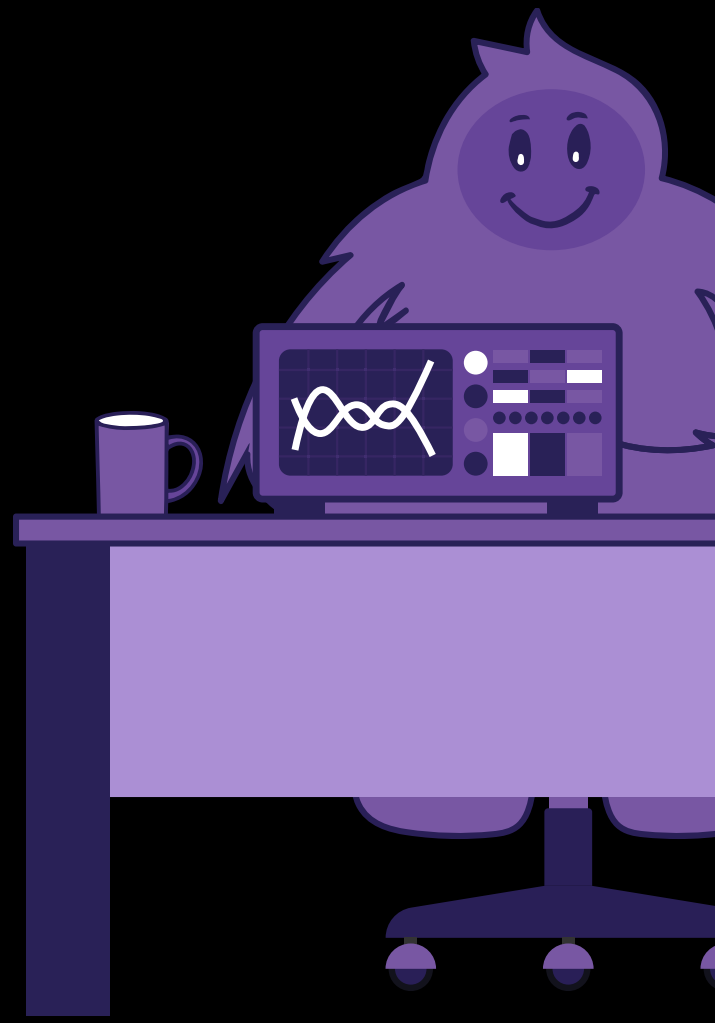
INTERRUPT LABS

Python 101

Overview

- This slide deck covers the basics of Python, it's fairly large and intended to be more of a *reference* you can come back to in the future. If you're already familiar with Python, feel free to skim through and skip parts you already know!



What is Python?

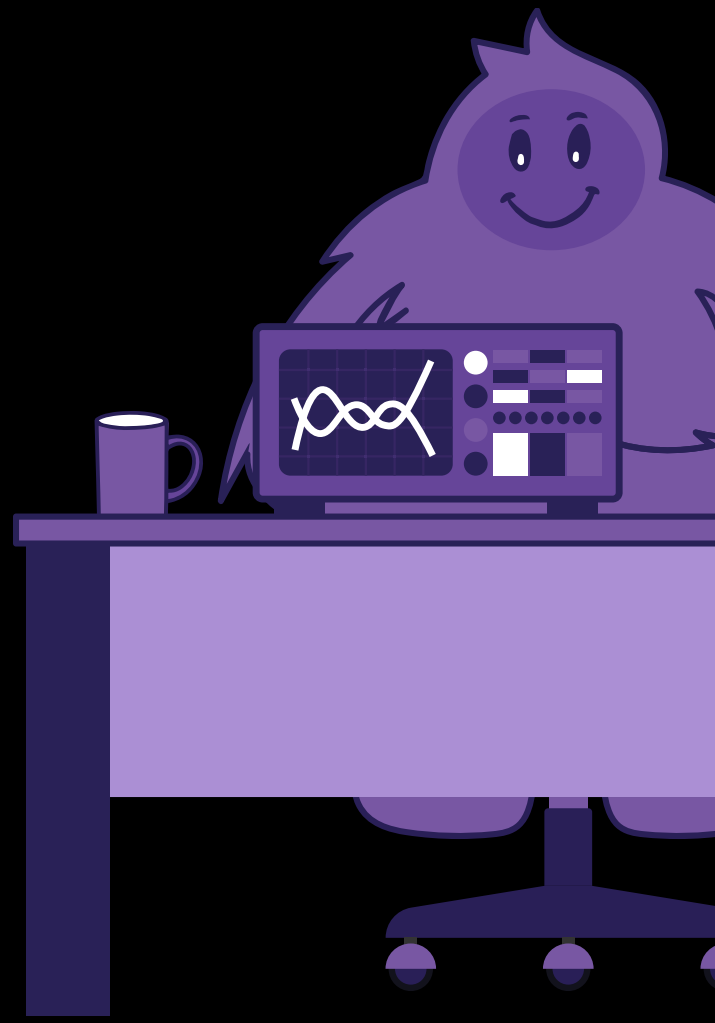
- Python is a programming language created by Guido van Rossum in 1991
- Python is an *interpreted* language, meaning programs written in Python aren't compiled to produce executables, they are instead run by a program called the interpreter
- This slows Python down at runtime, but being *interpreted* offers a whole host of benefits. For one, we don't have to spend ages re-compiling our code each time we make a change! Our code is also way more portable, any system with a Python interpreter should be able to run our code!
- You'll see two major versions of Python; Python 2 and Python 3. Python 2 is now considered deprecated; however, some older projects and libraries still use it. Programs written in Python 2 can't be run using Python 3 and vice-versa
- Python is hugely popular, packages exist for pretty much every conceivable purpose. As of May 2025, over 600,000 packages are currently available through PyPI!

