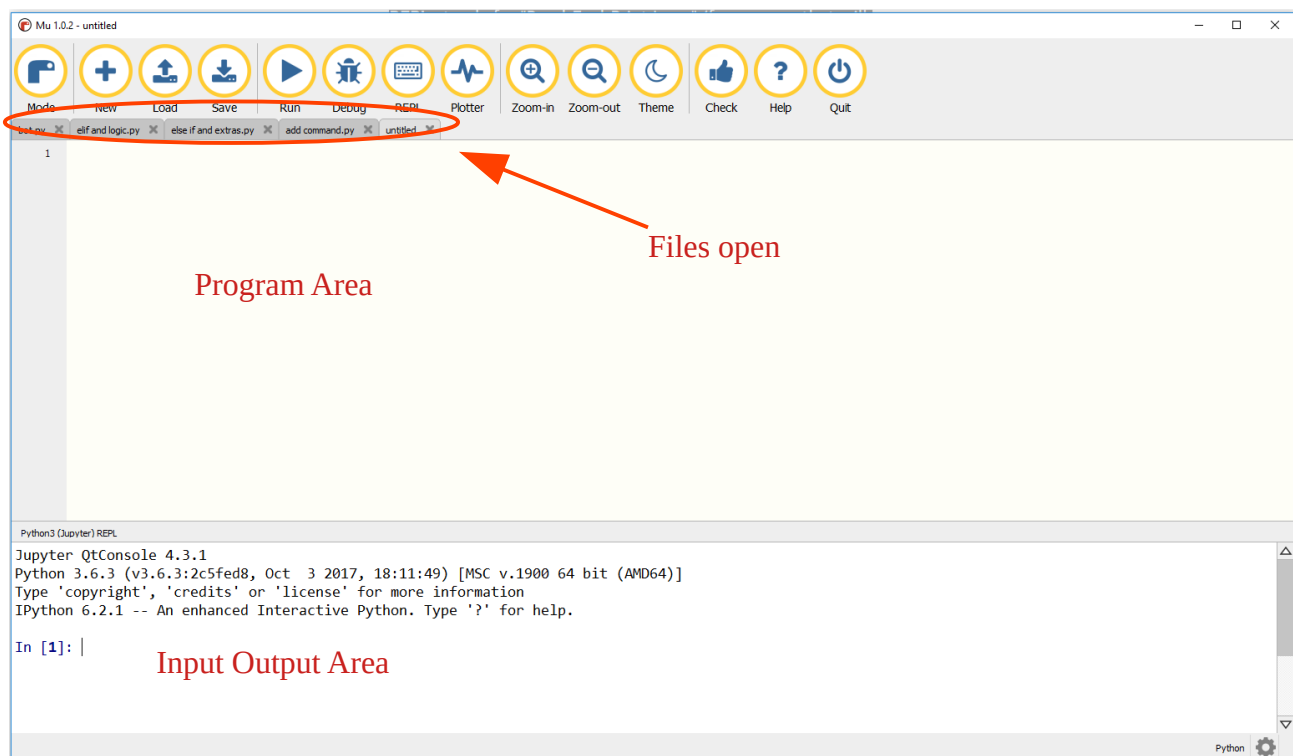
You will be using an IDE to create, run and test your Python programs. You can install an IDE on your computer, or you can use an internet browser to access an online IDE. An installed IDE has the benefit of being able to work when you are not connected to the internet. On the other hand, an online editor doesn't require anything to be installed.

Instructions are provided below for an installed IDE called [Mu](#) (and for the online IDE [Trinket](#).)

There are three key areas to the Mu and Trinket IDE's:

- **Files:** this is where all your programs will be stored. Each filename will end in `.py` meaning that the file is a Python file.
- **Editor:** this is where you will create your programs.
- **REPL** stands for "Read-Eval-Print Loop" (for reasons that will become clear). Your program will run in the REPL, which is where you will see any output and provide any input.

Basic screen for the Mu IDE



If you have permission to install software on your computer, I would recommend you download and install the IDE Mu.

If you can't install software on your computer but can access the internet, you can use the online IDE Trinket.

Mu

Mu is available to download at the website codewith.mu where you will also find comprehensive installation instructions.

If you have the latest Raspbian Mu is already installed and can be found in the Programming section.

Note: When Mu starts for the first time, select the **Python 3** mode and click **OK**.

Note: If you experience problems or would just like to know more about Mu, have a look at Raspberry Pi's Getting started with Mu guide at :

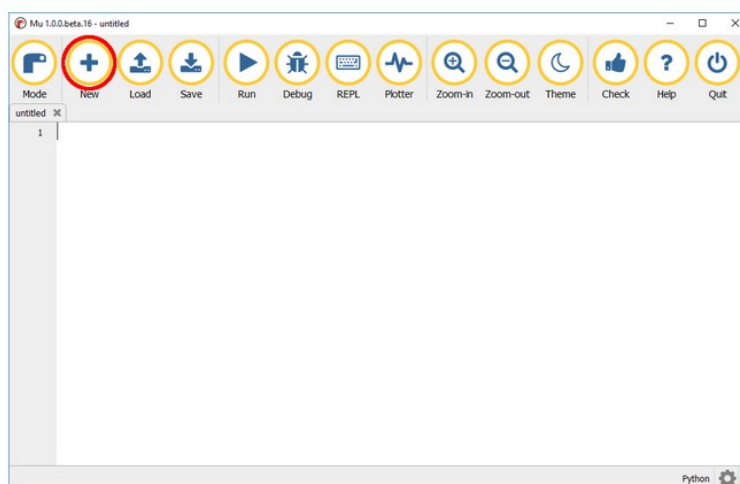
<https://projects.raspberrypi.org/en/projects/getting-started-with-mu>

Write your first Python program

The start of every new programming experience is creating a “Hello World” program. It's one of the simplest programs you can create, and it will put the message “hello world” on the screen.

Programmers will often create a “Hello World” program to test that everything is working before they start anything new, and this is what you are going to do.

Create a new program in Mu by clicking the **New** icon.

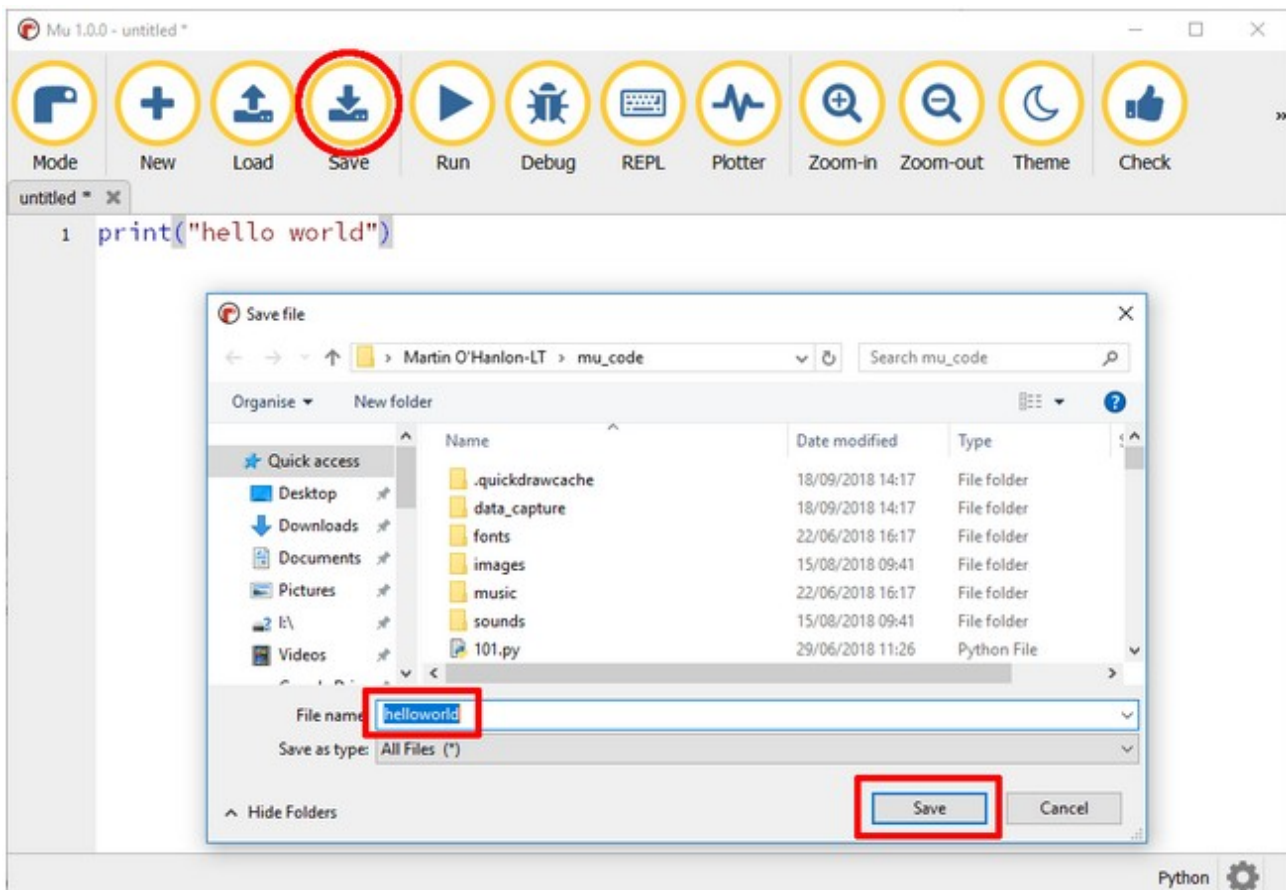


Type the following code into the editor:

```
print("hello world")
```

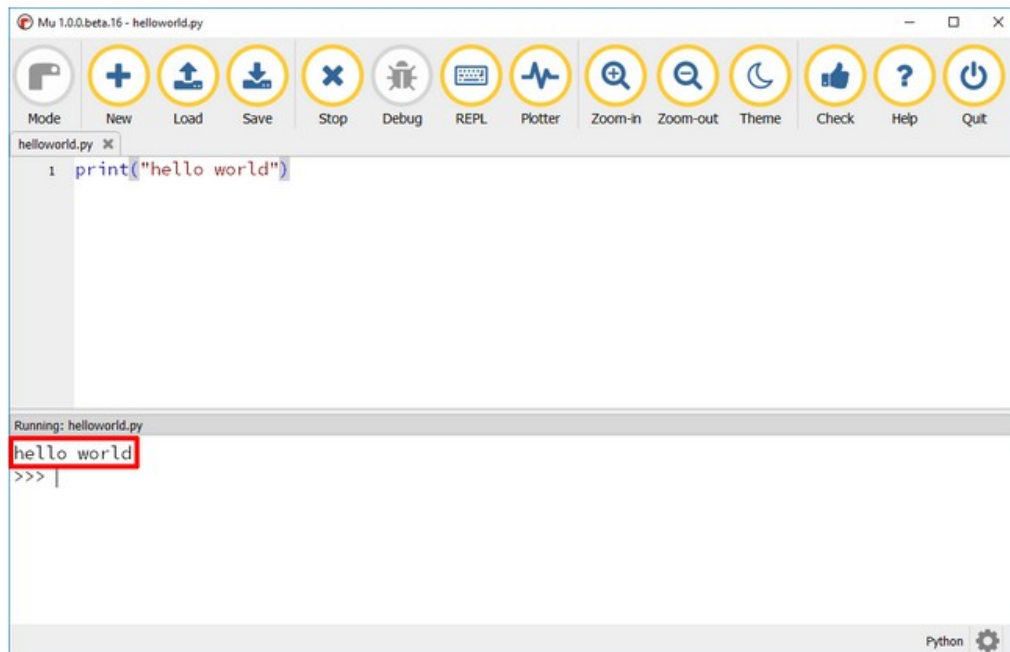
Click Save to save the program.

Enter the file name helloworld and click the Save button.



Click **Run** to run the program.

The message “hello world” should appear in the REPL.



Any errors in your program will appear in the REPL with a description.

If you get an error, look over the code carefully to make sure everything is correct.

There are a few important points to make which might help if you experience an error:

- The code is **case-sensitive**, so capital letters are important - `print` is not the same as `Print`
- Text, such as a message to be printed, needs to be between speech marks so be sure to put the "hello world" message in between " "
- `print` expects the message to be in between parentheses (brackets)
- If you need to change your program, stop it and run it again by clicking Stop and then Run.

Change the output

Two main characteristics of a computer are that it can handle input and output. You have already learnt how to create an output using `print` to put a message on the screen.

In this activity you will:

- continue to explore the use of `print` to output text to the screen
- use variables to store information in your program
- learn how to use `input` to get text from the user

Displaying “hello world” on the screen isn’t a very typical way to introduce yourself, so you should change the program so it says “Hi” rather than “hello world” when it starts.

Your program only has one instruction at the moment: `print("hello world")`. Whatever is in between the speech marks (") is what is displayed (printed) to the screen.

- Change the `print("hello world")` instruction in your program to display “Hi”:

```
print("Hi")
```

- Run the program by clicking **Run**.

The message “Hi” should now be displayed in the REPL.

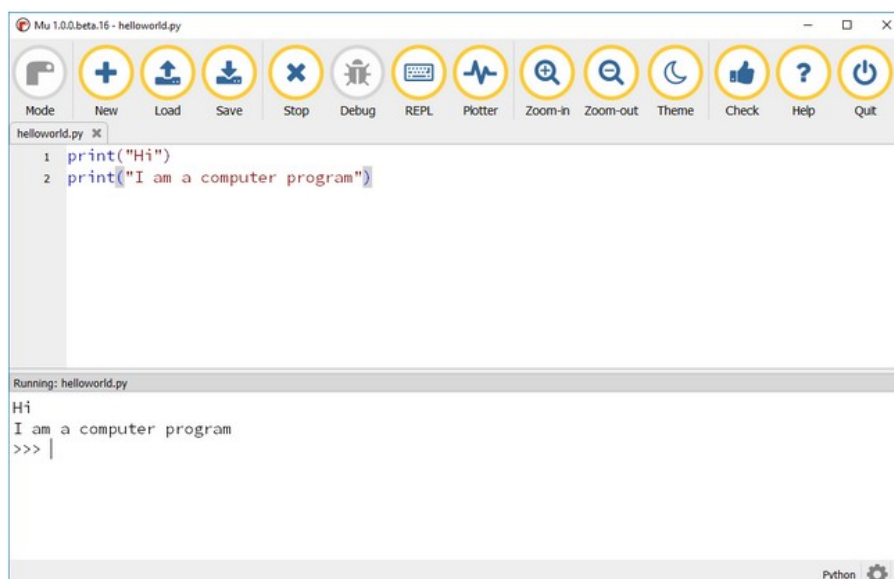
To add more messages, you can use additional `print` instructions on the lines below saying other things.

- Add a second message to your program:

```
print("Hi")  
print("I am a computer program")
```

- Run the program to see your second message.

Tip: - you may need to stop your program before you can run it again, by clicking the Stop icon.



Note how the messages are displayed in this order:

```
Hi  
I am a computer program
```

This represents an important concept in computer programming: **sequencing**. The messages are guaranteed to always appear in this order, as the computer runs each of these instructions in the order given and one at a time.

Variables

Variables are a way of storing pieces of data in your program. When you create a variable, an area of the computer's memory is reserved for you and the data is stored in it. This area of memory is now controlled by you: you can retrieve data from it, change the data, or get rid of it all together.

To create a variable in Python you give it a name and make it equal to a value. This is known as **assignment**. For example:

```
my_variable = "some useful data"
```

Next you will change your program to store your name in a variable and then display it.

- Add the following code to the bottom of your program to create a new variable called `my_name` to hold your name.

```
my_name = "Martin"
```

You should put your name (or any other name you like!) in between the speech marks: "Martin".

When your computer runs this instruction, a variable called `my_name` will be created and the text of your name will be stored.

Once a variable has been created, it can then be used in your program.

- Use `print` to output your name to the screen by adding this code to the bottom of your program.

```
print(my_name)
```

Notice how when using `print` you put the name of the variable between the () instead of text. The `print` instruction will retrieve the value from the `my_name` variable and display it.

- Run your program to see your name appear under the first 2 print statements.

Tip: a common problem is to put the variable name inside speech marks, e.g. `print("my_name")`. (What will this do? If you are not sure, try it!)

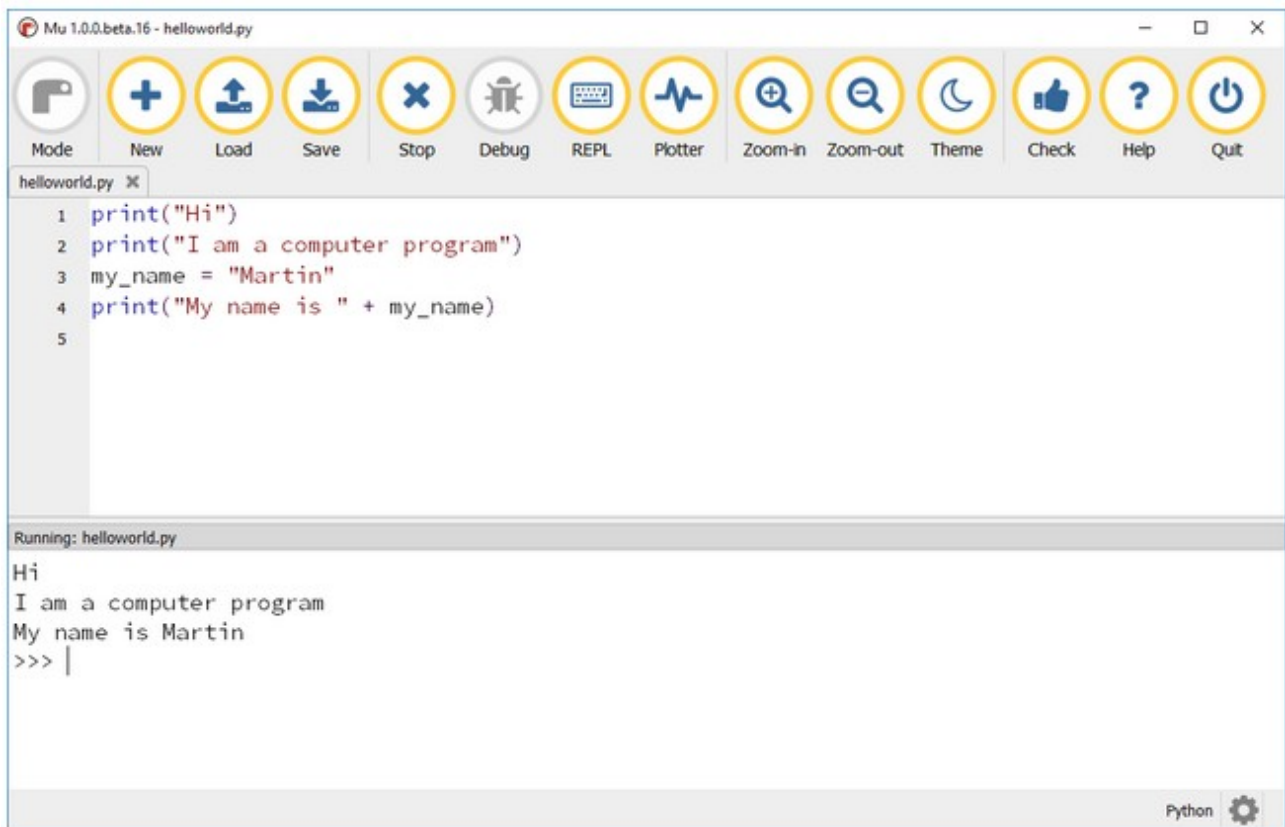
You can improve the message by putting the text “My name is “ in front of your name.

- Modify the code to add the text "My name is " before the my_name variable:

```
print("My name is " + my_name)
```

Note the use of the + sign to add the first piece of text to the text in the variable. Adding pieces of text together like this is known as **concatenation**.

- Run your program.



Reflect: Which line of code would you change if you wanted the final printed line to say “My name is Robin” instead? (There may be more than one answer!)

Getting input from the user

So far you have only been using `print` to output text; your program currently lacks the ability to interact with the user.

In this section you will be using `input` to prompt the user to enter some data.

`input` will put a message on the screen, wait for the user to give a response and press Enter, and then store the response as text in a variable so you can use it in your program.

`print` and `input` are examples of *functions*. Functions are pre-made programs which are typically created to perform a single task. Python has many inbuilt functions allowing you to perform a large range of tasks.

- Add the following code to the bottom of your program to use `input` to capture the user's name and store it in a variable.

```
users_name = input("What is your name? ")
```

Breaking down this line of code you can see what each element does:

- **`users_name =`** creates a variable called `users_name` and assigns it a value
- **`input()`** uses the function `input` to display a prompt and capture the user's response
- **`"What is your name? "`** – the value between the `()` – is the prompt which will be output to the screen

Now the text the user entered is stored in an variable (`users_name`) you can use the data in your program.

- Lets use `print` to output the user's name.

```
print(users_name)
```

- Improve the output by adding some text to the start and end of the message.

```
print("Hello " + users_name + ", welcome.")
```

- Grab a friend, run your program and ask them to input their name.

Note: you should enter your name in the REPL after the “What is your name?” prompt and press Enter when you have done so.

Syntax

Now you are starting to create more complex instructions, it's time to talk about **syntax**, the inevitable "syntax errors", and tips for resolving them.

Syntax is a set of rules which describe how the code must be laid out. Computers are sticklers for precision, and if you break the rules they will tell you in the only way they know how: with errors. They just don't have the imagination to cope with anything unusual!