What You'll Need

- To follow along, you'll need to install Python on your device
- Python can be downloaded from: https://www.python.org/downloads/

- You'll also need an editor, we'll be using VSCode however any text editor will work
- You can download VSCode from: https://code.visualstudio.com/

- Next, we'll write and run our first Python program!

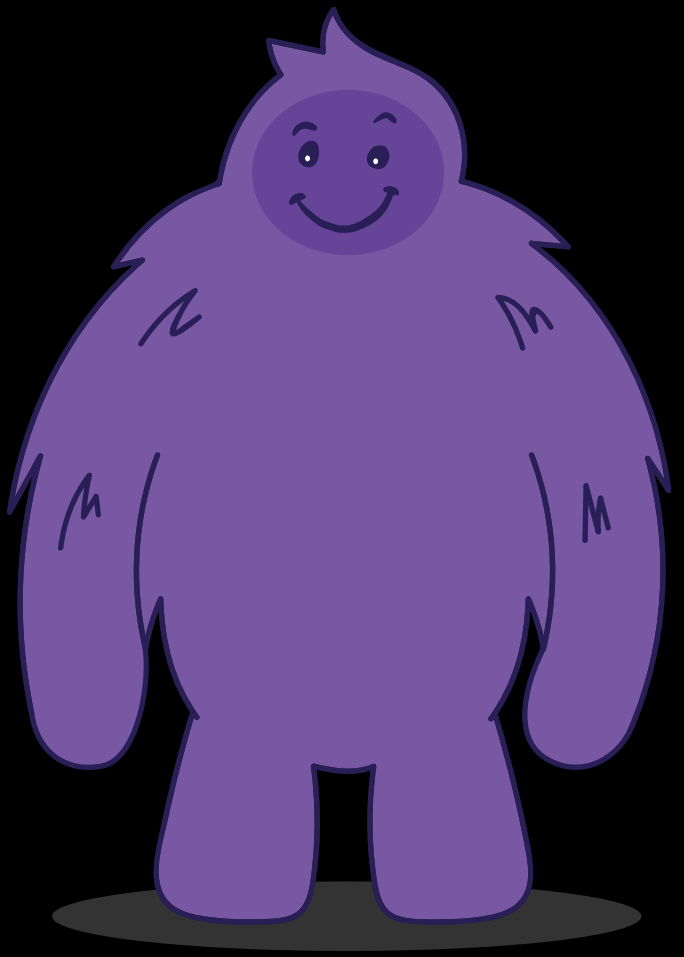


Hello, World!