Let's talk through some of the syntax rules for this line of code:

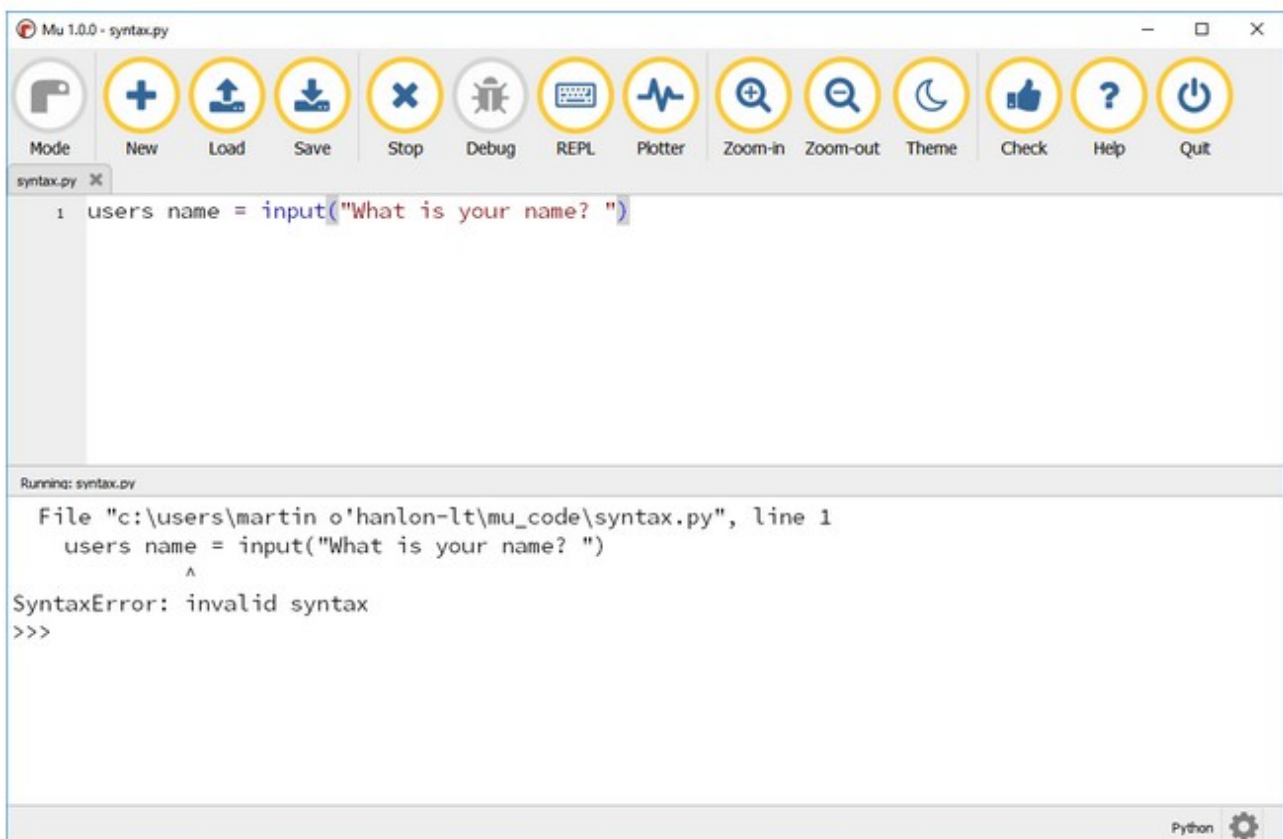
```
users_name = input("What is your name? ")
```

`users_name` – this is the variable name and there are syntax rules about how variables can be named, one being that variable names cannot contain a space.

If you were to run the following code, you would be presented with an error (also below):

```
users name = input("What is your name? ")
```

```
^  
SyntaxError: invalid syntax
```



Python is telling you that you have broken the rules and you need to fix the problem otherwise your program cannot be run.

In this snippet of code, the function `input` is being used. We must give it a prompt for the user, and the syntax rules say we must enclose this prompt in round brackets (). Missing out either or both of the brackets will result in an error.

```
users_name = input "What is your name? "  
                ^  
SyntaxError: invalid syntax
```

There are many rules which govern syntax and a deep understanding comes with practice and experience, but the basics are easy to understand and will be introduced throughout this course.

Tip: when presented with a `SyntaxError`, use the error message to find the **approximate** position of the error. It could be the line before or after the one shown. Be critical of your own code or get someone else to look over it for you.

Print and input instructions with your students

What other activities could you create just using the `print` and `input` instructions?

Add a comment describing your ideas for activities and get inspiration from others.

Case studies: remixing code

Our educators have a variety of methods to avoid students just copying and pasting code that they don't understand, instead adding to or remixing code. Here are a few of them:

- High school teacher Joe Mazzone avoids giving students copy-and-pastable code:
“I have them type all the code so there is never copy and pasting (use pictures of code for your assignments instead of plain text). All of my activities are skill-building and lead to a project where students write their own unique code.”
- Grade 5-8 teacher Bob Irving also prevents copying and pasting, this time by using printed out handouts:
“I have a series of “hackpacks” that I use in teaching Minecraft coding in Python, based mainly on the book by O’Hanlon and Whale. I print out the hackpacks, and the students must type in the code. After each program, I assign what I call a “reverse”, which is a challenge to change that code to something slightly different. I firmly believe that they must TYPE the code to really learn it. Copying and pasting is useful when you get a lot farther along.”
- Former school teacher Laurel Bleich uses copying code as a teachable event:
“I spend a lot of time on this because it is a such a touchy subject. I taught Middle School, and the students are still trying to understand what plagiarism is. So my first rule is if they do take code from somewhere else (unless it is code we developed as a class) then they are to cite the source in the comments in their code. The second issue with this is that many times the kids do not understand what they are copying and I end up having to help them debug that code. I found I can use this to push back a bit with the student and have a conversation about not blindly copying code. A way I worked through this was to have the students use comments in the copied code to explain what was going on.”

Create a new project

Now you are going to create a project which you will work on throughout the course. It's going to be a bot, which will help you with day to day tasks.

First you will need to create a new program for your bot. Here's how to do that in Mu

Mu

Click New to create a new program.

Click **Save** to save your program with the filename "bot".

Introductions

When your program starts, your bot should introduce itself by using `print` to put a message on the screen.

- Add the code to your program to print a welcome message.

```
print("Hi, I am Marvin, your personal bot.")
```

You can use whatever message you want. Perhaps you want to display a few messages, such as "Let's get started".

Challenge

1. Use `input` to ask the user's name and store it in a variable called `users_name`.
2. Use `print` and the `users_name` variable to display a "Welcome [name]" message.

Calculation

You are now going to program your bot to help add numbers together. It will ask the user for two numbers, add the numbers together, and display the result.

In order to do this, your program will need to create **variables** for the following pieces of data:

1. The first number the user inputs
 2. The second number
 3. The result of adding the first and second inputs
- Add code to ask for the two numbers and store them in variables, by adding the following code to the end of your program.


```
print("lets add some numbers")
input1 = input("Number 1> ")
input2 = input("Number 2> ")
```

- Create a new variable to store the sum of the two numbers entered.

```
result = input1 + input2
```

- Print the variable `result` to the screen.

```
print(result)
```

Your complete code should look like this:

```
print("Hi, I am Marvin, your personal bot.")
print("lets add some numbers")
input1 = input("Number 1> ")
input2 = input("Number 2> ")
result = input1 + input2
print(result)
```

- Run your program, enter 2 numbers and see what the result is.

The result might not be what you were expecting. If you enter 1 and 2, you get the result 12 and not 3.

Instead of adding the two numbers, your program has concatenated the two inputs as if they were text.

To resolve this problem you need to tell the computer how to interpret the data it's been given. It has interpreted the input as a piece of text like "1", but we want it to interpret it as a number like 1.

Problems in computer programs are known as **bugs**, and it's usual when programming to come across bugs which need to be fixed. In the next step you will go through the process of fixing this problem, which is known as **debugging**.

Reflect: Before you move on to the next step, explain to yourself what each line of your code is doing and why the program doesn't give you the expected result.

Text and numbers

In the previous step you didn't get the result you were expecting because `input` stores **text** in the variable `input1`, instead of a number.

```
input1 = input("Number 1> ")
```

Variables in Python not only store data; they also have a **data type** which says what sort of data the variable holds.

Variables that store text data have the data type **string**, whereas variables that hold whole numbers have the data type **integer**.

When the user types some input, it is stored as a string. In order to get Python to add the numbers together, you need to convert the variables `input1` and `input2` from strings to integers.

Converting variables from one data type to another is known as **casting**.

To cast your **string** variables `input1` and `input2` to **integer** you can use the `int` function (the name is short for 'integer').

- After you have got the two numbers using `input` with these lines:

```
input1 = input("Number 1> ")
input2 = input("Number 2> ")
```

- add the following code to create two new variables `number1` and `number2`, and use `int` to cast the `input` strings.

```
number1 = int(input1)
number2 = int(input2)
```

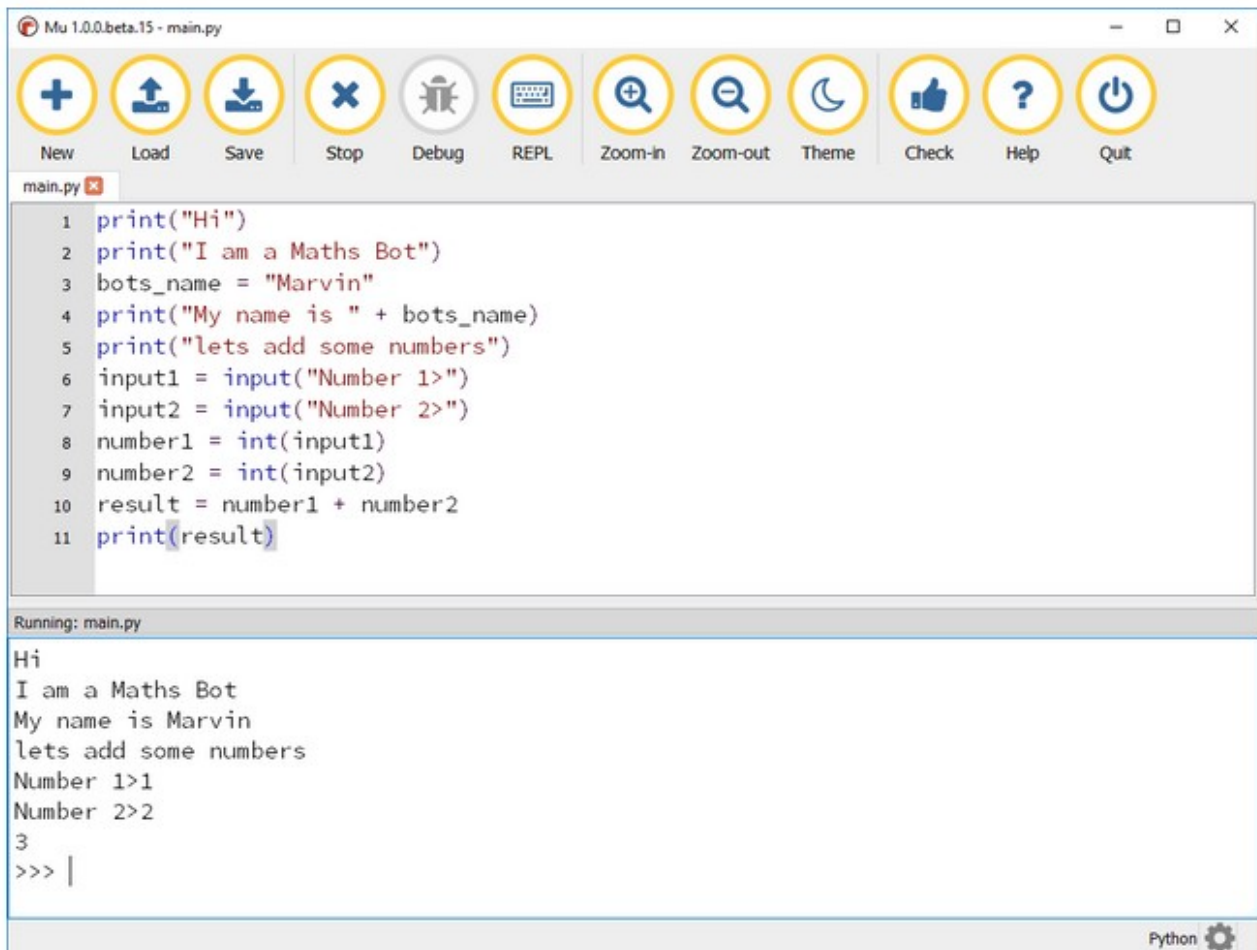
- Modify your program so that the `result` variable is the addition of the variables `number1` and `number2`.

```
result = number1 + number2
```

Your complete code should now look like this:

```
print("Hi, I am Marvin, your personal bot.")
print("lets add some numbers")
input1 = input("Number 1> ")
input2 = input("Number 2> ")
number1 = int(input1)
number2 = int(input2)
result = number1 + number2
print(result)
```

- Run your program. You should now get the correct result.

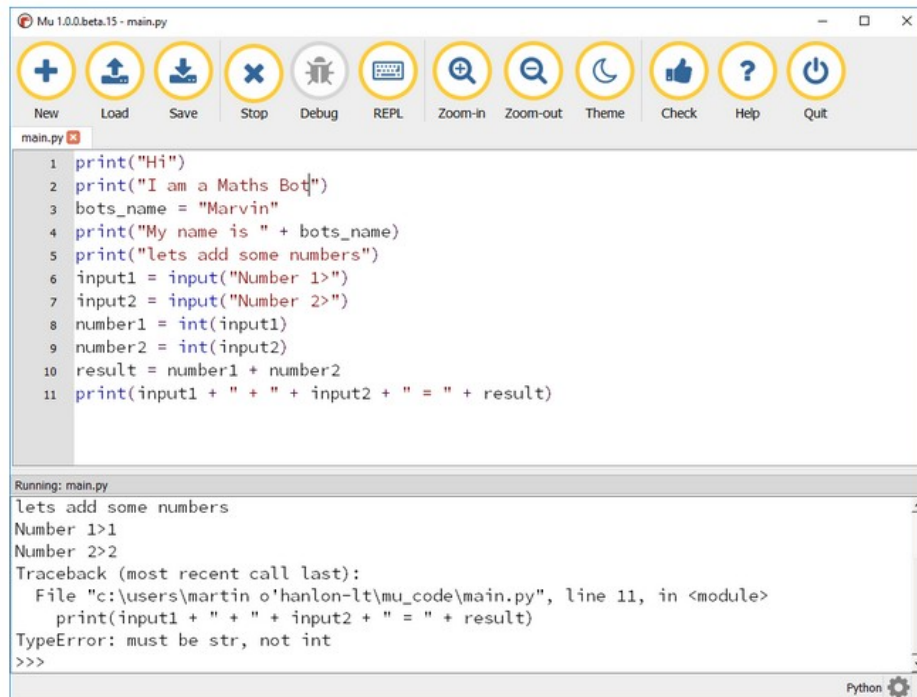


At the moment the output of the result is very basic. Using string concatenation, we can improve the result so it looks like this: $1 + 2 = 3$.

- You might try this line, to replace the instruction `print(result)`:

```
print(input1 + " + " + input2 + " = " + result)
```

- Now run your program again. Oh dear! the change has resulted in an error:



```
print(input1 + " + " + input2 + " = " + result)
TypeError: must be str, not int
```

The error message `TypeError: must be str, not int` is telling you that it cannot concatenate an `int` (integer) to a `str` (string).

Just as you had to cast strings into integers to do calculations on them, you also have to cast the `result` variable which is an integer to a string so it can be concatenated.

- After the code where you calculate the `result`:

```
result = number1 + number2
```

- add a new line of code to create a new variable called `output` and use the `str` function to cast the variable `result` to a string, like this:

```
result = number1 + number2
output = str(result)
```

- Change the final line of your program which prints the `result` variable to use the `output` variable:

```
print(input1 + " + " + input2 + " = " + output)
```

- Run your program and test it to see if you now get the correct result.

```

1 print("Hi")
2 print("I am a Maths Bot")
3 bots_name = "Marvin"
4 print("My name is " + bots_name)
5 print("lets add some numbers")
6 input1 = input("Number 1>")
7 input2 = input("Number 2>")
8 number1 = int(input1)
9 number2 = int(input2)
10 result = number1 + number2
11 output = str(result)
12 print(input1 + " + " + input2 + " = " + output)

```

Running: main.py

```

Hi
I am a Maths Bot
My name is Marvin
lets add some numbers
Number 1>1
Number 2>2
1 + 2 = 3
>>>

```

Other data types

You have learnt about the variable data types `string` and `integer` which are used to store text and whole numbers. Python also has many other data types. These are some of the most commonly used.

| data type | description | example |
|----------------------|--|-----------------------------|
| <code>boolean</code> | Used for storing values which can only ever be true or false | True or False |
| <code>float</code> | Similar to <code>integer</code> but for non whole (decimal) numbers | 1.234 |
| <code>list</code> | Used to store many items of data in an order e.g. a list of people's names | ["Martin", "Rik", "Hitesh"] |

During this course you will use all of these data types as you continue to make your bot.

Challenge

Create a new program which solves a different maths problem, such as:

- adding three numbers together

- subtracting numbers
- calculating the area of a square
- or anything else you can think of.

Glossary

- **sequence:** describing code running one line at a time in order
- **function:** a self contained piece of code which performs a specific task
- **parameter:** data passed to a function
- **code:** the instructions within a program
- **variable:** a stored value in your program which is reference by a name
- **data type:** defining the type of data stored in a variable (string, integer)
- **string:** a text-based value
- **integer:** a whole number
- **bug:** a problem with a program not behaving as intended
- **debugging:** the process of finding and resolving bugs
- **IDE:** Integrated Development Environment
- **concatenation:** adding text together
- **syntax:** the correct structure or layout of code

How computers make decisions

Last week you created your first computer programs which followed instructions, dealt with input and output and used variables to store data.

The programs you created last week always followed the same path. They were **sequential**: every instruction was carried out in sequence so they will always do the same thing.

To break computers out of this sequential pattern of mindlessness you need to give them the ability to make choices.

For example:

- Is it raining? If so, use an umbrella.
- Am I on fire? If so, extinguish the flames.
- Can I safely jump over the cliff? If not, stay where I am!

Making choices in a computer program is known as **selection**, and involves working out whether something is **true** or **false**. If it's true, you take an action; if it's false you don't (and perhaps you take a different action).

“Is the height of the cliff less than the height I can fall?”

- True – Jump off, you will be fine.
- False – DON'T JUMP!

True or False

Computers make choices by determining if the value of a statement is **True** or **False**. This is known as **evaluation**. If the statement is **True** then the program will run the next instruction, otherwise it won't.

You can try out some evaluations by using the REPL, also called the console in Trinket. The REPL allows you to run one line of code at a time and see the result immediately. It's a great tool for testing to see how something works.

Tip: REPL stands for “**Read-Eval-Print Loop**”. it reads one line of code, evaluates it, prints the result and loops back to the start.

Mu

If you are using Mu, follow these instructions to use the REPL:

- Open Mu and click on the “REPL” button.

Try some evaluations

- Enter the following into the REPL window and press Enter:

```
1 == 1
```

Note: the double == operator means “**is this equal to?**” as opposed to a single = which follows a variable and means “**make this equal to**”.

When you press Enter, **True** will be displayed. This is because it's **true** that **1 is equal to 1**.

- Try entering `1 == 2`. **False** will be displayed, because it's **false** that **1 is equal to 2**.
- Enter these evaluations and see if you can predict whether the outcome will be **True** or **False**.

```
1 < 2
```

The operator < means **less than**. Ask yourself **is 1 less than 2?** If the answer is yes then the statement is true.

```
1 > 2
```

> means **greater than**, so is **1 greater than 2?**

```
1 != 2
```

!= is the operator for **is not equal to**, which means this statement means **is 1 not equal to 2**. != is the opposite of the == operator.

Whats do you think will be the result of:

```
1 < 1
```

This evaluates as **false** because 1 is not **less than** 1. However, instead of <, you can use <= test if something is **less than or equal to**:

```
1 <= 1
```

This would give True.

You can evaluate other data types, not just integers (whole numbers) as you have been doing.

- Try some string evaluations:

```
"hi" == "bye"
```

```
"hi" != "bye"
```

What do you think the result would be for:

```
"a" < "b"
```

- Run it in the REPL and observe the result.

Perhaps the result makes sense to you, but what about other character evaluations such as "=" < "!"? Discuss the results in the comments and try to draw some conclusions as to why you get the results you do.

Evaluation is one of the core concepts of computer programming, so it's important that you understand the principle. Throughout this course there will be lots of opportunities to practise using and creating statements to be evaluated.

You can close the REPL by clicking on the REPL button in Mu

Create an if statement

- Add the following code, which will use `input` to get some text from the user and store it in a variable called `phrase`.

```
phrase = input("Talk to me > ")
```

- Use an `if` statement to check if the phrase typed by the user is equal to “hi”.

```
if phrase == "hi":
```

Note: remember that a double equals sign is used (`==`) when testing whether two things are equal.

- Indented under your `if` statement, add the code to print out “hello”.

```
    print("hello")
```

Note: the indentation (gap) before `print` is very important. This is how Python knows what code to run if the statement is True. When you pressed enter after the colon `:` your IDE may have automatically indented the cursor for you, if not you can use the TAB key to create an indent.

- Run your program and enter “hi” when prompted. You should see the message “hello”.

If you put a space before “hi”, you won’t see the message “hello”. Why do you think this is? Put a comment in the discussion with your thoughts.

- Run your program again but enter a different message. You should notice that you don’t see the “hello” message. The “hello” message will only be shown if you enter “hi”.

Next you will change your program so that it always says “bye” when the program finishes. To have code run after the `if`, you need to write it without an indent, otherwise Python will still think it is part of the `if`.

- Add a print statement to the very end of your program:

```
print("bye")
```

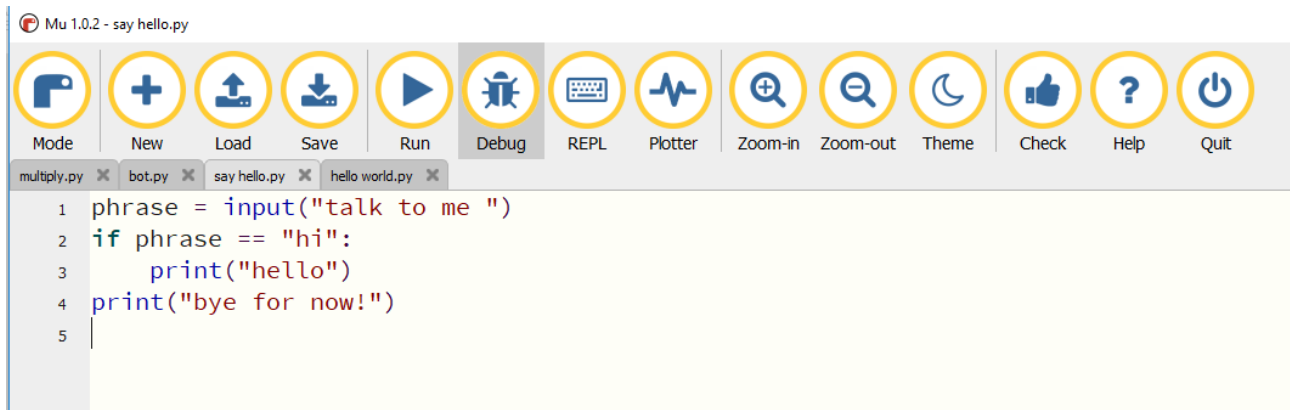
Note: there is NO indent before `print("bye")`.