- Our first program will be a simple one, let's use the print function to display a message
- Create a new file named hello_world.py and enter the following code…

hello_world.py

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



- Now let's run it!
- You can run a python program in VSCode by clicking the play button in the top right corner of the screen. You can also run a python program from the command line, but we'll cover that in the next slide

- If everything's worked, you should see a terminal window in the bottom half of the screen, with the result of our program being displayed

```
Hello, World!
```

INTERRUPT LABS

Running Python Programs

- Python can be invoked directly from the command line/terminal/shell in order to run programs

- Try typing ‘python’ followed by the name of the file containing your python code…

```
root@DESKTOP-021GAMA:~# python hello_world.py
Hello, World
```

- Depending on your install, you might need to use the command ‘python3’ instead

```
root@DESKTOP-021GAMA:~# python3 hello_world.py
Hello, World
```

- You can check the version of Python you have installed using the command ‘python –V’
- On a Linux system, you can also check the python binary present in your system path using the command ‘which python’

#Comments

- Comments are small notes you can place in your program. They don't run or impact the program in any way. They might as well be invisible to the Python interpreter!
- Any line starting with a # is a comment in Python
- Comments are a useful tool when it comes to making your programs understandable, you can use them to explain your code, why you've done things a certain way etc.

```
1   # This line prints a hello message to the screen
2   print("Hello, World!")
```

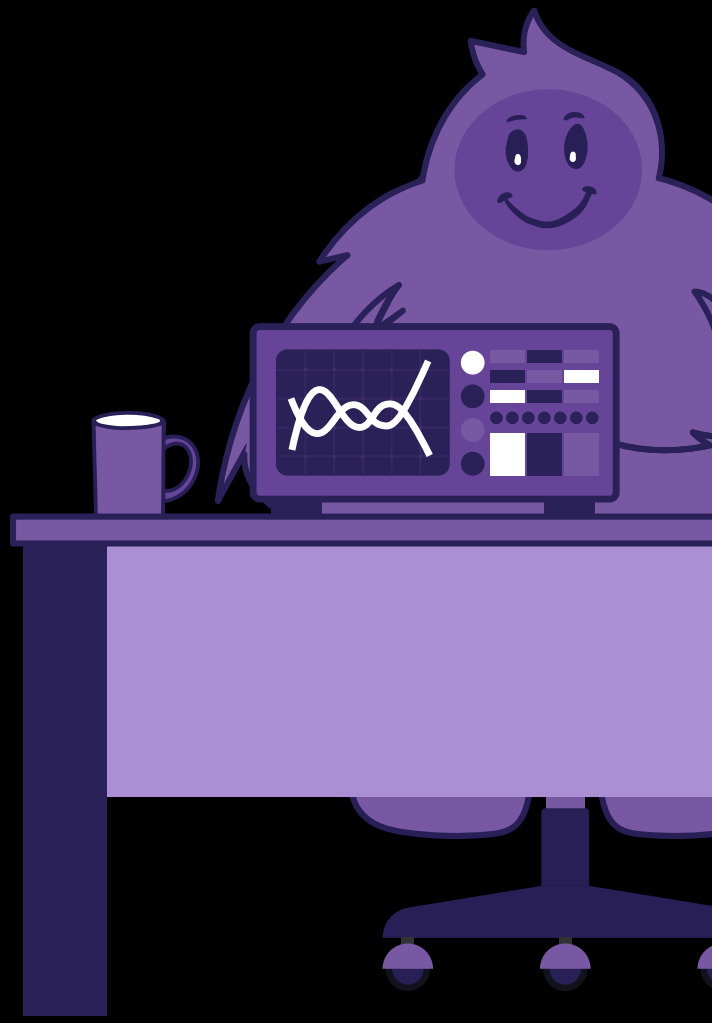
- You can also use comments to remove old code

```
1   # This line prints a hello message to the screen
2   #print("Hello, World!")
3   print("Goodbye, World!")
```

- You can even put comments on the same line as code

```
3   print("Goodbye, World!") # I'm a comment!
```

- Good comments are clear and concise, you don't need to write a novel!