Your program should now look like this.

```
phrase = input("Talk to me > ")
if phrase == "hi":
    print("hello")
print("bye")
```

- Run your program a few times. Experiment by entering the phrase “hi” and then different greetings. Do you get the results you expect?

Your program should only say “hello” if you say “hi”, but it should always say “bye”.



Reflect: What real-world situations do you think could be represented by an “if–then” statement? Share your thoughts in the comments.

??

If it's this, then do that

Now that you have an understanding of how to make computers test whether something is true or false, you can use this in your programs using `if`.

When a computer is asked to make a choice, it carries out a process like this:

if this *statement* is true **then** I will run this *code*”

This is known as **selection**. Your job as the programmer is to provide:

1. the *statement* which will be evaluated as either true or false, and
2. the *code* which will be performed if the statement is true.

The instruction is known as an **if–then** and is one of the most commonly used structures in programming. A version of the **if–then** exists across all general purpose programming languages.

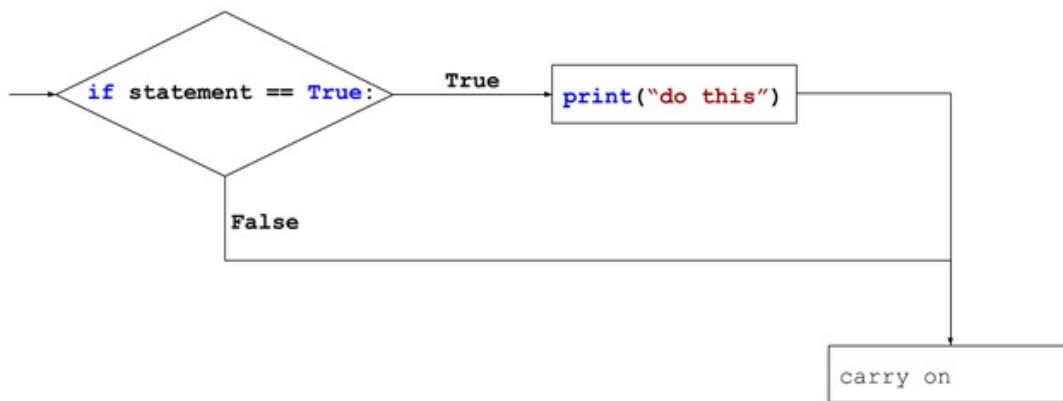
The syntax of an **if–then** in Python looks like this:

```
if statement == True:
    print("do this")
```

Between the `if` and the `:` is the code which will be evaluated, this is known as the **condition**, namely `statement == True`.

Indented under the `:` is the code which will be run if the statement is equal to `True`, e.g. `print("do this")`.

A flow chart of this **if, then** would look like this:



Let's put this into practice by creating a simple program which will talk back to you when you type certain phrases. For example, if you say "hi" it will say "hello".

Mu

If you are using Mu, follow these instructions to create a new program:

- Click on the **New** icon to create a new program.

Click **Save** and give it the filename `ifs.py`.

Create an if statement

- Add the following code, which will use `input` to get some text from the user and store it in a variable called `phrase`.

```
phrase = input("Talk to me > ")
```

- Use an `if` statement to check if the phrase typed by the user is **equal to** "hi".

```
if phrase == "hi":
```

Note: remember that a double equals sign is used (`==`) when testing whether two things are equal.

- Indented under your `if` statement, add the code to print out "hello".

```
    print("hello")
```

Note: the indentation (gap) before `print` is very important. This is how Python knows what code to run if the statement is `True`. When you pressed enter after the colon `:` your IDE may have automatically indented the cursor for you, if not you can use the TAB key to create an indent.

- Run your program and enter "hi" when prompted. You should see the message "hello".

If you put a space before "hi", you won't see the message "hello". Why do you think this is? Put a comment in the discussion with your thoughts.

- Run your program again but enter a different message. You should notice that you don't see the "hello" message. The "hello" message will only be shown if you enter "hi".

Next you will change your program so that it always says "bye" when the program finishes. To have code run after the `if`, you need to write it without an indent, otherwise Python will still think it is part of the `if`.

- Add a print statement to the very end of your program:

```
print("bye")
```

Note: there is NO indent before `print("bye")`.

Your program should now look like this.

```
phrase = input("Talk to me > ")
if phrase == "hi":
    print("hello")
print("bye")
```

- Run your program a few times. Experiment by entering the phrase "hi" and then different greetings. Do you get the results you expect?

Your program should only say "hello" if you say "hi", but it should always say "bye".

Reflect: What real-world situations do you think could be represented by an "if-then" statement? Share your thoughts in the comments.

If it wasn't that, do this

In the previous step you learnt how to use an **if-then** statement to allow your computer to take an *action* when a *statement* was true. It is often really useful to take a different action if the statement is false.

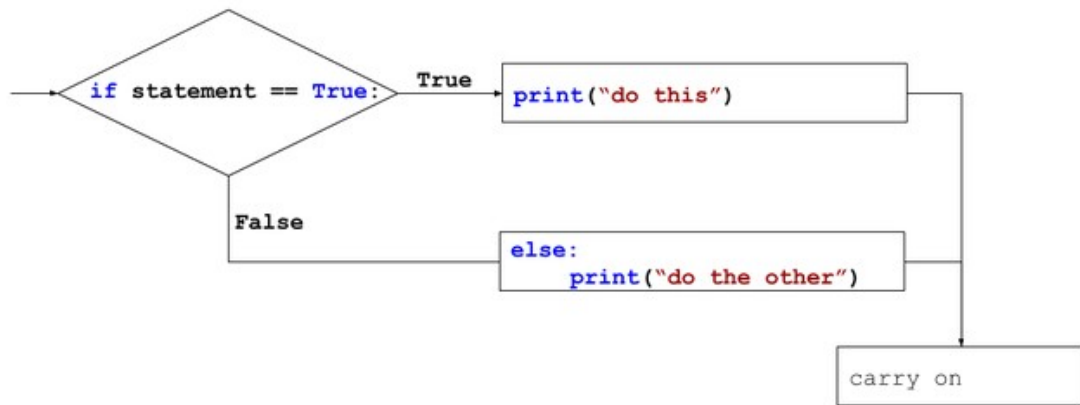
This is done using an **if-then-else** statement, which works like this:

"if this *statement* is true, **then** I will run this *code*, **else** I will run this *other code*"

The syntax for an **if-then-else** in Python looks like this:

```
if statement == True:
    print("do this")
else:
    print("do the other")
```

Note: The `else` isn't indented and is in line with the `if`. By aligning them, you tell Python that this `else` belongs to that `if`.



You will now change your program so that if the user enters anything other than “hi”, it replies with “sorry I dont understand this”.

- After the code `print("hello")` add an `else` and a `print` indented under the `else`, like this.

```

else:
    print("sorry I dont understand this")

```

Your complete code should look like this:

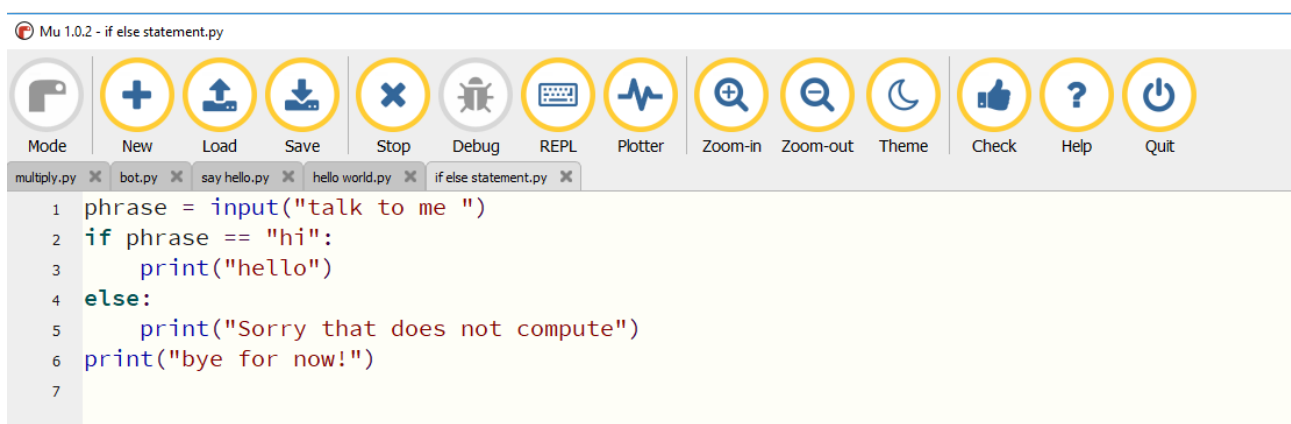
```

phrase = input("Talk to me > ")
if phrase == "hi":
    print("hello")
else:
    print("sorry I dont understand this")
print("bye")

```

Note: the `else` is not indented and is in line with the `if`. This is very important, as the level of indentation tells Python that this `else` belongs to this `if`.

- Run your program and test that you get the results you are expecting.



Indentation

Python and indentation are like siblings who always fight. They like each other because they are so closely related, but they are always causing each other problems.

Indentation problems are something almost everyone experiences the first time they use Python. However, if you understand and follow the rules they are very easy to fix. So let's take some time to talk about:

- why indentation is so important in Python
- how it works and how to create indents
- some tips to avoid problems

Indentation is important to Python because it's how you determine **scope**. Scope is how Python knows what code belongs to what part of the program.

Indent?

Lets recap by looking at some code you have already created:

```
phrase = input("Talk to me > ")
if phrase == "hi":
    print("hello")
print("bye")
```

This small program contains one indent: it's before the code `print("hello")` and it is under `if phrase == "hi":`:

```
if phrase == "hi":
    print("hello")
```

The indentation tells Python that the code `print("hello")` should only be run `if phrase == "hi"` is True. Any code that is indented under the `if` will form part of that `if` statement's **scope**.

```
if phrase == "hi":
    print("hello")
    print("this is also part of the if's scope")
print("not part of the if's scope")
```

Have a look at the program below, where there is another indent: after the `else` before `print("sorry I dont understand this")`.

```
phrase = input("Talk to me > ")
if phrase == "hi":
    print("hello")
else:
    print("sorry I dont understand this")
print("bye")
```

Using the same scope rules as the `if` statement, the `print` statement indented under the `else` is part of that `else` statement's scope.

```
else:  
    print("sorry I dont understand this")
```

The final `print("bye")` is not indented under the `if` or the `else`, so is not part of the `if` or the `else`. That makes sense as you want this line to run no matter what phrase the user entered.

Without the use of indentation, Python would not be able to determine what code to run as part of the `if` or the `else`. This is a common theme throughout Python and one you will use a lot in the coming weeks.

How it works

To create an indent using Mu or Trinket, use the TAB key on the keyboard. This will create a space and indent your cursor. Mu and Trinket (and most code editors) will also try and help you indent your code, automatically putting in indents when you press Enter after a `:`.

Although the TAB key will insert an indent, Mu and Trinket don't insert an actual TAB character for the indent: like most code editors, they will use spaces. If you are using Mu you will notice that Mu creates four spaces for an indent, whereas if you are using Trinket, it only uses two spaces. Most of the time you don't need to worry about this though – all you need to do is press TAB.

Python doesn't mind whether you use two spaces or four spaces (or any other number of spaces) as long as you are consistent. For example if you start off using four spaces for an indent, then you should *always* use four spaces.

In the example below, four spaces have been used for the first indent, but only two for the second, and you can see that as a result the code doesn't "line up".

The screenshot shows the Mu Python IDE interface. The title bar reads "Mu 1.0.2 - indentation error.py". The top toolbar contains icons for Mode, New, Load, Save, Stop, Debug, REPL, Plotter, Zoom-in, Zoom-out, Theme, Check, Help, and Quit. The file explorer shows several open files: bot.py, if else statement.py, say hello.py, multiply.py, hello world.py, and indentation error.py. The main editor window displays the following Python code:

```
1 #Indentation error!
2 phrase = input("Talk to me > ")
3 if phrase == "hi":
4     print("hello")
5     print("how are you?")
6
```

The bottom status bar indicates "Running: indentation error.py". The console output shows the error message:

```
File "c:\users\howard\mu_code\indentation error.py", line 5
    print("how are you?")
    ^
IndentationError: unindent does not match any outer indentation level
>>> |
```

So you see running this program will result in an error stating `IndentationError: unindent does not match any outer indentation level`.

Note: All Python requires is that you are consistent.

Tips for indentation

1. Use the Tab key on your keyboard. This will ensure you are using same number of spaces per indent.
2. Don't fight with your editor: let it indent code for you where possible, and use the number of spaces it gives you.
3. Keep your code tidy! If it's not lined up it's not going to work.

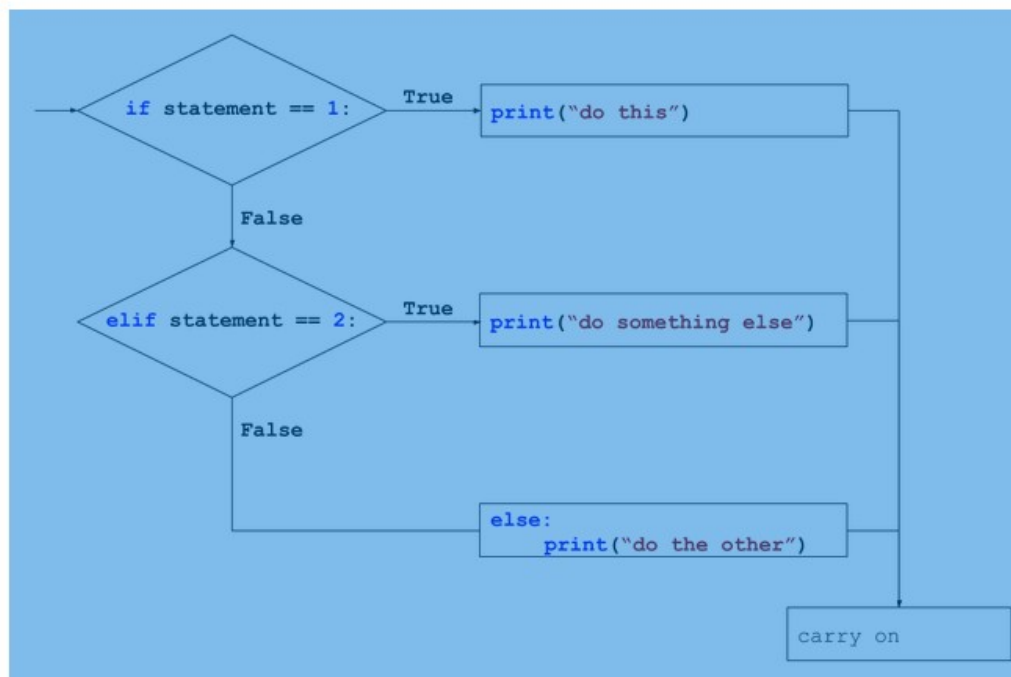
If it's not that, is it this?

You have learnt how to make your program test a statement and run some code depending whether it's true or false, by using an **if-then-else** statement. What if you want to test more than two possibilities, such as whether you entered “hello”, or “bye”, or “no way”? For these scenarios you can use an **else if**.

The syntax for an **else if** in Python, which abbreviates it to **elif**, looks like this:

```
if statement == 1:
    print("do this")
elif statement == 2:
    print("do something else")
else:
    print("do the other")
```

Note: The **elif** goes in between the **if** and the **else**.



Let's change your program so that it uses an **elif** to check if the phrase is “whats your name”, and to reply to that phrase with “Marvin”.

- After the code `print("Hello")`, use an **elif** and **print** out a name.

```
elif phrase == "whats your name":
    print("Marvin")
```

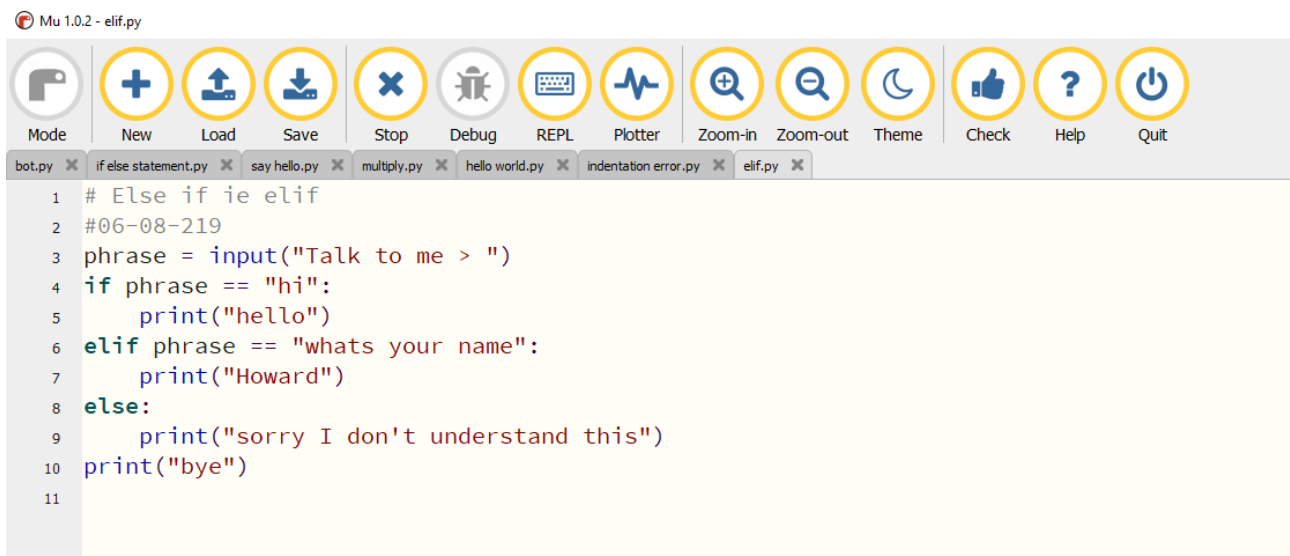

You complete code should look this this:

```
phrase = input("Talk to me > ")
if phrase == "hi":
    print("hello")
elif phrase == "whats your name":
    print("Marvin")
else:
    print("sorry I don't understand this")
print("bye")
```

Run your program and test to make sure your `elif` works.

The phrase you have tested for is “*whats your name*” not “*what’s your name*”, so when you test the program out, don’t include the apostrophe otherwise it will appear not to work. In a later step, you will learn how to test multiple evaluations.

My bit

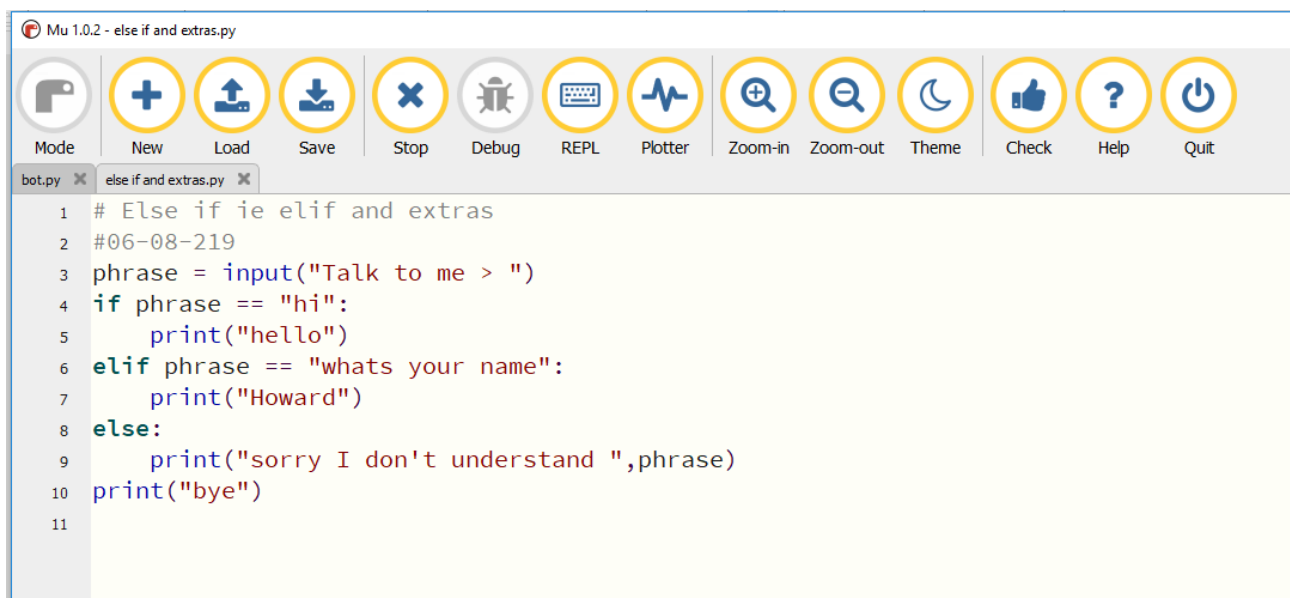


Challenge

Can you use the lessons you learnt last week to change your program so that instead of just saying:
sorry I dont understand this

it includes the phrase you entered? E.g. If you entered “what are you”, the program will respond with:

sorry I dont understand 'what are you'



The screenshot shows the Mu Python IDE interface. The title bar reads "Mu 1.0.2 - else if and extras.py". The toolbar contains icons for Mode, New, Load, Save, Stop, Debug, REPL, Plotter, Zoom-in, Zoom-out, Theme, Check, Help, and Quit. The code editor displays the following Python code:

```
1 # Else if ie elif and extras
2 #06-08-219
3 phrase = input("Talk to me > ")
4 if phrase == "hi":
5     print("hello")
6 elif phrase == "whats your name":
7     print("Howard")
8 else:
9     print("sorry I don't understand ",phrase)
10 print("bye")
11
```

Case studies: helping students when they are stuck

Getting stuck while working on a problem is common through Computer Science (and life in general), so it's important to have ways to get unstuck. Here are a few that some educators find useful in their classrooms:

Working with peers

"I always have the students in pairs (or at a table of 3) even if they are not pair programming. I think it is important that the students learn how to ask for help as well as provide help. This needs to be a class norm, no matter the class."

Laurel Bleich, former school teacher

"Talking the problem through to someone else (real or of the lego/plastic duck variety) also helps."

Jo Wakefield, secondary school teacher

"I encourage collaborations, and students have 'elbow partners': other students around them that they can trust and ask questions."

Joe Mazzone, high school teacher

"I get them to share their code with someone else and see if a different person can explain what should happen and where the problem might be."

Beverley McCormick, primary school teacher

Talking out loud

"I get the pupils to read through their code and explain what each step does."

Beverley McCormick, primary school teacher

"I have the student talk through what each line of code does out loud. This is a great debugging strategy and allows students to discover what they have missing or wrong."

Joe Mazzone, high school teacher

Having a list of strategies or common errors

"We have a checklist of common errors that they should look at first – brackets, closing speech marks, capital letters, etc, which catches a lot first."

Jo Wakefield, secondary school teacher

“I have a list on my board of strategies for getting unstuck: ‘Ask 3 B4 me’, ‘Error messages are your friends’, ‘Use another pair of eyes’, ‘Take some time and THINK’, ‘It’s usually spelling or punctuation’ ... and if all else fails, go get a drink of water and walk slowly back.”

Bob Irving, grade 5-8 teacher

“I give them a check list of things to do before they ask me or another a question, such as: 1) Is this error similar to another one I have had before and what did I do about it? 2) If it is different, where is the problem at? (I try and stress that while the IDEs try to be helpful in highlighting some errors, the actual bug is usually elsewhere.) Is there a problem with my logic? 3) Is there another program that I know of that does something similar to what I want to do? If so, what does the code look like? Similarities? Differences? 4) Always trying googling it or referring to the class webpage where I have LOADS of online resources. And finally 5) Ask the teacher or another student.”

Laurel Bleich, former school teacher

What do you think you’ll do to help your students get unstuck? Share any tips you have in the comments.

This "and" that "or" the other

Until now you have only been doing single tests when **evaluating** a statement e.g.

```
phrase == "hi"
```

Using **and** and **or**, you can build compound evaluations which test multiple statements, e.g.

```
phrase == "hi" and name == "Marvin"
```

```
phrase == "hi" or phrase == "hey"
```

Let’s walk through this example:

```
phrase == "hi" and name == "Marvin"
```

1. Python will evaluate both statements before and after the **and**:

1. `phrase == "hi"`
2. `name == "Marvin"`

2. Python will apply a logic **and** to the evaluations:

1. if statement 1 is **True and**
2. statement 2 is **True**
3. then the whole statement is **True**
4. otherwise it’s **False**

It is similar for the statement

```
phrase == "hi" or phrase == "hey"
```

1. Python will evaluate both statements before and after the **or**:

1. `phrase == "hi"` = True / False
2. `phrase == "hey"` = True / False

2. Python will apply a logic **or** to the evaluations:

1. if statement 1 is **True or**
2. statement 2 is **True or**
3. both are **True**
4. then the whole statement is **True**

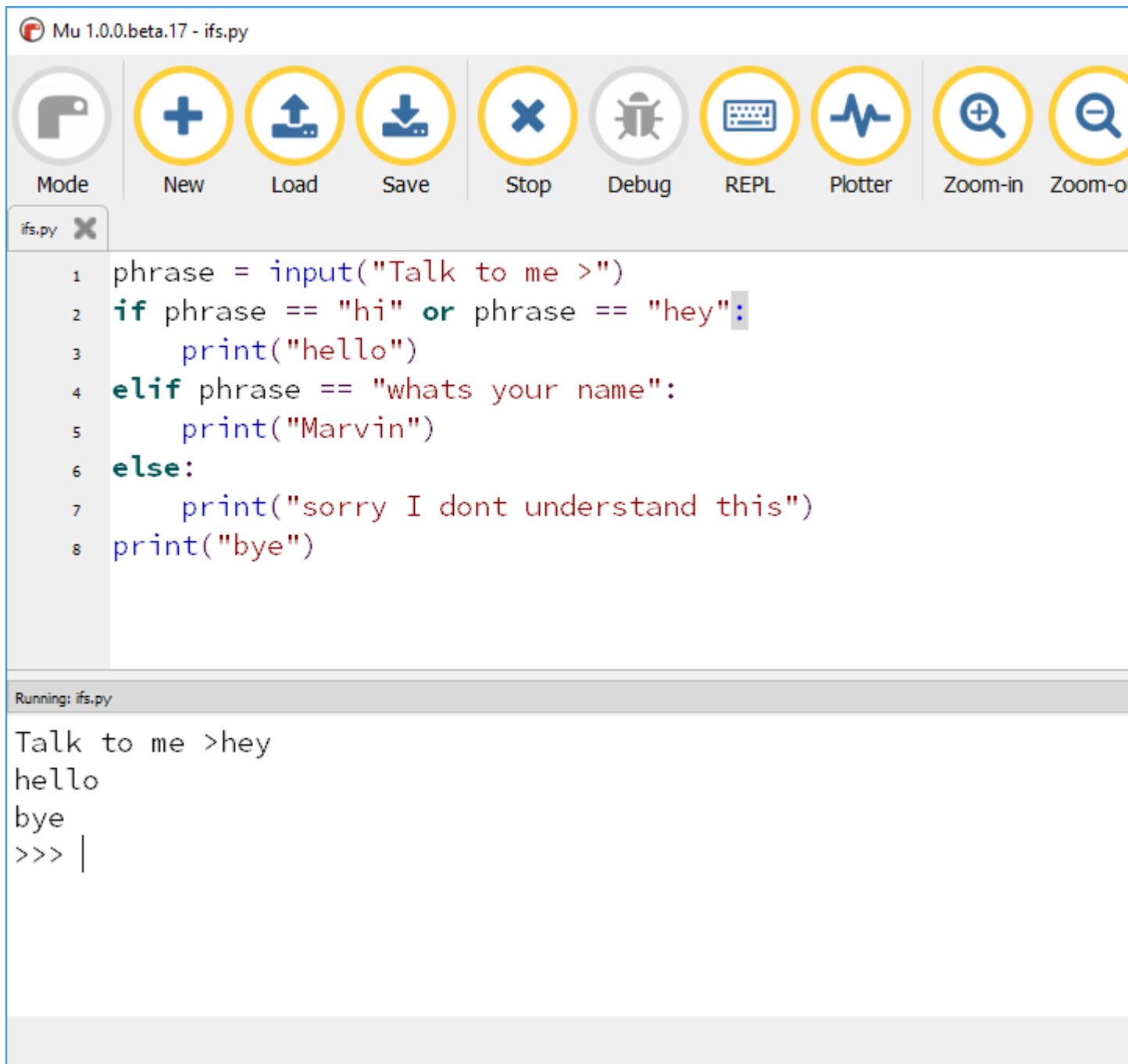
Let's use this example to change your program to use an **or**, so that it responds with "Hello" if you give the phrase "hi" or "hey".

- Change the first if statement in your program `if phrase == "hi":` to:

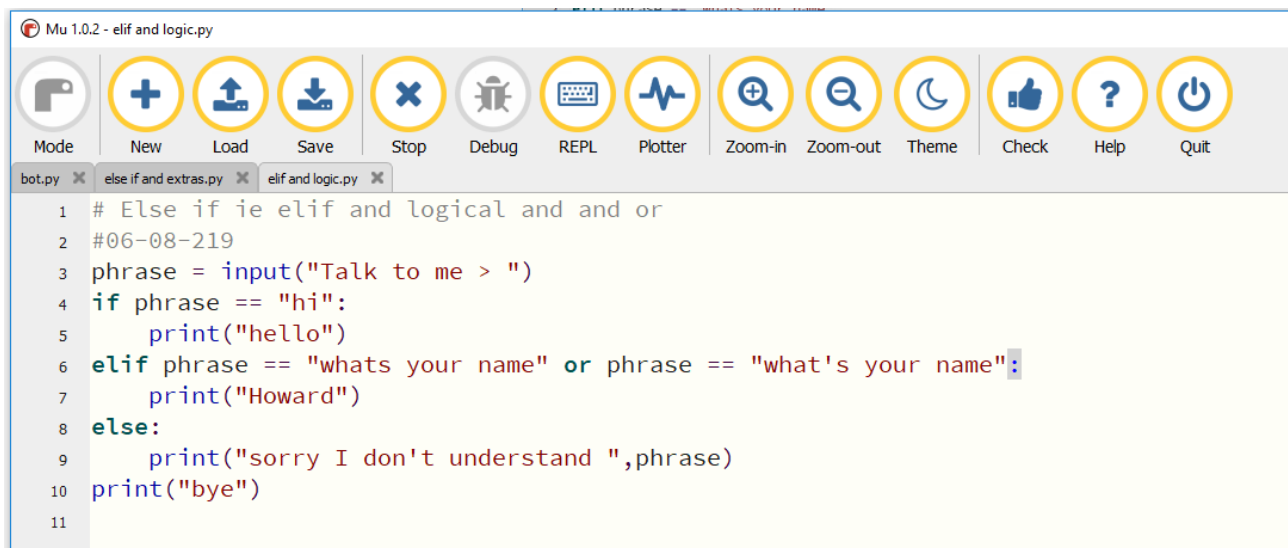
```
if phrase == "hi" or phrase == "hey":
```

Tip: a common error is to write `if phrase == "hi" or "hey"` as this is how we would say it in English, but in Python, you have to state the condition in full each time. This code will run, but not have the result you expect as it won't matter what phrase is input the if statement will always be true.

- Run your program. If you give the response "hi" or "hey" then your program should respond with "Hello".



Change your program so it tests for “whats your name” or “what’s your name”.
My bit



The screenshot shows the Mu Python IDE interface. The title bar reads "Mu 1.0.2 - elif and logic.py". The toolbar contains icons for Mode, New, Load, Save, Stop, Debug, REPL, Plotter, Zoom-in, Zoom-out, Theme, Check, Help, and Quit. The code editor displays the following Python code:

```
1 # Else if ie elif and logical and and or
2 #06-08-219
3 phrase = input("Talk to me > ")
4 if phrase == "hi":
5     print("hello")
6 elif phrase == "whats your name" or phrase == "what's your name":
7     print("Howard")
8 else:
9     print("sorry I don't understand ",phrase)
10 print("bye")
11
```

Getting to grips with **evaluation**, **selection** and using `if` is really important, and you should take time to practise by adding additional instructions, playing with the code and trying new things.

Let your imagination guide you and take some time to come up with your own program that uses evaluation and selection. Here are some ideas to get you started:

- Modify your program so it can respond to more phrases by adding additional `elif` instructions.
- Get your program to ask questions by adding additional `input` and `if` statements.
- Create a quiz where your computer asks a question and lets you know if you were correct.

Remember to share what you create with your colleagues.

Building commands into your bot

Now that you have learnt how to use **selection**, you can use these skills to upgrade the bot you created last week so that it can help you do more.

At the moment, your bot is only useful for adding numbers together. Let's change it so it can also subtract numbers.

You will need to add the code to:

- Get a command from the user using `input`. Does the user want to add or subtract?
- Use an *if, then, elif, else* to **choose** whether the bot is adding or subtracting.
- Add the code to get two numbers from the user, and the code to subtract them (when required).

Open your bot program

The first step is to use `input` to get a command from the user when your bot starts. If the command "add" is entered, your bot will add two numbers, using the code you've already written.

Go back to your bot code by opening the "bot" program you created last week.

Mu

- Your "bot.py" might still be open in another tab; if so, open it.

If not, click **Load**, and select the "bot.py" file.

Add an "add" command

- Add the code to get a command from your user under the first `print` statement (where the bot says `Hi, I am Marvin, your personal bot`).

```
command = input("How can I help? ")
```

- After you have got the command from the user, use an `if` statement to see if it is "add".

```
if command == "add":
```

- Below this `if` statement, you should then indent all the code to add numbers, so that it is part of the `if` and therefore only runs when the user enters the command "add".

```
print("Hi, I am Marvin, your personal bot.")
command = input("How can I help? ")
if command == "add":
    print("lets add some numbers")
    input1 = input("Number 1> ")
    input2 = input("Number 2> ")
    number1 = int(input1)
    number2 = int(input2)
    result = number1 + number2
```

```
output = str(result)
print(input1 + " + " + input2 + " = " + output)
```

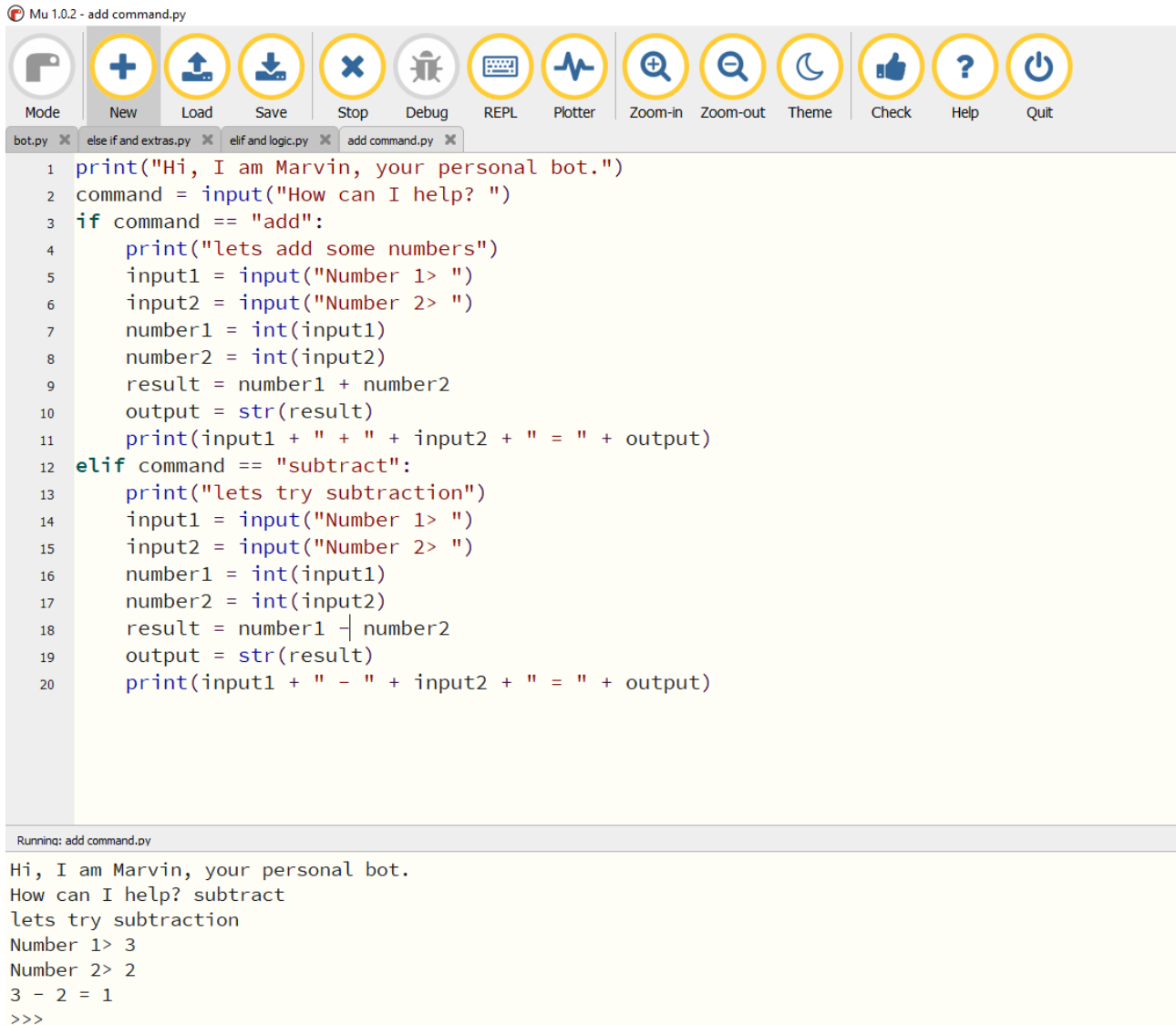
Tip: You can indent multiple lines in both Mu and Trinket by selecting them and pressing TAB.

Run your program to test it. If you enter the command “add” your bot should ask you for two numbers and add them together. If you enter anything else, the program should just finish.

Reflect: How would you change your bot to also accept “plus” as a command?

Next we will add the code for subtracting numbers.

My bit



```
Mu 1.0.2 - add command.py

1 print("Hi, I am Marvin, your personal bot.")
2 command = input("How can I help? ")
3 if command == "add":
4     print("lets add some numbers")
5     input1 = input("Number 1> ")
6     input2 = input("Number 2> ")
7     number1 = int(input1)
8     number2 = int(input2)
9     result = number1 + number2
10    output = str(result)
11    print(input1 + " + " + input2 + " = " + output)
12 elif command == "subtract":
13     print("lets try subtraction")
14     input1 = input("Number 1> ")
15     input2 = input("Number 2> ")
16     number1 = int(input1)
17     number2 = int(input2)
18     result = number1 - number2
19     output = str(result)
20     print(input1 + " - " + input2 + " = " + output)

Running: add command.py

Hi, I am Marvin, your personal bot.
How can I help? subtract
lets try subtraction
Number 1> 3
Number 2> 2
3 - 2 = 1
>>>
```

Subtracting

Now your bot has a way of understanding commands, you can change your program so that it can do more than just add numbers.

In this next step, you will add the code to enable your bot to subtract numbers. After this you will have the opportunity and challenge to add more code to undertake other tasks.

- Add an `elif` to the end of your program to test if the `command` is equal to `"subtract"`.

```
elif command == "subtract":
```

Note: the `elif` is not indented and is in line with the `if`.

- Indented under the `elif`, add the code to subtract two numbers. It is very similar to the code for adding.

Tip: it's OK to copy and paste when you have to repeat some code.

```
print("Let's subtract some numbers")
input1 = input("Number 1> ")
input2 = input("Number 2> ")
number1 = int(input1)
number2 = int(input2)
result = number1 - number2
output = str(result)
print(input1 + " - " + input2 + " = " + output)
```

- Run your program and test it.

There is a lot of duplicated code in your program right now. In Week 4, we will look at how we can change the logic of your program to make the code more efficient, but for now, it's a good idea to keep it simple.

As a last step, add a message so that if a command the bot doesn't understand is entered, it responds with `sorry I dont understand`.

- Add an `else` to the end of your program to print a message if it doesn't understand the command.

```
else:
    print("sorry I dont understand")
```

Your complete program should now look like this:

```
print("Hi, I am Marvin, your personal bot.")
command = input("How can I help? ")
if command == "add":
    print("lets add some numbers")
    input1 = input("Number 1> ")
    input2 = input("Number 2> ")
    number1 = int(input1)
    number2 = int(input2)
    result = number1 + number2
    output = str(result)
```

```

        print(input1 + " + " + input2 + " = " + output)
    elif command == "subtract":
        print("lets subtract some numbers")
        input1 = input("Number 1> ")
        input2 = input("Number 2> ")
        number1 = int(input1)
        number2 = int(input2)
        result = number1 - number2
        output = str(result)
        print(input1 + " - " + input2 + " = " + output)
    else:
        print("sorry I dont understand")

```

My bit

```

Mu 1.0.2 - add command.py
Mode New Load Save Stop Debug REPL Plotter Zoom-in Zoom-out Theme Check Help Quit
bot.py x else if and extras.py x elif and logic.py x add command.py x
1 print("Hi, I am Marvin, your personal bot.")
2 command = input("How can I help? ")
3 if command == "add":
4     print("lets add some numbers")
5     input1 = input("Number 1> ")
6     input2 = input("Number 2> ")
7     number1 = int(input1)
8     number2 = int(input2)
9     result = number1 + number2
10    output = str(result)
11    print(input1 + " + " + input2 + " = " + output)
12 elif command == "subtract":
13     print("lets try subtraction")
14     input1 = input("Number 1> ")
15     input2 = input("Number 2> ")
16     number1 = int(input1)
17     number2 = int(input2)
18     result = number1 - number2
19     output = str(result)
20     print(input1 + " - " + input2 + " = " + output)
21 else:
22     print("Sorry I don't understand ",command)

Running: add command.py
Hi, I am Marvin, your personal bot.
How can I help? multiply
Sorry I don't understand multiply
>>>

```

Comments

As your programs are getting more complex, you may find it more difficult to keep track of what each part of your code does.

A way to help you remember what different parts of your code do is to include **comments** within the code.

Comments are text within code that are notes for the programmer or another person using the code. The computer ignores all comments when it runs the code.

In Python, you create a comment by putting a hash character # before some text. For example:

```
# this is a comment
```

You can add comments anywhere within your program.

Typically, comments are reminders about what particular lines or sections of code do. For new learners such as yourself, comments are also a useful tool for explaining what programs do step by step.

```
# ask the user to enter their name  
name = input("what is your name?")  
  
# print a message including the name  
print("Hello " + name)
```

In Python, you can make a comment span multiple lines by putting a # at the start of each line.

```
# comments can span  
# multiple lines  
# if a hash character is placed  
# at the start of each line
```

Add some comments to your bot program to describe what it does.

My bit

<https://pastebin.com/3SMMGuvN>

Challenge

```
#add, subtract and multiply
#howard kirkman
# add command
# update to include subtract
# update to include multiply
#-----
#get command
print("Hi, I am Marvin, your personal bot.")
command = input("How can I help? ")
#if it's add get the numbers and add
if command == "add":
    print("lets add some numbers")
    input1 = input("Number 1> ")
    input2 = input("Number 2> ")
    number1 = int(input1)
    number2 = int(input2)
    result = number1 + number2
    output = str(result)
    print(input1 + " + " + input2 + " = " + output)
#if it's add get the numbers and add
elif command == "subtract":
    print("lets try subtraction")
    input1 = input("Number 1> ")
    input2 = input("Number 2> ")
    number1 = int(input1)
    number2 = int(input2)
    result = number1 - number2
    output = str(result)
    print(input1 + " - " + input2 + " = " + output)
#if it's add get the numbers and multiply
elif command == "multiply":
    print("lets try multiplication")
    input1 = input("Number 1> ")
    input2 = input("Number 2> ")
    number1 = int(input1)
    number2 = int(input2)
    result = number1 * number2
    output = str(result)
    print(input1 + " times " + input2 + " = " + output)
#if command is unknown say so
else:
    print("Sorry I don't understand ",command)
```

Your challenge is to update your program so it can:

- solve division or multiplication problems when given the commands “divide” or “multiply”.

- work out another problem you have chosen. This could be calculating an area, how much your weekly shopping costs, or anything else you like.

Copy your code to [pastebin](#) and share your experiences, successes, failures, and learning in the comments.