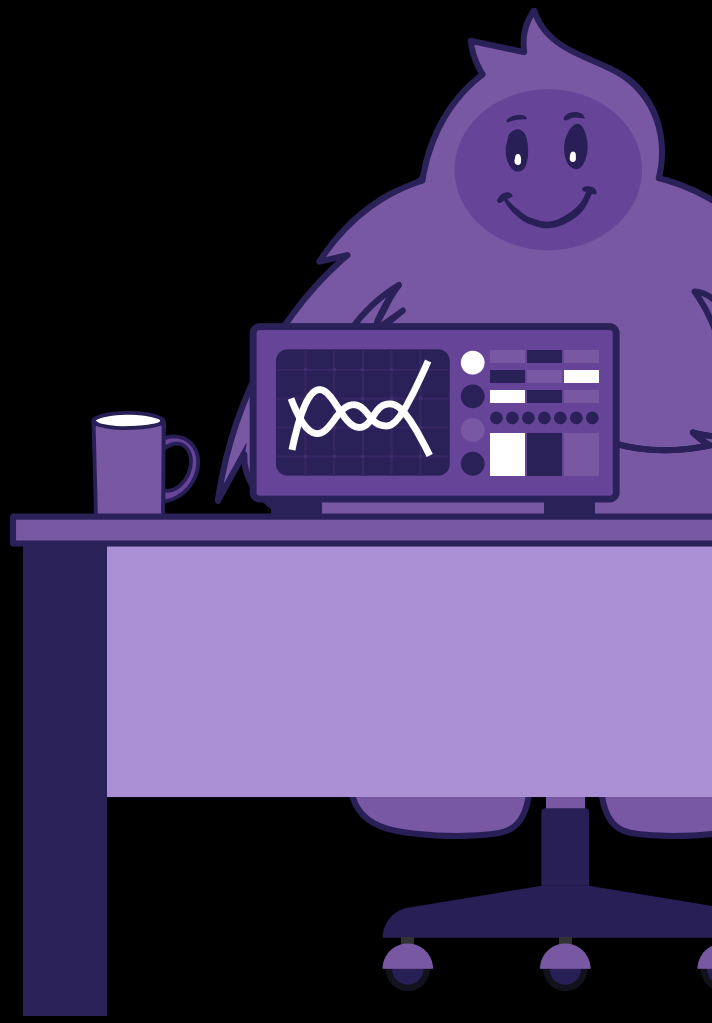


Using the Python Interpreter

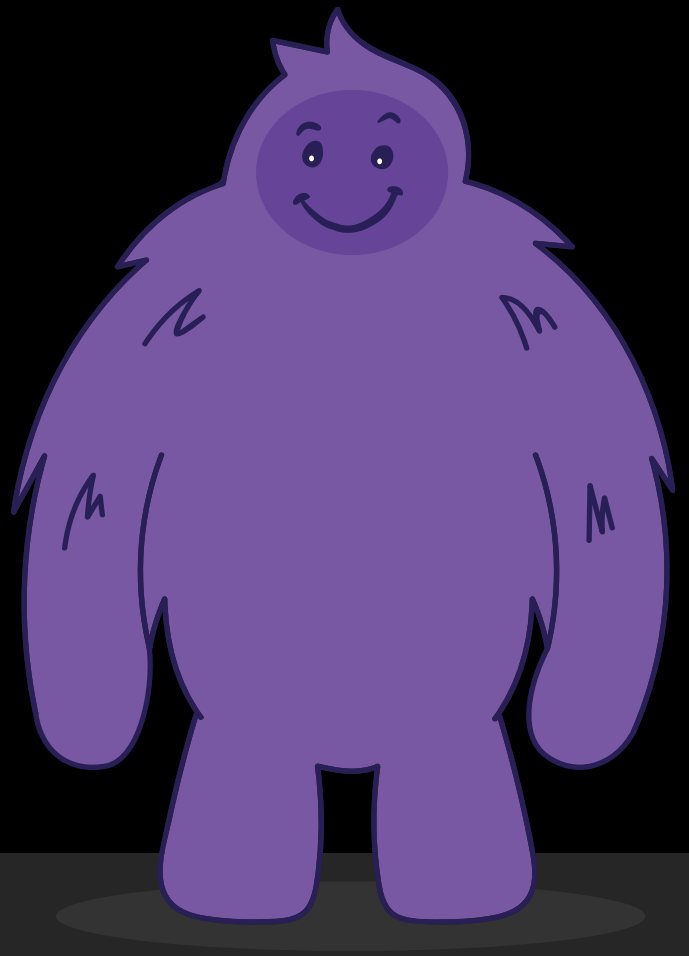
- If you use the ‘python’ or ‘python3’ command without giving it a filename, you’ll be dropped into the Python *interpreter*
- This is a program that executes code directly as you type it
- The *interpreter* can be useful for quickly testing ideas or checking that things work/don’t work

```
root@DESKTOP-021GAMA:~# python3
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- Give it a try! Drop into the Python *interpreter* and re-create the “Hello, World!” program we made in the previous slides



Variables, Operators and Data Types



Understanding Variables

- We use named *variables* to store information in our programs
- *Variables* are declared using the '=' operator. We can give our variables almost any name, so long as it doesn't start with a number and isn't the same as any of the Python *keywords*
- Let's assign some *variables*…