What computers do best

Computers excel at doing the same thing over and over again. They are always consistent and never get bored or tired. Computers love a repetitive task.

Fortunately, to get a computer to do the same thing over and over again, you don't have to write the same code over and over again:

```
left foot forward
right foot forward
left foot forward
right foot forward
left foot forward
```

Instead, you can loop the same piece of code over and over again:

```
run this 100 times:
    left foot forward
    right foot forward
```

When it reaches the last line of code in the loop, it will go back to the top and start again, for as many times as you've asked for.

Looping and running the same code over and over again is known as **iteration**.

In this week you will learn about:

- iterables
- lists
- for loops
- using counts

Iterables

You might be thinking ...

iter-what-ables?

That's OK. An **iterable** is just something which can be "iterated", or stepped through one bit at a time, such as a list of numbers or words. Before delving into the subject of iteration in Python, we will take some time to explore iterables and how they work.


Though you may not have realised it, you have already used an iterable during this course: a **string**.


Conceptually a string is a list of characters in an order: for example, "hello" is a list of 5 characters in the order "h" "e" "l" "l" "o".


Understanding how lists and iterables work is important because this is how Python iterates: by running the same code for each item in a list.


First let's explore the strings as iterables in the REPL to see how they work.


- Open the REPL (or console in Trinket).


 Mu 1.0.0.beta.17



Mode



New



Load



Save



Run


Debug


REPL


Plotter

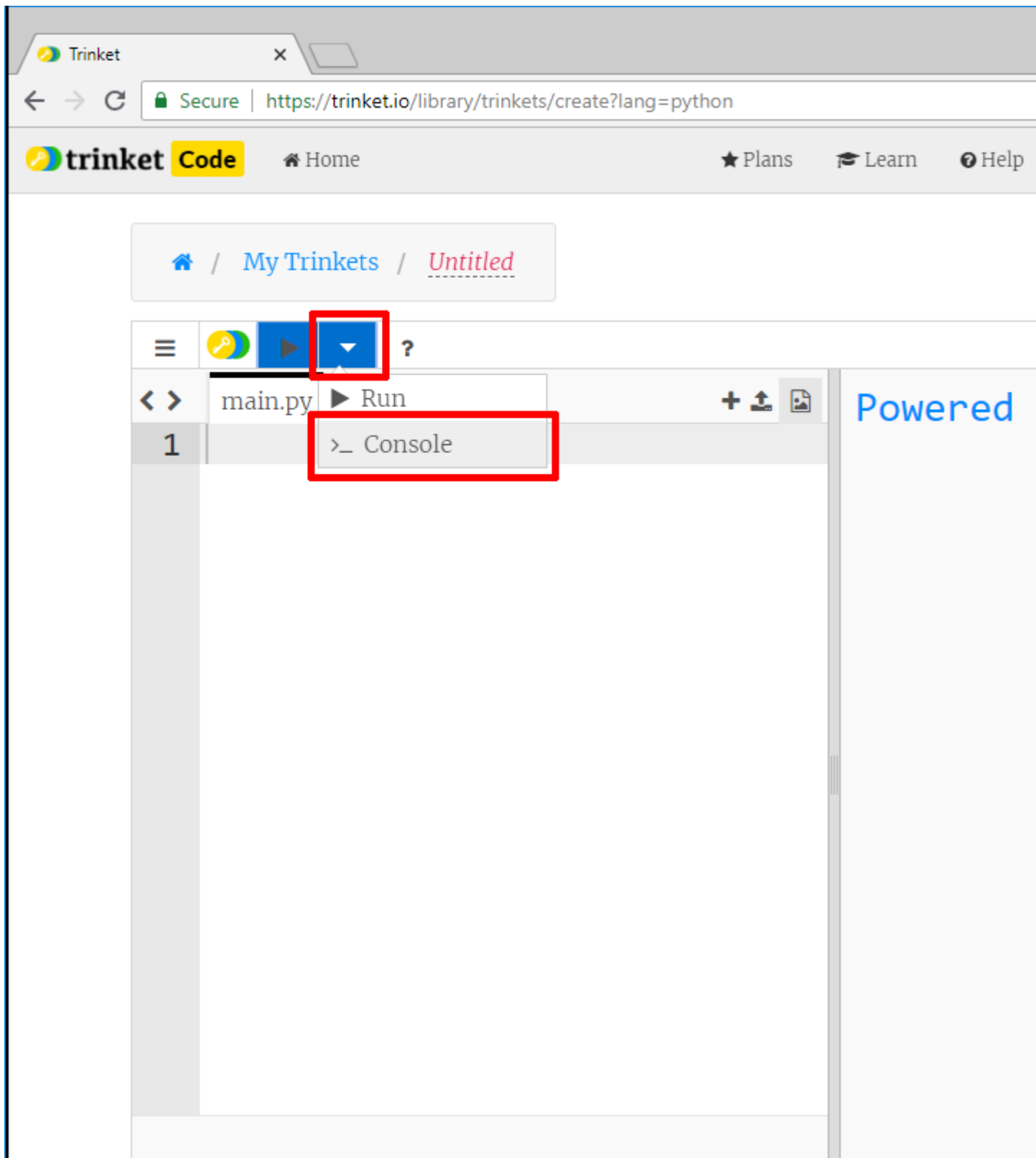

Zoom-in


Zoom-out

Python3 (Jupyter) REPL

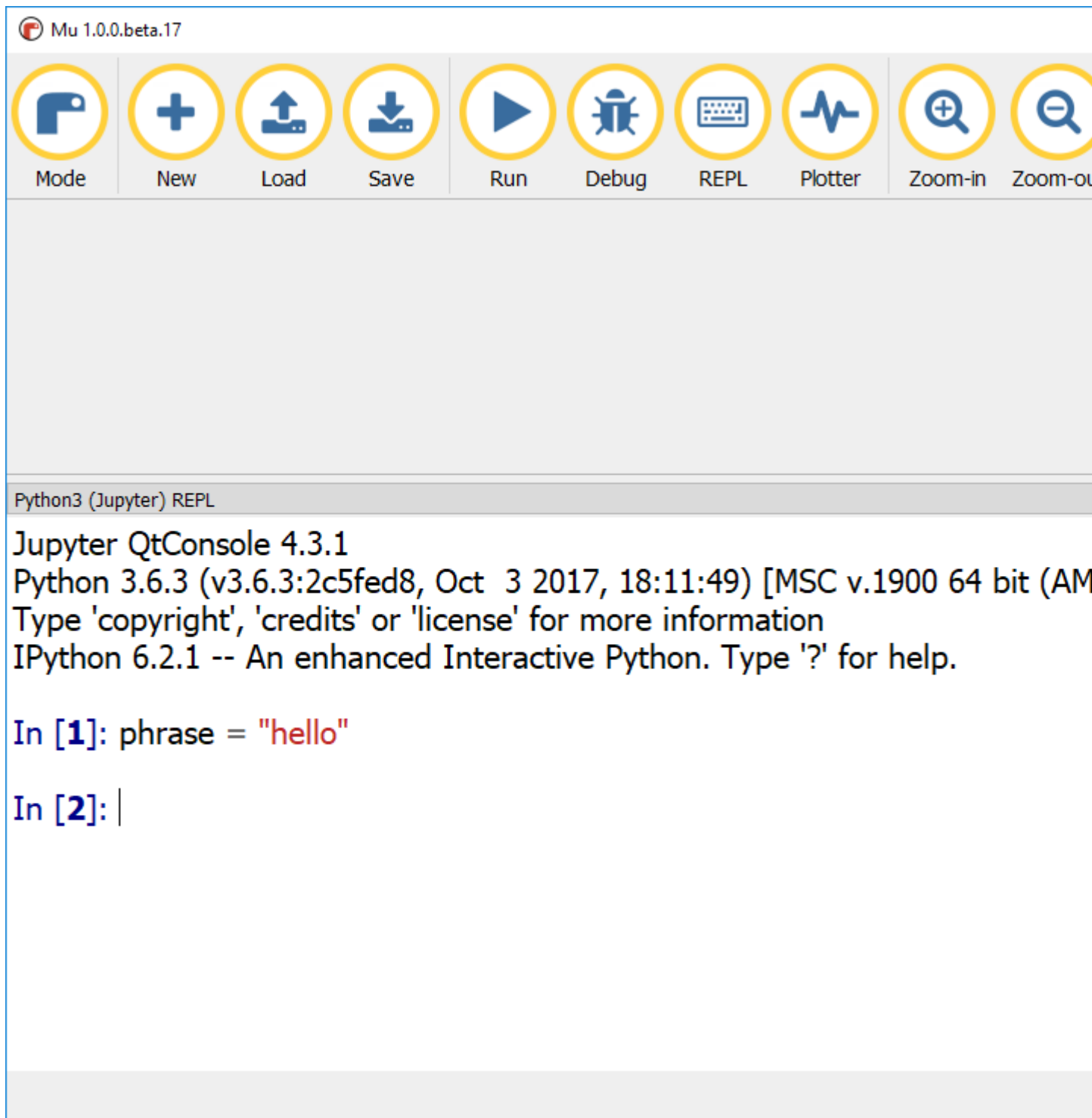
Jupyter QtConsole 4.3.1
Python 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017, 18:11:49) [MSC v.1900
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]:



- Create a string variable called `phrase` and assign it the value `"hello"`, by typing the following into the REPL and pressing Enter.

```
phrase = "hello"
```



You can access individual elements (or items) in a list or split them up into parts by using **indexes**, an index being a position in a list.

The string `hello` has 5 characters, the first element being `h` which has an index of `0`, the second element `e` has an index of `1` and so on:

```
0 1 2 3 4
h e l l o
```

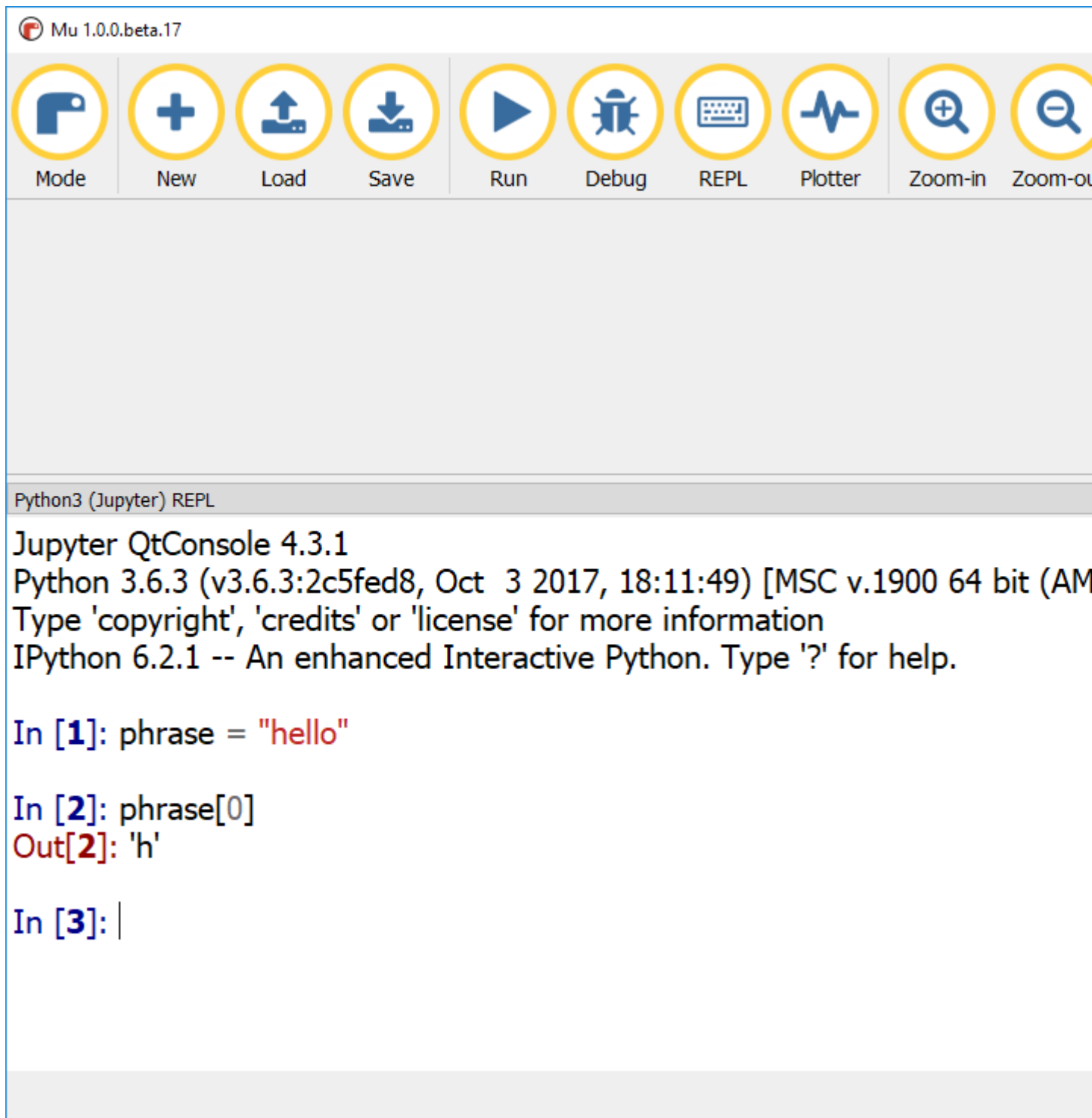
Tip: It's a controversial subject, but the majority of computer programmers agree that indexes should start at 0 and not 1, so it's a good idea to embrace the idea that numbers start at zero!

You access elements in a list by using square brackets [] after the variable name, adding the index of the element you wish to get:

```
variable[index]
```

- Get the first character from your string variable by entering this into the REPL.

```
phrase[0]
```



Challenge: repeat the step above, changing the index to get the fourth character ("l") in the string.

Indexes don't have to be positive numbers. You can also use negative indexes to get elements from the end of the list. E.g. the last element in a list would have the index of -1:

```
-5 -4 -3 -2 -1  
h e l l o
```

- Get the last character from your string using the REPL.

```
phrase[-1]
```

What do you think the advantage is of using `phrase[-1]` over using `phrase[4]` to get the last element?

Indexes and slices

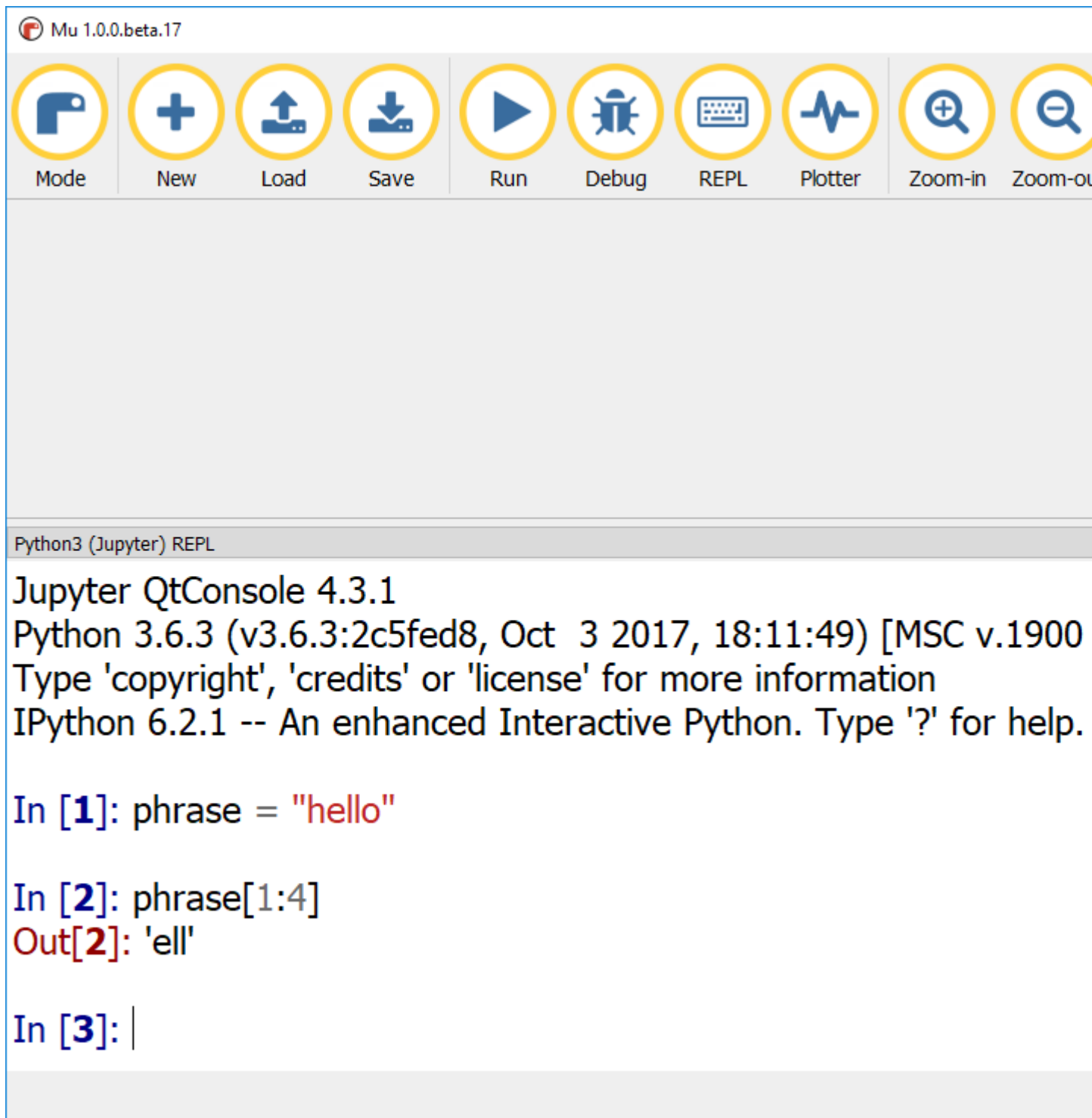
In addition to getting single elements from an iterable you can also use indexes to get “slices” of a list, such as the first three characters or the 3rd to 4th characters.

To get a slice, you need to provide two indexes (the start and end of the slice you want), separated by a colon `:` inside square brackets:

```
variable[start_index:end_index]
```

- Get the middle three characters from the `phrase` variable.

```
phrase[1:4]
```



Note: To get the middle 3 characters `ell` from `hello` you used `phrase[1:4]`. The element at index 4 is “o”, but “o” isn’t output. The end_index in a range is **up to, but not including**.

```
0 1 2 3 4
h e l l o
- - - >
```

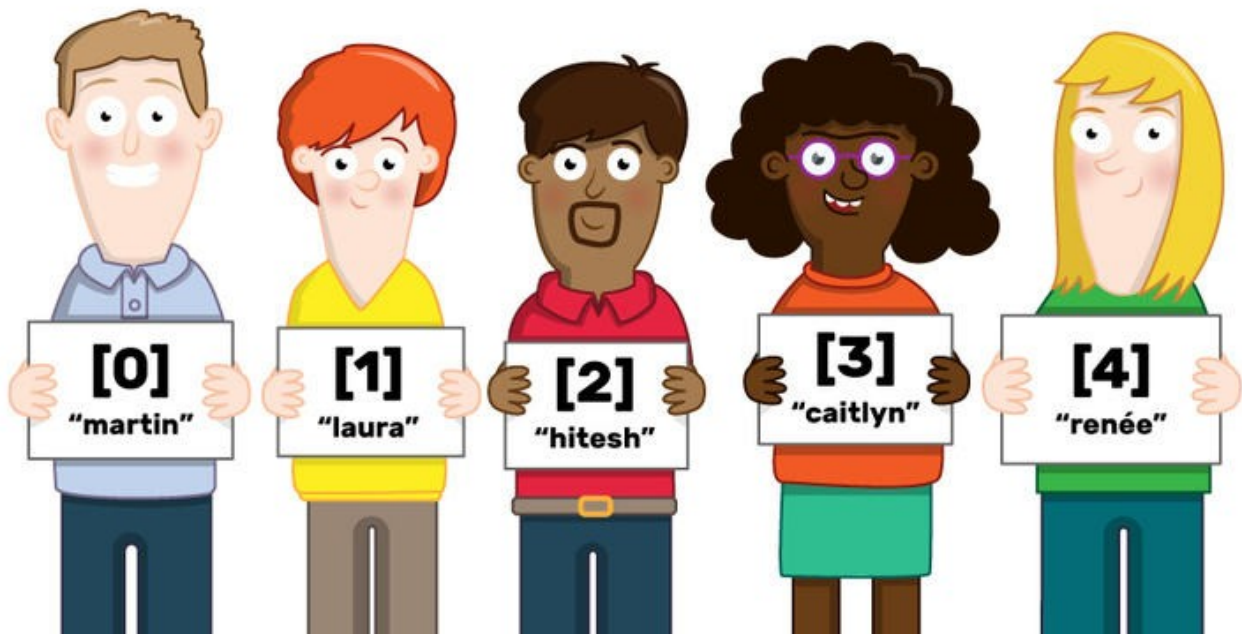
Challenge: Get the last two characters from the `phrase` variable.

Tip: If you want to get a range which is from the start or up to the end of a list, you can leave the start or end index blank. E.g.

```
phrase[:3]  
phrase[3:]
```

- Practice using ranges, create some longer strings and see how you can slice them up.

Lists



Python has a data type for storing lists, imaginatively called `List`. A list can store a sequence of any other pieces of data of any types (integers, strings, booleans, etc.).

Note: Other programming languages might refer to lists as **arrays**. While they have a different name and their usage might also be different, conceptually they are the same.

You can create a list variable in Python in just the same way as any other variable: by giving it a name and assigning it a value. For a list, the value will be a sequence of elements to be put in the list, written between square brackets `[]` and separated by commas:

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

If you want to create an empty list, which you will append items to later in your program, you can use empty square brackets: `[]`.

```
my_list = []
```

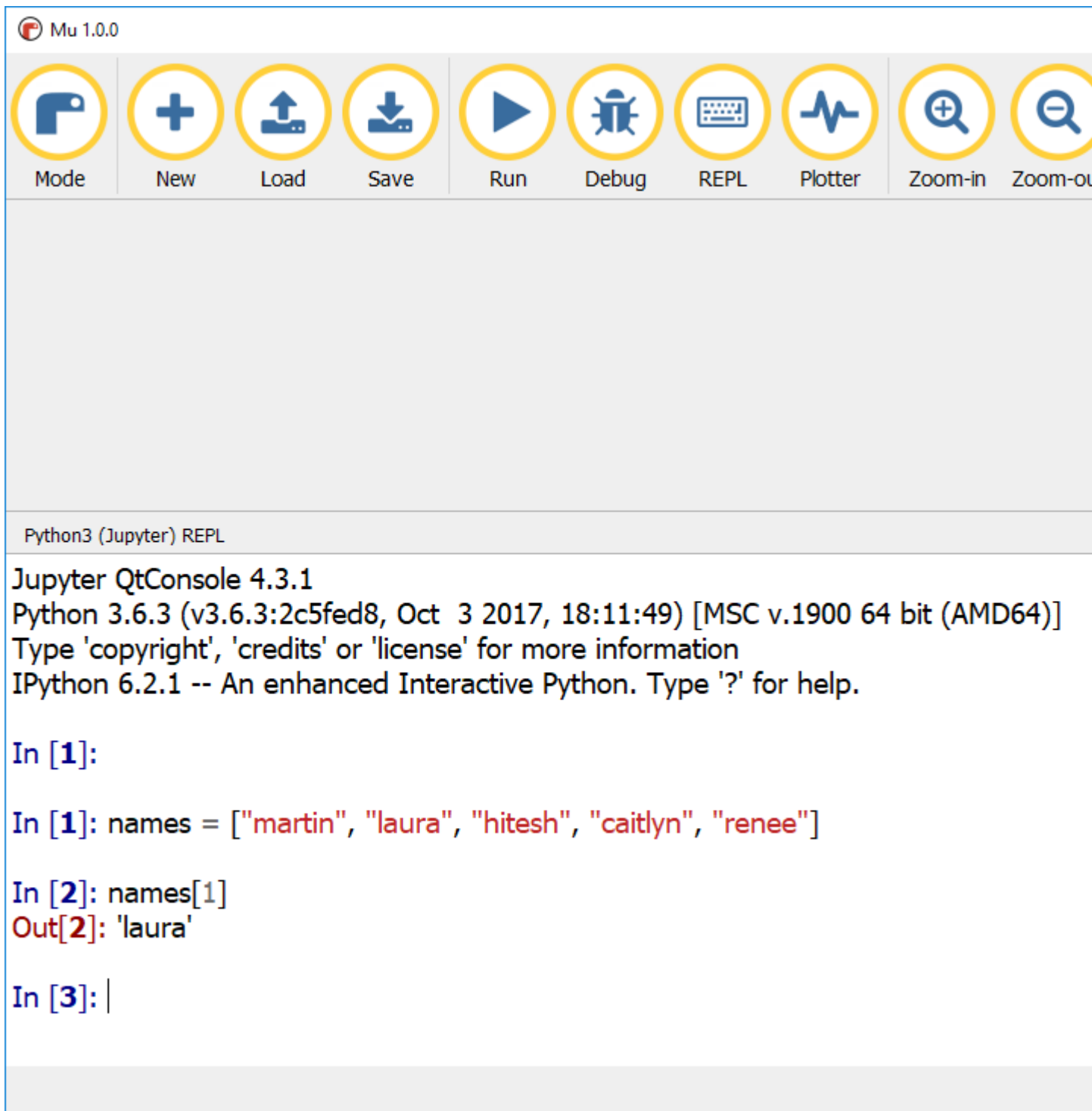
Note: Lists can contain all different types of data: integers, strings, and anything else. They can even contain other lists. (Lists of lists are very useful for storing tables of data, but that's a subject for another course!)

- Create a list of `names` using the REPL.

```
names = ["martin", "laura", "hitesh", "caitlyn", "renee"]
```

- Using an index, as you did with a string, get the second name (index 1) from the list.

```
names[1]
```



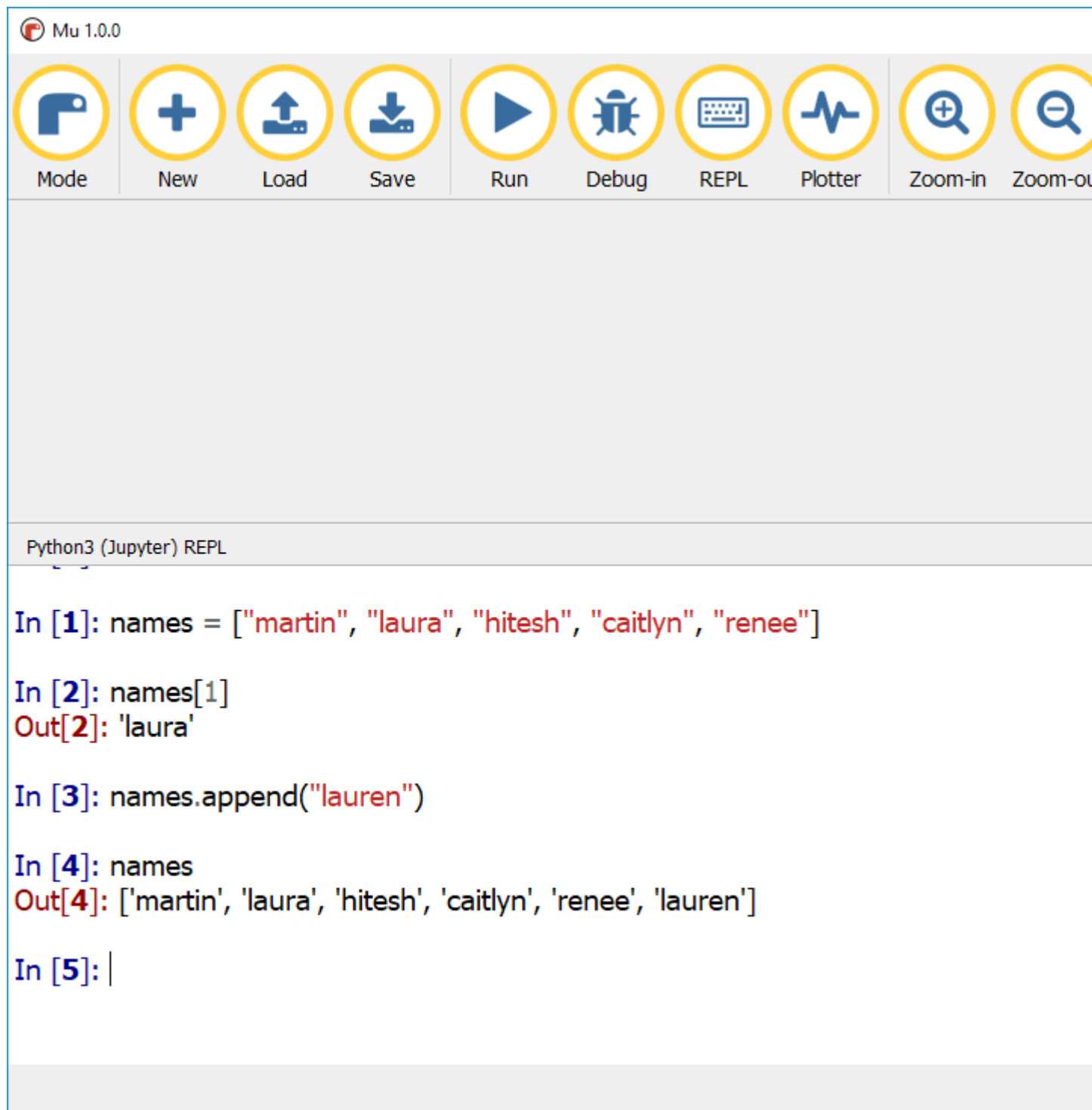
A feature of lists in Python is that they can be changed after they have been created: you can append new elements to the end of the list, insert them in the middle, or remove them completely using the three functions `append`, `insert` and `remove`. (As these are special list functions, they are joined with a dot to the name of the list they apply to, as in the examples below. Technically functions of this kind are called “methods”.)

- Append a new name to the end of your list of names.

```
names.append("lauren")
```

- Check to make sure your new name has been appended to the end of your list.

names



```
Mu 1.0.0

Mode New Load Save Run Debug REPL Plotter Zoom-in Zoom-out

Python3 (Jupyter) REPL

In [1]: names = ["martin", "laura", "hitesh", "caitlyn", "renee"]

In [2]: names[1]
Out[2]: 'laura'

In [3]: names.append("lauren")

In [4]: names
Out[4]: ['martin', 'laura', 'hitesh', 'caitlyn', 'renee', 'lauren']

In [5]: |
```

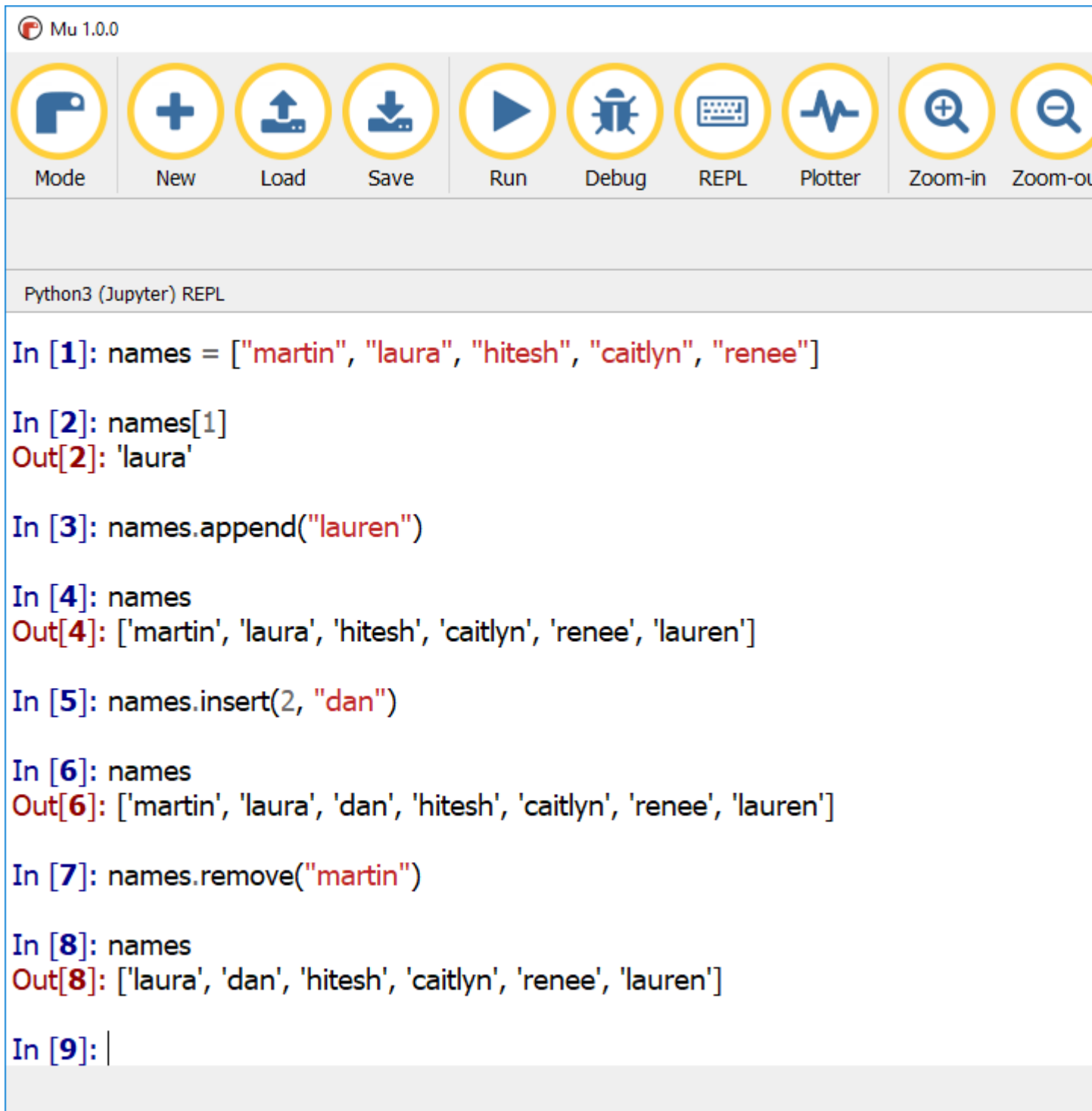
- Next let's use `insert` to put a new name in the middle of the list, by also using the index of where to insert the new name.

```
names.insert(2, "dan")
```

- Check to make sure that the names variable has been updated.

- As a last step let's remove the name "martin" from the list.

```
names.remove("martin")
```



The screenshot shows the Mu Python IDE interface. At the top is a toolbar with icons for Mode, New, Load, Save, Run, Debug, REPL, Plotter, Zoom-in, and Zoom-out. Below the toolbar is a tab labeled "Python3 (Jupyter) REPL". The main area displays a Jupyter-style REPL session with the following interactions:

```
In [1]: names = ["martin", "laura", "hitesh", "caitlyn", "renee"]

In [2]: names[1]
Out[2]: 'laura'

In [3]: names.append("lauren")

In [4]: names
Out[4]: ['martin', 'laura', 'hitesh', 'caitlyn', 'renee', 'lauren']

In [5]: names.insert(2, "dan")

In [6]: names
Out[6]: ['martin', 'laura', 'dan', 'hitesh', 'caitlyn', 'renee', 'lauren']

In [7]: names.remove("martin")

In [8]: names
Out[8]: ['laura', 'dan', 'hitesh', 'caitlyn', 'renee', 'lauren']

In [9]: |
```

Lists are really powerful and are the cornerstone of how iteration is done in Python, so it is worth exploring them to make sure you have a good understanding of how they work. Make sure that you can explain how the `append`, `insert` and `remove` commands work in your own words.

Next let's put all these ideas together and create a new program which uses lists to create a loop.

Looping through a list

You are going to use iteration and lists to create a program which will create and display a shopping list for you. You will create this program in two parts: one to displaying the shopping list, and another to create it. Your challenge will be to combine these two programs together.

We'll start with the code to display the shopping items. The first step is to create a program which will loop through all the items in a list and print them to the screen.

- Create a new program and save it as `display_shopping`.

Note: If you are unsure how to do this, take a look at the “If it’s this, then do that” step from Week 2.

- Create a list variable for your shopping and fill it with some items.

```
shopping = ["bread", "cheese", "apple", "tomato", "biscuits"]
```

To loop through each of the items in the list, you will use a `for` loop. `for` loops exist in most programming languages, and their purpose is to allow code to be run a set number of times – in our case, equal to the number of things in our shopping list.

The syntax of a `for` loop in Python looks like this:

```
for element in list:
    print("do this")
```

- Use a `for` loop to print out each of the items in your shopping list.

```
for item in shopping:
    print(item)
```

Note: The `print` statement is indented under the `for` loop. Indentation in a `for` loop works exactly the same as in an `if` statement; whatever code is indented under the `for` is run as part of that loop.

- Run your program.

```
Mu 1.0.0.beta.17 - display_shopping.py
```

Mode New Load Save Stop Debug REPL Plotter Zoom-in Zoom-out

display_shopping.py x

```
1 shopping = ["bread", "cheese", "apple", "tomato",
2 for item in shopping:
3     print(item)
4     |
```

Running: display_shopping.py

```
bread
cheese
apple
tomato
biscuits
>>>
```

Let's break this down and consider what steps the computer takes to carry out this program.

- First it creates the list called `shopping`.

```
shopping = ["bread", "cheese", "apple", "tomato", "biscuits"]
```

- The `for` loop is entered. The first element in the `shopping` list, `"bread"`, is put into the variable `item`.

```
for item in shopping:
```

- `print` is used to display what is in the variable `item`.

```
print(item)
```

- `item` holds the value “bread”, so “bread” is printed.

bread

- The `for` then runs again and the second element in the shopping list, "cheese" is put into the variable `item`.

```
for item in shopping:
```

- `print` is used to display what is in the the variable `item`.

```
print(item)
```

- This time, `item` holds the value “cheese”, so “cheese” is printed.

cheese

- The `for` loop runs again and the third element, "apple", is put into the variable `item`.

```
for item in shopping:
```

- This continues until there are no more items in the list. Each item has been printed in turn, and then the `for` loop stops.

Counting

A really useful coding technique is being able to **count** as your loop runs. That way you will know how many times your loop has run.

In your shopping list program, you can use this to print out how many items are in your list at the end of your program.

To keep track of how many items are in your list, you will need to create a variable which will hold a running total of the number of items.

- Create a variable called `count` at the top of the `display_shopping` program and set it to zero:

```
count = 0
```

- Indented under your `for` loop, add 1 to the `count` variable.

```
count = count + 1
```

It's worth looking at this line of code in a little more detail, as there is a new concept here.

Remember that `=` is an assignment: it asks the computer to assign a new value to the variable called `count`. When the computer run this line of code it does it in the following order:

1. `count + 1` is evaluated first: the computer calculates the current value of `count` plus 1.
2. Next the computer sets the value of `count` to the result of step 1.
3. `count` is now equal to 'whatever it was before', plus 1.

Conceptually this can be difficult to understand, but it is very common in computer programming.

Now let's print out how many items are in the shopping list.

- At the end of your program, **not** indented under the `for`, print the value of `count`.

```
print(count)
```

Your complete code should now look similar to this:

```
shopping = ["bread", "cheese", "apple", "tomato", "biscuits"]
count = 0

for item in shopping:
    print(item)
    count = count + 1

print(count)
```

- Run your program and check that the count is the same as the number of items in your shopping list.

```
Mu 1.0.0.rc.1 - display_shopping.py
```

Mode New Load Save Stop Debug REPL Plotter Zoom-in Zoom-out

display_shopping.py x

```
1 shopping = ["bread", "cheese", "apple", "tomato", "biscuits"]
2 count = 0
3
4 for item in shopping:
5     print(item)
6     count = count + 1
7
8 print(count)
```

Running: display_shopping.py

```
bread
cheese
apple
tomato
biscuits
5
>>> |
```

Reflect: In what other situations would it be useful to keep a running total?

- Improve the message so that it prints “you have X items in your shopping list”, inserting the value from the `count` variable.

Tip: Counting is a widely-used and really useful technique. However, in this instance, there is an easier way to find out how many items are in a list, using the function `len`, which is short for “length”. E.g.

`len(shopping)`

Case studies: "unplugged" activities

Non-computer-based or “unplugged” activities can help students understand a concept before they touch a computer. Here are some of our educators’ ideas for using “unplugged” activities.

- Secondary school teacher Jo Wakefield uses props and her acting skills:

” We use unplugged activities all the time. We “act” out the code using bits of paper/simple props/me making a fool of myself (again) so that students understand the concept first.

To give you an example – to teach sequence, I show a short animation in Scratch where two sprites (characters) tell each other a joke. I explain that that each sprite is following a sequence of instructions and use two students to “step-through” the program, including the “wait” command. To start with, the students often try and hold up their pieces of paper together (I write each stage of the joke on a piece of paper) and we quickly realise that this sequence doesn’t work.

It is not just programming concepts that I use “unplugged” activities to explain. We also look at packets travelling across a network by getting students to draw a picture, rip it up into four pieces, label each piece and then “send” it across the network to its desired location (another student) who then puts it back together again.

I just find that if students understand the concept “simply” – without the “distraction” of getting the coding right – they can apply their knowledge a lot quicker. It also seems to help them break down problems once we get to bigger tasks – they can identify that this bit is sequence, this is a loop, this is selection, etc.”

Jo Wakefield, secondary school teacher

- Former school teacher Laurel Bleich uses PE sessions, particularly running a timed mile around the school’s running track:

“As far as the loop goes, I mainly stress the counting of laps around the track. Inside the body of the loop, I usually print out how many laps have been run, or are to go, depending if we are counting up or down. I usually draw a track on the board and model the running around the track, so you could add the different turns in the body of the loop since that is something that you do every time. From my experience, the students understand that the set of instructions of the `while` loop gets executed a certain amount of times: it is how to set up what to initialise the counting variable to, and what conditional to set up, which seems to be the issue. They either run the loop one too many times or not enough or get an infinite loop. So I use this example to have them change the initial value of the counting variable and relate it to how they can think about counting laps when running their mile.”

Laurel Bleich, former school teacher

- Primary school teacher Beverley McCormick also uses PE activities to help her students understand some of the language used in computing.:

“**Algorithms:** have students give step by step instructions on how to play the game (rules and game play).

Repetition: saying the steps individually and then having students tell me how to code the sequence in a more efficient way.

Sequence works well in gymnastics or circuits, to describe which area the pupils should start and progress to and how they know to move on (eg when the whistle blows move to next circuit). Another example: ‘throw the balls until no more balls are in the box’.”

Beverley McCormick, Primary school teacher

Looping a set number of times

Being able to run the same piece of code a number of times can be really useful. Say you want to capture 11 names for a football team, or the lengths of the sides in a triangle. You can do this by adjusting the way you use a **for** loop.

To do something five times, you could create a **for** loop which looped for all the numbers between 0 and 4 by creating a list of those numbers e.g.

```
numbers = [0, 1, 2, 3, 4]
for number in numbers:
    print(number)
```

However this is very restrictive. It will only ever be able to loop five times, and what if you want to do something a million times? It wouldn't be feasible to create a list with every number up to 1,000,000.

To overcome this issue, Python has a function called **range** which will create a list of numbers for you.

By using **range** we can simplify the example program above:

```
for number in range(5):
    print(number)
```

Note: - **range(5)** will create a list of five numbers from **0 to 4**, not six numbers from 0 to 5.

You will now use a **for** loop with **range**, to create a program which will loop for a set number of times asking the user for an item of shopping and then appending it to the **shopping** list variable.

- Create a new program and save it as **create_shopping**.
- Create an empty list variable called **shopping**.

```
shopping = []
```

- Add a **for** loop with a **range** to loop 5 times.

```
for item_number in range(5):
```

- Indented under the for loop, use `input` to ask the user for something to go on the shopping list:

```
item = input("what is the item? ")
```

- Still indented under the for loop, append the item to the shopping list.

```
shopping.append(item)
```

- Finally after the for loop let's print the shopping list, so we can see what in it.

```
print(shopping)
```

The complete `create_shopping` program should now look similar to this:

```
shopping = []
```

```
for item_number in range(5):  
    item = input("what is the item ? ")  
    shopping.append(item)
```

```
print(shopping)
```

- Run your program and make sure you can enter 5 items of shopping and that they are displayed at the end.