```
1  hello_message = "Hello from a variable!"
2  print(hello_message)
3
4  number_value = 101
5  print(number_value)
6
7  hello_message = "Hello from a re-assigned variable!"
8  print(hello_message)
```

- Here, we've declared two *variables*, `hello_message` and `number_value`
- You'll notice that we *re-assign* `hello_message`. If you run this code, you'll see that this changes the message printed to the screen

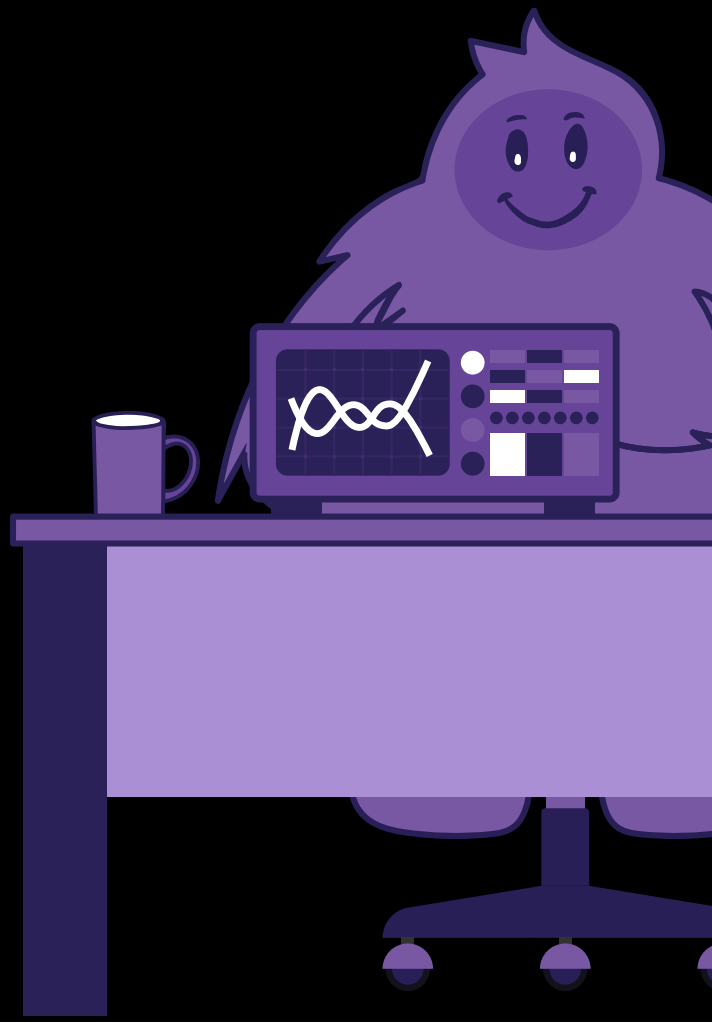
Naming Variables

- You can call variables anything you want, however there are a few *conventions* we recommend following to make your code nice and readable

- When naming *constants* (variables whose values you don't expect to change), we generally use `BLOCK_CAPITALS` to name the variable

- When it comes to other variables, we use *`snake_case`**
- With *`snake_case`*, we simply swap out spaces for underscores and avoid capital letters. `a_bit_like_this!`

- The actual name of each variable should be clear and related to its usage, please don't name your variables "x", "y" and "z" outside of the interpreter!



Assigning Multiple Variables At Once

- Python has some cool shortcuts we can use to assign multiple variables at once
- By separating variable names and values with commas on either side of the = sign, we can assign multiple variables at once