The screenshot shows the Mu Python IDE interface. At the top, the title bar reads "Mu 1.0.0.rc.1 - create_shopping.py". Below the title bar is a toolbar with icons for Mode, New, Load, Save, Stop, Debug, REPL, Plotter, Zoom-in, and Zoom-out. The main editor area displays a Python script named `create_shopping.py` with the following code:

```
1 shopping = []
2
3 for item_number in range(5):
4     item = input("what is the item ? ")
5     shopping.append(item)
6
7 print(shopping)
8
```

Below the editor, a console window titled "Running: create_shopping.py" shows the program's execution. It displays five prompts: "what is the item ? raspberry pi", "what is the item ? power supply", "what is the item ? sd card", "what is the item ? keyboard", and "what is the item ? mouse". The final output is a list: `['raspberry pi', 'power supply', 'sd card', 'keyboard', 'mouse']`. The prompt `>>> |` is visible at the bottom of the console.

At each stage, the variable `item_number` will contain the number of the times the program has been around the `for` loop, starting at 0 and going up to 4. Let's make this more apparent.

- Change your program to include each `item_number` in the corresponding prompts to the user for shopping items.

```
item = input("what is item number " + str(item_number) + "? ")
```

- Run your program so you can see this displaying each `item_number`.

```
Mu 1.0.0.rc.1 - create_shopping.py

Mode New Load Save Stop Debug REPL Plotter Zoom-in Zoom-out

create_shopping.py x
1 shopping = []
2
3 for item_number in range(5):
4     item = input("what is item number " + str(item_number))
5     shopping.append(item)
6
7 print(shopping)
8

Running: create_shopping.py

what is item number 0? coffee
what is item number 1? milk
what is item number 2? sugar
what is item number 3? sprinkles
what is item number 4? cup
['coffee', 'milk', 'sugar', 'sprinkles', 'cup']
>>> |
```

Reflect: Make sure that you can explain in your own words what each line of code does here. You may want to work through what's going on in the program on paper, making sure to increase 'item_number' for each iteration of the for loop.

At the moment your program will only ever collect 5 items of shopping. What if you have more or less? In the next step you will change the program so that it starts by asking the user how many items of shopping they need, and use that number in your for loop.

- Add to the top of your program an `input` to get the number of items of shopping.

```
how_many = input("how many items of shopping do you have? ")
```

- The variable `how_many` will be a string, so use `int` to cast it to an integer.

```
how_many = int(how_many)
```

Note: rather than create a new variable to hold the `how_many` integer, called e.g. `how_many_number`, you have cast the `how_many` variable back to itself. You no longer need the string variable `how_many` so you may as well “overwrite” it with an integer variable!

- Now change the `for` loop so that the variable `how_many` is used by `range`.

```
for item_number in range(how_many):
```

Your program should now look similar to this:

```
shopping = []
```

```
how_many = input("how many items of shopping do you have? ")  
how_many = int(how_many)
```

```
for item_number in range(how_many):  
    item = input("what is item number " + str(item_number) + "? ")  
    shopping.append(item)
```

```
print(shopping)
```

- Run your program and enter a number of items.

The screenshot shows the Mu Python IDE interface. At the top, the title bar reads "Mu 1.0.0.rc.1 - create_shopping.py". Below the title bar is a toolbar with icons for Mode, New, Load, Save, Stop, Debug, REPL, Plotter, Zoom-in, and Zoom-out. The main editor area displays a Python script named "create_shopping.py" with the following code:

```
1 shopping = []
2
3 how_many = input("how many items of shopping do you have? ")
4 how_many = int(how_many)
5
6 for item_number in range(how_many):
7     item = input("what is item number " + str(item_number)
8     shopping.append(item)
9
10 print(shopping)
```

Below the editor, a status bar indicates "Running: create_shopping.py". The output area shows the execution results:

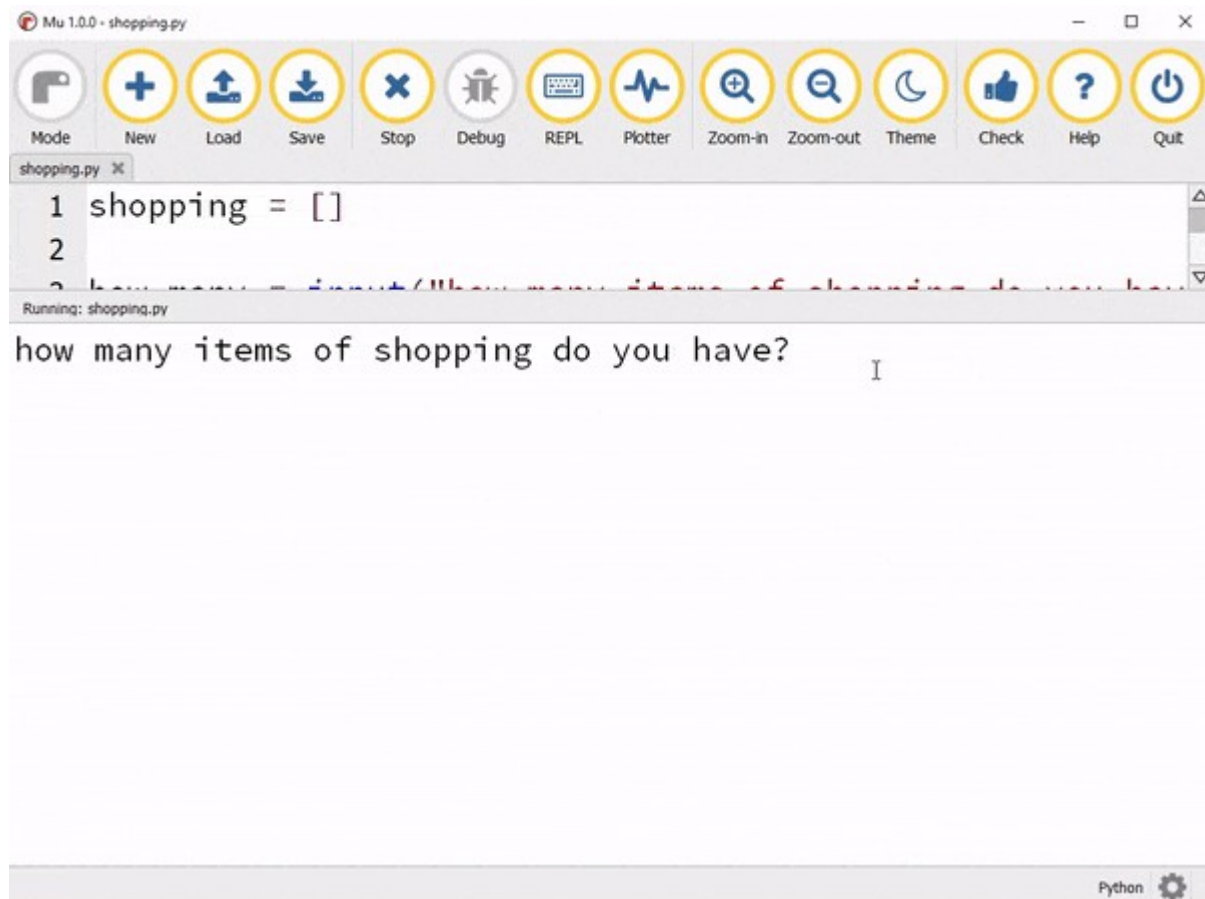
```
how many items of shopping do you have? 3
what is item number 0? tea
what is item number 1? lemon
what is item number 2? cup
['tea', 'lemon', 'cup']
>>> |
```

You have now learnt how to use a `for` loop. Next week we will also look at other forms of looping.

Challenge: combine your shopping programs

Now you have two parts of a program which will create a shopping list and display it. Your challenge is to combine them together to create a program which will:

- Ask how many items of shopping are needed.
- Collect the items and store them in a list.
- Loop through the shopping list and display them.
- Show a total at the end so the user can see how many have been added.



Share your successes and reach out for help if you need it.

Case studies: reading and interpreting code

A key step in being able to write code is being able to read and interpret code that's already been written. Here are some ways our educators help their students develop this skill.

- Starting with the programming language Scratch is a common method:

“We look at games in Scratch and decompose them to work out the structures that are involved in creating the games (variables, loops, selection, etc). We will often use unplugged activities to demonstrate the concept. Once they're happy that they understand the principle, we look at the “language” involved in Python. I stress that it is more important to understand the “concept” before the language. A language can be learnt/looked up - but understanding the concept can't.”

Jo Wakefield, secondary school teacher

- Dave Hartley uses another approach:

“I use the PRIMM approach championed by Sue Sentence and others. Predict, then run.”

Dave Hartley

- PRIMM is an approach to planning programming lessons and activities and includes the following stages:
 - Predict
 - Run
 - Investigate
 - Modify
 - Make

These stages are used in planning lessons and activities and are designed to support learners at all stages of learning programming in school, not just complete beginners.

- To find out more, head to <https://primming.wordpress.com/>

How will you develop your student's code-reading skills?

Calculating averages

Now you have learnt how to use loops, you will use these new skills to update your bot so it calculate an average (or mean).

To do this, your bot will need to:

1. Capture how many numbers you wish to average
2. Loop that many times, capturing the numbers
3. Keep a running total of the numbers
4. When the loop has finished divide the total by the number of numbers

- Open your bot program.
- Add a new command to your bot by adding a new `elif` to test for a new “average” command.

```
elif command == "average":
```

Note: the `elif` should be indented inline with the `if` and other `elif`s.

- Indented under the new `elif`, use `input` to capture how many numbers are to be averaged.

```
how_many = input("How many numbers> ")
```

- The variable `how_many` is a string and will need to be cast to an `integer`.

```
how_many = int(how_many)
```

- Create a variable called `total` which will be used to keep a total of the numbers entered, and set it to 0.

```
total = 0
```

- Using `for` and `range`, loop as many times as there are numbers to get.

```
for number_count in range(how_many):
```

- Capture the numbers from the user using `input` and add the value to the total variable.

```
    number = input("Enter number " + str(number_count) + "> ")
    total = total + int(number)
```

Note: these lines of code are indented two levels, once under the `elif` and again under the `for`.

- After the `for` loop you need to add the code to calculate the result and print the result.


```
result = total / how_many
print("the average = " + str(result))
```


Note: Make sure your indents are correct, as there are multiple levels of indentation in this program. Your code should be structured similar to this:


```
elif command == "average":
    how_many = input("How many numbers> ")
    how_many = int(how_many)
    total = 0
    for number_count in range(how_many):
        number = input("Enter number " + str(number_count) + "> ")
        total = total + int(number)
    result = total / how_many
    print("the average = " + str(result))
```


- Run your new bot program and calculate the average from a list of numbers.


Mu 1.0.0 - bot.py


Mode


New


Load


Save


Stop

Debug

REPL

Plotter

Zoom-in

Zoom-out

bot.py x

```
20     print(input1 + " - " + input2 + " = " + output)
21 elif command == "average":
22     how_many = input("How many numbers>")
23     how_many = int(how_many)
24     total = 0
25     for number_count in range(how_many):
26         number = input("Enter number " + str(number_count) + ">")
27         total = total + int(number)
28     result = total / how_many
29     print("the average = " + str(result))
30 else:
31     print("sorry I dont understand")
32
```

Running: bot.py

Hi, I am Marvin, your personal bot.
How can I help?average
How many numbers>5
Enter number 0>10
Enter number 1>20
Enter number 2>30
Enter number 3>40
Enter number 4>50
the average = 30.0
>>>

Challenge: add a new command

It's now your turn to add a new command to your bot. It should be something which you would find helpful.

Here's a few ideas to help get you started:

- include the shopping list creator into your bot
- add up the cost of your shopping
- calculate a discount
- divide the cost of a meal in a restaurant

Discuss your ideas and share your code

Recap of Week 3

This week you learnt about iterables and how to use them in Python to make your computer run the same code many times.

You used `for` loops to run through the items in a list, and learnt how to use `range` to create a list of numbers.

You also learnt about the following:

- **iteration:** repeating the same action more than once
- **counting:** how to count by incrementing the value of a variable
- **lists:** storing a collection of data together in a sequence
- **indexes:** how to access data in an iterable

Iteration is a key programming concept and one which makes computers into really powerful tools for automation.

Next week you will explore other forms of looping, and learn how to create re-usable code which can be used for many different scenarios.

Glossary

- **iterable:** data which can be iterated or “stepped over”
- **iteration:** repeating the same action more than once
- **list:** a data type which can hold many items of data in order
- **index:** the location of an item in a list
- **for:** a type of loop used to iterate through the items in a list
- **condition:** an expression which will be evaluated as either true or false
- **evaluation:** determining the value of an expression, e.g. `True` or `False`
- **selection:** using **conditions** to make choices of what to do next
- **if, then, else:** tool for making code run depending whether a statement is true or false

- **and, or**: logical operators which can combine two conditions
- **scope**: defining which code belongs to which part of the program
- **indentation**: adding spaces to the start of lines of code in Python so that it understand scope
- **sequence**: describing code running one line at a time in order
- **function**: a self-contained instruction which performs a specific task
- **parameter**: data passed to a function
- **code**: the instructions that make up a program
- **variable**: a stored value in a program which is referenced by a name
- **data type**: the type of data stored in a variable (e.g. string, integer)
- **string**: a text based value
- **integer**: a whole number
- **bug**: a problem within a program
- **debugging**: the process of finding and resolving bugs
- **IDE**: Integrated Development Environment
- **concatenation**: adding pieces of text together
- **syntax**: the correct structure or layout of code

Conditional loops

Last week you learnt how to make computers do the same thing a number of times using a `for` loop. But what if, when you start your loop, you don't know how many times the computer will need to do it? Or what if you want to do something forever?

In such cases, you can use a different type of loop which will keep doing the same action until told to stop. This is known as a **conditional** loop or `while` loop.

```
while the music is playing:
    step right
    step left
```

At the start of the loop the computer will evaluate the condition (e.g. is this statement **true**), if so it will run the code inside the loop. The next time around the loop it will evaluate the condition again - if it's still true it will run the code again, if its not true it will exit the loop and carry on running the rest of the program.

If the music never stopped, the robot above would dance forever.

Reflect: What do you think would happen if the music stopped playing after the robot had stepped right, but before it had stepped left?

While loops

Python's uses a "`while`" loop to do conditional loops, and it works by following this process:

while this *statement* is true, I will continue to run this code, *until* it is false.

The syntax for a while loop in Python looks like this:

```
while statement == True:
    print("do this")
```

The syntax is very similar to an `if`: between the `while` and the `:` is the condition to be evaluated, and the code indented under it is the code to run as part of the loop.

Next you will create a small program which uses a `while` loop to continually ask the user to “guess my name” until they get it correct.

- Create a new program and save it as “while”.
- Use `input` to ask for a guess:

```
guess = input("guess my name ")
```

- Now use a `while` loop which will run as long as the value in `guess` is not equal to your name.

```
while guess != "Martin":
```

- Indented under the `while`, let the user know they got it wrong and tell them to “guess again”, updating the value of the `guess` variable.

```
    guess = input("wrong - guess again ")
```

- Finally at the end of your program and **not indented** under the `while` loop let the user know they got it correct.

```
print("well done")
```

Your program should look similar to this.

```
guess = input("guess my name ")
while guess != "Martin":
    guess = input("wrong - guess again ")
print("well done")
```

- Run your program and test the following scenarios:
 1. guess the name correctly the first time.
 2. get the name incorrectly once.
 3. get the name incorrectly many times.


```
Mu 1.0.0 - while.py
```

Mode New Load Save Stop Debug REPL Plotter Zoom-in Zoom-out

while.py

```
1 guess = input("guess my name ")
2 while guess != "Martin":
3     guess = input("wrong - guess again ")
4 print("well done")
5
```

Running: while.py

```
guess my name Rik
wrong - guess again Martin
well done
>>> |
```

The following logic is applied when running this code:

- When the name is guessed correctly the first time, the code inside the `while` loop is never run, because `guess != "Martin"` is false.
- If the name is guessed incorrectly, the code inside the `while` loop is run and the variable `guess` is updated with the user's next guess.
- The `while` loop's statement is then evaluated again. If now `guess != "Martin"` is true (i.e. the guess is still not 'Martin'), then the code inside the loop will run again.
- This continues until `guess != "Martin"` is false, i.e. until the user has finally guessed Martin. At this point, the program continues from after the `while` loop, and "well done" is printed.

Challenge

Change the program so that it keeps a count of the number of times it took you to guess the name, and then outputs the total when you guess correctly.

Time for reflection

It's important to take some time out to reflect on what you've learnt so far.

People who take time to reflect on their learning are much more likely to make use of it later. And you will help us make our courses even better by sharing your reflections with us. With that in mind, we'd like you to fill in our [five-question Reflection Form](#). It will help you connect the dots between all the new knowledge you've taken in, and get your synapses firing to find new ways to apply your learning.

Do this forever

Using a `while` loop you can get a computer to continue to do the same task forever, known as an infinite loop. This special type of conditional loop can be really useful, but it's also a really easy way to use up all of your computer's processing power!

- Create a new program, add this code, save it as `infinite`, and run it.

```
while True:
    input("press enter")
```

It doesn't matter how many times you press the enter key, the program will keep running forever. It will never leave the `while` loop – it is infinite.

Tip: If you want to end an infinite program click the **Stop** button on Mu or Trinket.

`while True:` is the “usual” way of creating an infinite loop in Python and although it looks a bit strange, it makes sense when you understand what's happening.

First consider this program:

```
keep_running = True
while keep_running == True:
    input("press enter")
```

The variable `keep_running` will always be equal to `True` and as it is never changed to `False` this program will run forever and is infinite.

If `keep_running` is always `True` the same program could be written as follows replacing the `keep_running` variable with `True`:

```
while True == True:
    input("press enter")
```

As you learnt in week 2, if you were to evaluate `True == True` it will always be equal to `True`, so if you replace `True == True` with `True` and you end up with:

```
while True:
    input("press enter")
```

Reflect: In what sort of situations do you think an infinite loop might be useful? Give an example in the comments.

Loop your bot

You can use a conditional loop in your bot program so that it continues to ask “How can I help?” until you give it the command “bye”.

- Open your `bot` program.
- At the top of your program create a variable called `finished` and set it to `False`.

```
finished = False
```

This variable will be set to `True` when the user enters the “bye” command.

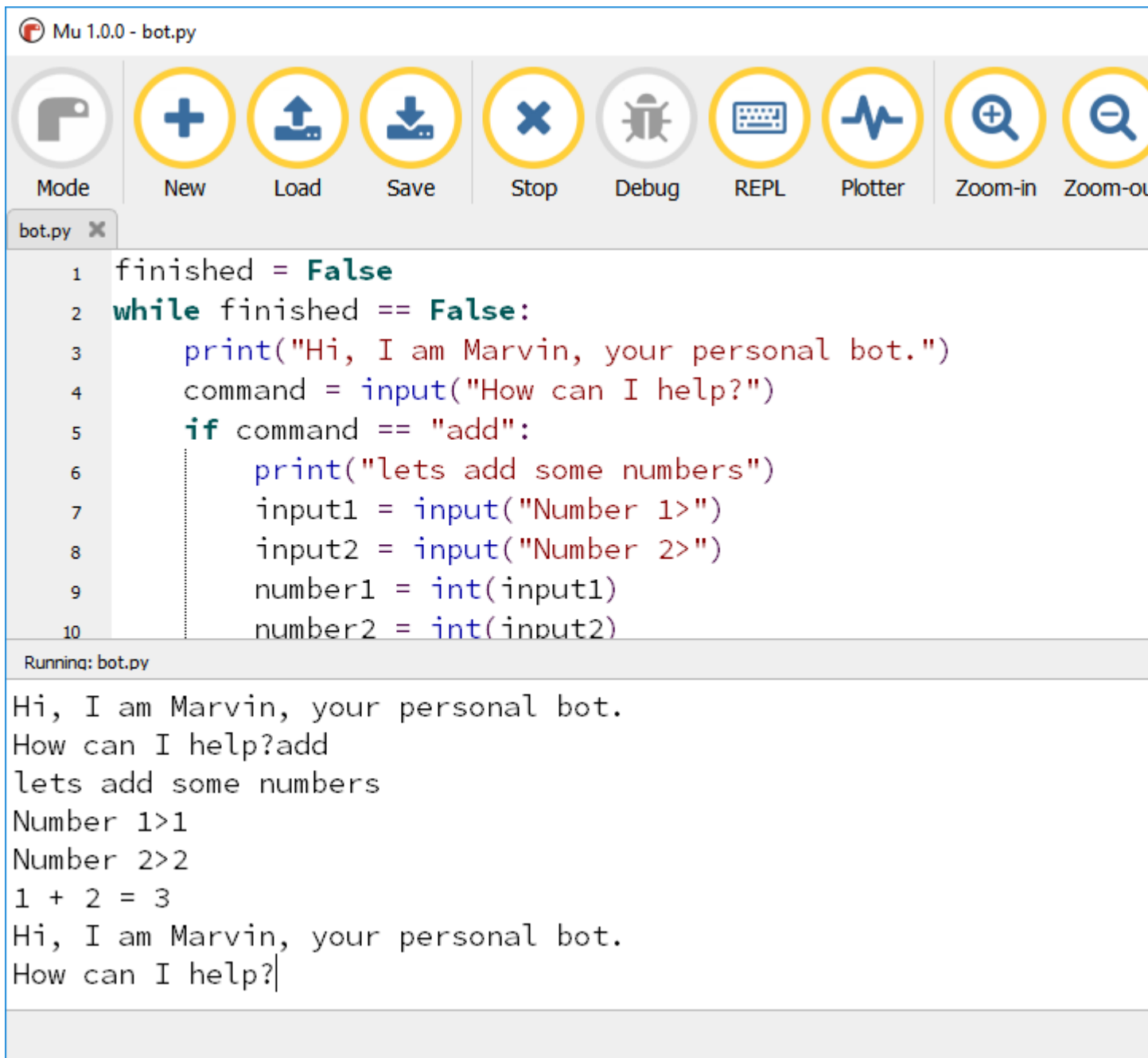
- Under this use a `while` loop to keep your program running as long as `finished` is equal to `False`.

```
while finished == False:
```

- The whole of your program should now be indented under the `while` loop.

Tip: you can indent multiple lines of code by selecting them in a block and pressing `TAB`.

- Now run your program.



The screenshot shows the Mu Python IDE interface. At the top is a toolbar with icons for Mode, New, Load, Save, Stop, Debug, REPL, Plotter, Zoom-in, and Zoom-out. Below the toolbar is a tab labeled 'bot.py'. The main editor area contains the following Python code:

```
1 finished = False
2 while finished == False:
3     print("Hi, I am Marvin, your personal bot.")
4     command = input("How can I help?")
5     if command == "add":
6         print("lets add some numbers")
7         input1 = input("Number 1>")
8         input2 = input("Number 2>")
9         number1 = int(input1)
10        number2 = int(input2)
```

Below the editor is a console area labeled 'Running: bot.py'. It shows the output of the program:

```
Hi, I am Marvin, your personal bot.
How can I help?add
lets add some numbers
Number 1>1
Number 2>2
1 + 2 = 3
Hi, I am Marvin, your personal bot.
How can I help?
```

Your bot will now continue to ask “How can I help you” forever as the variable `finished` is always set to `False`. You should change your bot so that you can give it the command “bye” and the program will finish.

- Using an `elif` add a new “bye” command, which sets `finished` to `True`.

```
elif command == "bye":
    finished = True
```

- Run your program and test that it continues to run until you enter the command “bye”.

Mu 1.0.0 - bot.py

Mode New Load Save Stop Debug REPL Plotter Zoom-in Zoom-out

bot.py

```
28         number = input("Enter number " + str(number_count))
29         total = total + int(number)
30     result = total / how_many
31     print("the average = " + str(result))
32 elif command == "bye":
33     finished = True
34 else:
35     print("sorry I dont understand")
```

Running: bot.py

Hi, I am Marvin, your personal bot.
How can I help?add
lets add some numbers
Number 1>1
Number 2>2
1 + 2 = 3
Hi, I am Marvin, your personal bot.
How can I help?bye
>>>

An examples

```
import random
```

```
def mathquestion():
    first = random.randint(1,10)
    second = random.randint(1,10)
    questiontype = random.randint(1,4)
    if questiontype == 1:
        answer = int(input("What is "+str(first)+" + "+str(second)+"? " ))
        if answer == first+second:
            print("Correct!")
        else:
            print("Sorry, the answer is: "+str(first+second)+"!")
    elif questiontype == 2:
        answer = int(input("What is "+str(first+second)+" - "+str(second)+"? " ))
        if answer == first:
            print("Correct!")
        else:
            print("Sorry, the answer is: "+str(first)+"!")
    if questiontype == 3:
        answer = int(input("What is "+str(first)+" x "+str(second)+"? " ))
        if answer == first*second:
            print("Correct!")
        else:
            print("Sorry, the answer is: "+str(first*second)+"!")
    elif questiontype == 4:
        answer = int(input("What is "+str(first*second)+" / "+str(second)+"? " ))
        if answer == first:
            print("Correct!")
        else:
            print("Sorry, the answer is: "+str(first)+"!")
```

```
stopwords = ["quit","end","bye","goodbye","see ya"]
finished = False
print("Hi, I'm HomeWorkBot!")
while not finished:
    subject = input("What would you like to talk about? ")
    if subject in stopwords:
        finished = True
    else:
        if subject!="maths":
            print ("That\'s boring.")
            print("Wouldn\'t you like to talk about maths instead?")
```

```
else:
    print("Great!")
    doingmaths = True
    while doingmaths:
        mathquestion()
        another=input("Would you like another question? ")
        if another.lower()[0:1]!="y":
            doingmaths = False
    print("OK!")
print("Bye - talk to you again soon!")
```

Which loop?

What is abstraction?

Abstraction is the act of removing complexity from a problem by hiding background details and information.

Let's imagine using abstraction to make our Robot wave.

To make the Robot wave you could give it the following instructions:

```
lift up your arm
open your hand
move your arm left
move your arm right
move your arm left
move your arm right
close your hand
lower your arm
```

That would be a lot of instructions to give each time we wanted to make our robot "wave". It would be much simpler if we could teach our robot how to wave once, and each time after that we could simply give it the instruction **wave**.

```
how to wave:
    lift up your arm
    open your hand
```

```
move your arm left
move your arm right
move your arm left
move your arm right
close your hand
lower your arm
```

```
wave
wave
wave
```

Abstraction is a fundamental concept of computing (not just programming): it's how technology has been able to progress at ever increasing speeds by continually abstracting problems, so that the next person who faces the same problem doesn't have to be concerned about the detail.

You are going to create functions in Python to simplify and reuse code in your programs.

Functions

You have already used some functions in Python, such as `print` and `input`. While they are “in-built” and you can't see their code, they were created in a very similar way to the programs you have written. In this next section you will create your own functions.

You may hear functions referred to as subroutines or procedures. In some other programming languages, subroutines and functions are different things, but in Python they are all just known as **functions**.

The syntax to create a function in Python looks like this:

```
def my_function():
    print("do this")
```

`def` stands for “definition”, since what you are doing when creating a function is defining it and giving it a name.

Between the `def` and the `()` is the name of your function followed by `:`. Indented under the `:` is the code which should be run as part of the function.

When you want to use (or **call**) a function you use the name with parentheses:

```
my_function()
```

Tip: Like variables, functions can be given any name you wish, but it is a good idea to name your functions after the job they do.

Let's explore how to use functions in Python.

- Create a new program called `functions`.
- Use `def` to add a function at the top of your program called `intro`.

```
def intro():
```

- Indented under this line, add some code to `print` a message.


```
print("lets do some coding.")
```

- Call your `intro` function from your program.

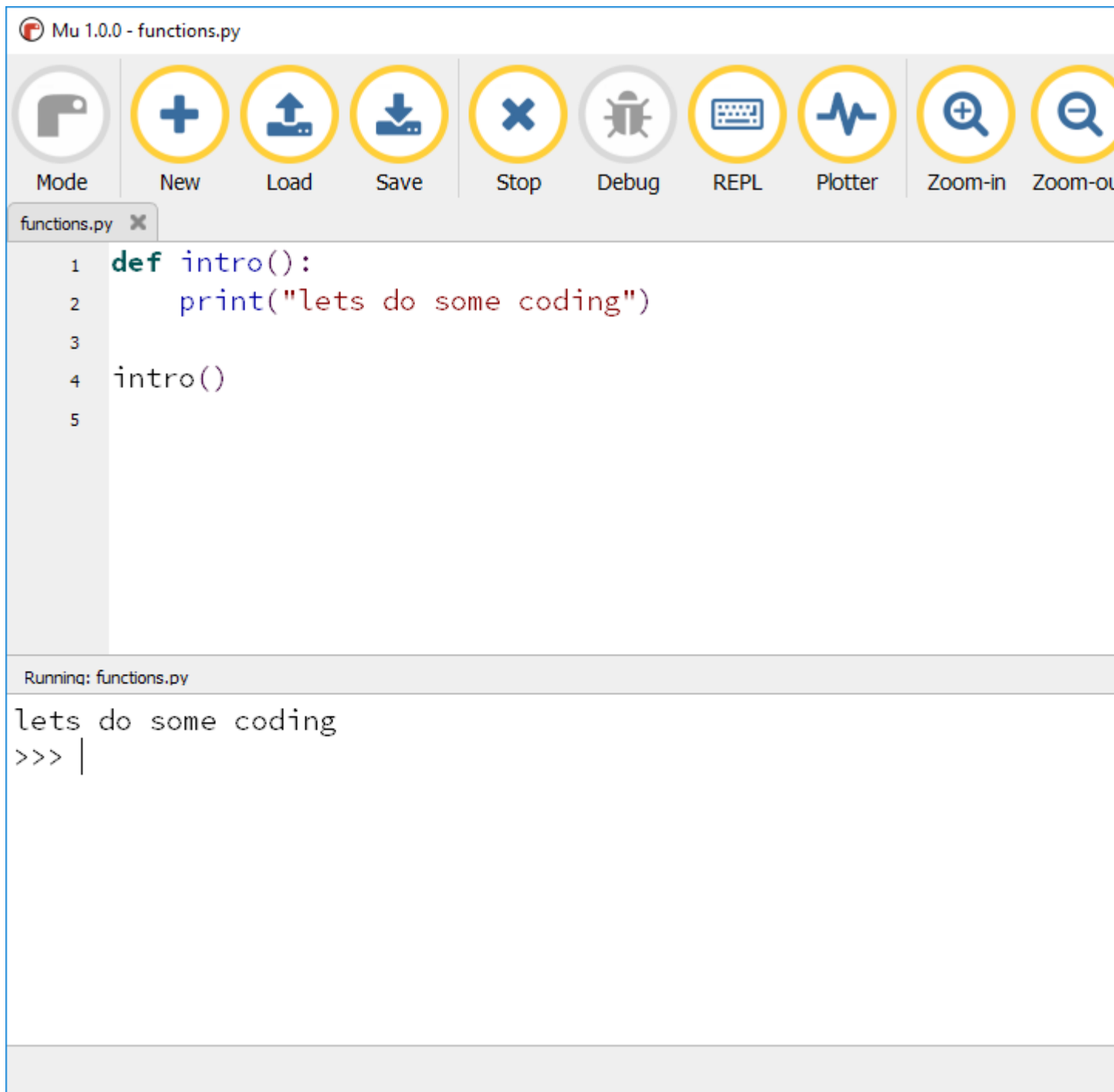
Note: this line of code is not indented under the `def` as it doesn't belong to the function definition.

```
intro()
```

Your program should now look similar to this:

```
def intro():  
    print("lets do some coding.")  
  
intro()
```

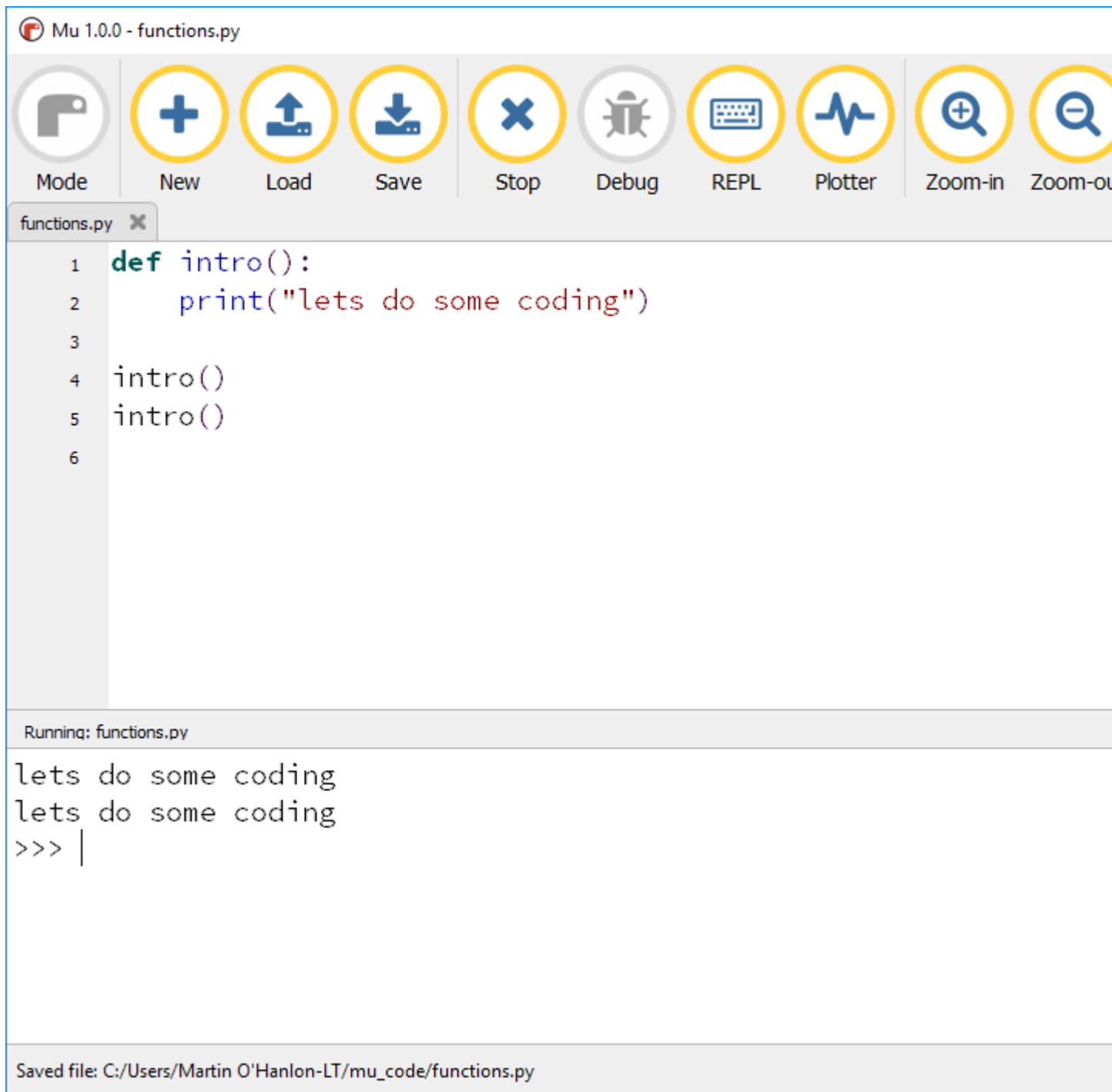
- Run your program. You should see your message.



- Change your code so the function is run twice, by adding a second call to `intro` at the bottom of your program.

```
intro()  
intro()
```

- Run your program again. You will see that the message is printed twice.



Using functions in this way allows you to reuse your code: you don't have to write out all the steps every time. It also makes the code easier to understand and change.

Try amending your intro function so it prints out a second message or asks the user for some input.

Re-using code in your bot

One question I am asked a lot is “When should I use a function?” There is no absolute rule to say when you should or have to introduce functions into your programs, but what I say is “If you can see that your program contains duplicated or very similarly structured code, then a function might help”.

If you look at your bot program, you will notice that the code which answers the maths questions (e.g. add, subtract) contains a lot of duplication and is very similar:

```
if command == "add":
    print("lets add some numbers")
    input1 = input("Number 1>")
    input2 = input("Number 2>")
    number1 = int(input1)
    number2 = int(input2)
    result = number1 + number2
    output = str(result)
    print(input1 + " + " + input2 + " = " + output)

elif command == "subtract":
    print("lets subtract some numbers")
    input1 = input("Number 1>")
    input2 = input("Number 2>")
    number1 = int(input1)
    number2 = int(input2)
    result = number1 - number2
    output = str(result)
    print(input1 + " - " + input2 + " = " + output)
```

In this section, you’ll turn these segments of code which add and subtract into one function called `do_calculation`, which will remove as much of the duplicate code as possible, and abstract the complexity of doing calculations on two numbers.

- First, let’s identify the differences between the code which adds and the code which subtracts. This will allow us to see where our new function will need to differ from the existing code:
 1. The `print` statement says “add” or “subtract”
 2. Either the `+` or `-` operator is used to calculate the `result`
 3. The operator printed by the final `print` statement is also different.

```

if command == "add":
    print("lets add some numbers")
    input1 = input("Number 1>")
    input2 = input("Number 2>")
    number1 = int(input1)
    number2 = int(input2)
    result = number1 + number2
    output = str(result)
    print(input1 + " + " + input2 + " = " + output)

elif command == "subtract":
    print("lets subtract some numbers")
    input1 = input("Number 1>")
    input2 = input("Number 2>")
    number1 = int(input1)
    number2 = int(input2)
    result = number1 - number2
    output = str(result)
    print(input1 + " - " + input2 + " = " + output)

```

There are only three areas where the code differs. The `do_calculation` function will cater for the differences by using data held in variables or using selection to either add or subtract.

- At the top of your bot program, use `def` to create a new function called `do_calculation`.

```
def do_calculation():
```

The first step in our function is to print out “lets [add/subtract] some numbers”, adding in the operation as appropriate.

- Use a `print` statement to output the message using the `command` variable to include the operation.

```
print("lets " + command + " some numbers")
```

Note: this code, and the rest of the function below, is indented under the `do_calculation` definition, to show Python that it is part of the function.

- The function should then ask the user for two numbers and convert them to integers.

```
input1 = input("Number 1> ")
```

```
input2 = input("Number 2> ")
number1 = int(input1)
number2 = int(input2)
```

- You then need to calculate the `result` using either a `+` or `-` operator, which you can do using an `if`:

```
if command == "add":
    result = number1 + number2
elif command == "subtract":
    result = number1 - number2
```

- The `result` then needs to be cast to a string.

```
output = str(result)
```

Note: this line of code needs to be run whether we are adding or subtracting, so it should be indented inline with the `if` and `elif`, **not** under the `elif`.

The final step in the function is to `print` out the result including the calculation, e.g. `"1 + 2 = 3"` or `"4 - 1 = 3"`. Again there is a difference which your function will need to deal with, to decide whether to output a `+` or a `-`.

You now have a choice about how you implement this change:

1. You could include an additional `if`, `elif` which checks the command again and prints the correct result.
2. You could create a new variable called `operator`, set this to either `+` or `-` in the existing `if`, and use that to output the variable.

Both options will work and are described below. You should review these options, decide which you prefer, implement it, and discuss your decision in the comments.

Option 1

- Add an `if` statement to the bottom of your function to print the correct result.

```
if command == "add":
    print(input1 + " + " + input2 + " = " + output)
elif command == "subtract":
    print(input1 + " - " + input2 + " = " + output)
```

Your completed `do_calculation` function should look like this:

```
def do_calculation():
    print("lets " + command + " some numbers")
    input1 = input("Number 1>")
    input2 = input("Number 2>")
    number1 = int(input1)
    number2 = int(input2)
    if command == "add":
        result = number1 + number2
    elif command == "subtract":
        result = number1 - number2
    output = str(result)
```

```

if command == "add":
    print(input1 + " + " + input2 + " = " + output)
elif command == "subtract":
    print(input1 + " - " + input2 + " = " + output)

```

Option 2

- In the `if` statement which calculates the `result`, create a variable called `operator` and set it to be either “+” or “-”.

```

if command == "add":
    result = number1 + number2
    operator = " + "
elif command == "subtract":
    result = number1 - number2
    operator = " - "

```

- Output the result using the `operator` variable.

```

print(input1 + operator + input2 + " = " + output)

```

The completed `do_calculation` function should look like this:

```

def do_calculation():
    print("lets " + command + " some numbers")
    input1 = input("Number 1> ")
    input2 = input("Number 2> ")
    number1 = int(input1)
    number2 = int(input2)
    if command == "add":
        result = number1 + number2
        operator = " + "
    elif command == "subtract":
        result = number1 - number2
        operator = " - "
    output = str(result)
    print(input1 + operator + input2 + " = " + output)

```

Discuss

Which option did you pick? Why did you feel that was the “better” solution? Did it seem simpler? Did it look nicer?

Share your thoughts in the comments.

Note: there is no right or wrong answer here, it’s just a choice. Making implementation choices is an important part of programming.

Run your function

Now that you have a function which will calculate either addition or subtraction, you need to call it from your bot program.

- Remove all the old code which did the calculation for “add” and “subtract” commands, and call your `do_calculation` function instead.

```

if command == "add":
    do_calculation()

elif command == "subtract":
    do_calculation()

```

- Run your program and test your bot.

Tip: - a mistake people often make when calling a function is missing off the round brackets () - if you miss them off your program will still start but your function won't run, so if there are no errors but nothing is happening, check your ().

Create a function for totals and averages

Now, see if you can create a function which will capture a list of numbers from the user and calculate either totals or averages.

1. Your bot should recognise the commands “total” or “average”
2. The bot should ask “How many numbers?”
3. It should use a loop to collect the numbers.
4. It should then calculate either the total or the average, and output the result.

Tip: look at the code you created to calculate averages from Week 3, and identify the changes needed to make it just display a total instead.

Case studies: normalising bugs

When students find that their code doesn't work, some of them feel it's because they are “not good at programming”, rather than it being a natural part of programming. Here are some of the ways that our educators help their students accept that finding and having to fix bugs is normal:

- Primary school teacher Beverley McCormick takes a historical route, before celebrating overcoming the inevitable bugs:

“I show them the original ‘bug’ that was found in Grace Hopper’s computer and talk about difficulties in life. I talk about Edison having to try hundreds of times to develop the light bulb and develop Positive Growth Mindset in all classes. I also develop the idea in pupils that this is just for fun and mistakes are OK! I celebrate overcoming bugs and problems with high fives and encouraging pupils to share their successes with other students.”

Beverley McCormick, primary school teacher

- Teachers Joe Mazzone and Bob Irving both include bugs while they are coding in front of their students, whether deliberately or not!

“The best strategy to make bugs and debugging a normal activity is including bugs in your lesson. As I am live coding with students, I often add a bug on purpose to see if the students can identify the problem. They then see that it is normal for people to have bugs and errors.”

Joe Mazzone, high school teacher

“I do a couple of live coding demos on the smart board and have them follow what I do. Invariably I make mistakes. I say it’s just normal! Even the teacher mistypes, makes an assumption that’s wrong, forgets to import a library, saves it in the wrong folder... I try to model that I expect to make mistakes. Fixing them is just part of the process.”

Bob Irving, grade 5-8 teacher

- Former school teacher Laurel Bleich turns bug-hunting into an activity of its own.

“At some point, I have printed out example code and I have the kids ‘find the bug’.”

Laurel Bleich, former school teacher

What techniques will you use to avoid your students feeling that they aren’t suited to coding? Share your ideas with other educators in the comment section.

Glossary

- **conditional loop**: a loop which keeps repeating for as long as a given condition is true
- **infinite loop**: a loop which will repeat forever
- **while**: a type of conditional loop in Python
- **abstraction**: removing complexity from a problem by hiding background details and information
- **function**: a reusable piece of code
- **iterable**: data which can be iterated or “stepped over”
- **iteration**: repeating the same action more than once
- **list**: a data type which can hold many items of data in order
- **index**: the location of an item in a list
- **for**: a type of loop used to iterate through the items in a list
- **condition**: an expression which will be evaluated as either true or false
- **evaluation**: determining the value of an expression, e.g. `True` or `False`
- **selection**: using **conditions** to make choices of what to do next
- **if, then, else**: tool for making code run depending whether a statement is true or false
- **and, or**: logical operators which can combine two conditions
- **scope**: defining which code belongs to which part of the program
- **indentation**: adding spaces to the start of lines of code in Python so that it understand scope
- **sequence**: describing code running one line at a time in order
- **function**: a self-contained instruction which performs a specific task
- **parameter**: data passed to a function
- **code**: the instructions that make up a program
- **variable**: a stored value in a program which is referenced by a name
- **data type**: the type of data stored in a variable (e.g. string, integer)
- **string**: a text based value
- **integer**: a whole number
- **bug**: a problem within a program
- **debugging**: the process of finding and resolving bugs
- **IDE**: Integrated Development Environment
- **concatenation**: adding pieces of text together
- **syntax**: the correct structure or layout of code