```
>>> variable_one, variable_two = 1, 2
>>> print(variable_one)
1
>>> print(variable_two)
2
>>>
```

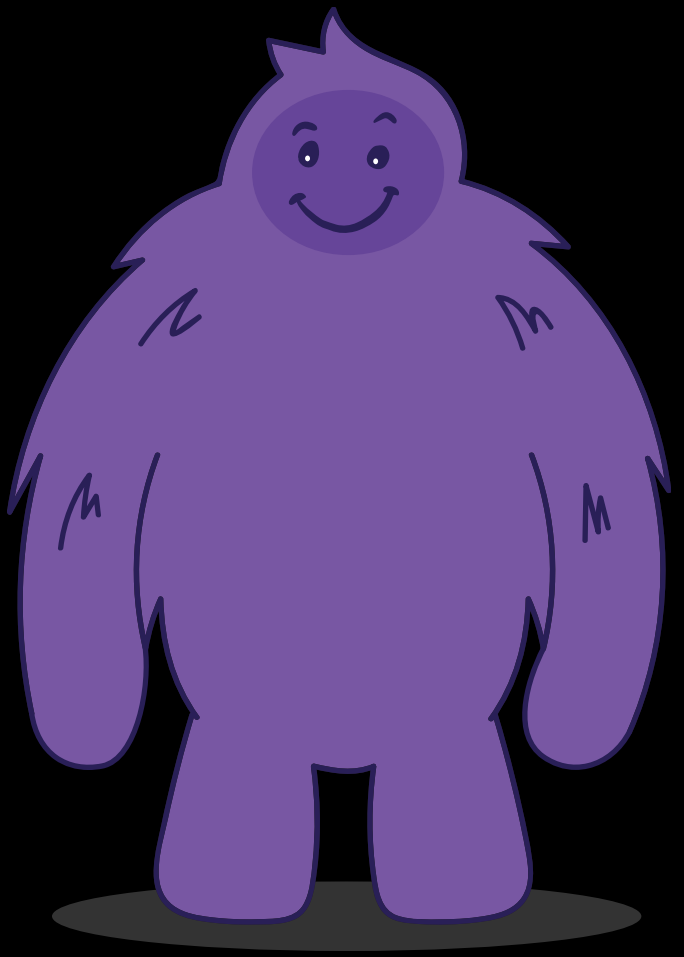
- So long as the number of values on the right is the same as the number of variables on the left, we can use this trick to store multiple values in multiple variables all at once

Operators

- Operators are used to carry out simple computations on data and variables
- There are quite a lot of operators available in Python for a range of purposes, here's a list of basic arithmetic operators as an example:

Operator	Operator Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Floor Division
%	Modulo
**	Exponent

- Let's jump into the interpreter and use some operators!



Using Operators

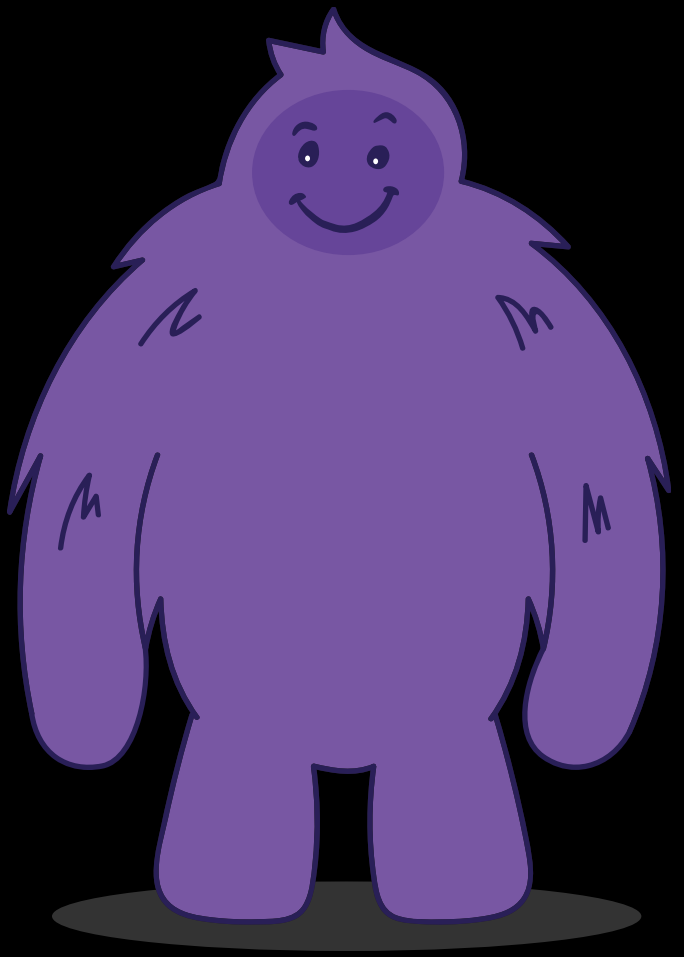
- Let's make use of some of these operators
- As you can see, we can use multiple operators in a row, but the order matters!
- Just like in algebra, we can use brackets to ensure operations happen in the order we want them to. Python will follow BODMAS ordering when carrying out operations

```
>>> 10 + 1
11
>>> 10 + 1 * 2
12
>>> (10 + 1) * 2
22
>>> 'Hello, ' + 'World!'
'Hello, World!'
>>> first_name = "Abominable "
>>> last_name = "Snowman"
>>> first_name + last_name
'Abominable Snowman'
>>>
```

- Another thing to note here is that the behaviour of operators can change depending on the *type* of data we are using
- When we add numbers using the '+' operator in the first few lines, we get a numeric result, yet using the same operator on two strings will combine them
- If you're unsure on how an operator works, make sure to reference the Python docs or just try playing around with it in the interpreter!

Challenge Time!

1. Open up the Python interpreter in a terminal or in your IDE and try assigning some data to variables
2. Declare a variable using another variable
3. Change the value of the original variable, what happens to the newly declared one?
4. Assign a variable to some large float or integer value, now try *re-assigning* that variable to 50 times itself



Data Types

- Python is a *dynamically typed* language…
- This means that we don't have to declare the type of a variable, Python will infer it for us from the value we assign to it

- A variable's *type* simply refers to the sort of data we are storing in that variable
- We can discover the type of data currently stored in variable using Python's `type()` method

```
>>> x = 101
>>> type(x)
<class 'int'>
```

- Python uses lots of different data types, though there are a few basic types you'll see again and again:

Data Type	Explanation
String: str	Used for any sort of text data, e.g. 'a' or 'hello, world!'
Integer: int	A whole number
Float: float	A number with a decimal point
Boolean: bool	Can be either True or False

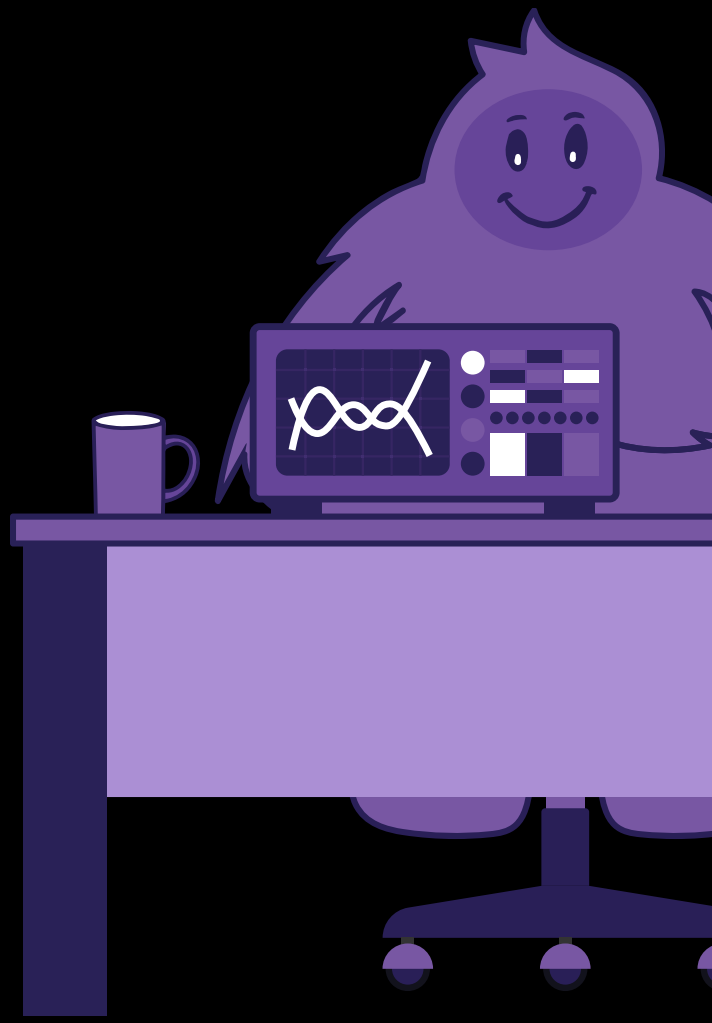
More on types…

- Python is a *strongly typed* language, all this really means is that Python will throw an error when we try to use or mix variables in ways that aren't supported

- We can see this if we try to use operations on data in ways that don't make sense. For example, we'll get an error if we try to add a string to a number…

```
>>> string = "Hello, World"
>>> number = 5
>>> string + number
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>>
```

- Sometimes, we can convert data to a new type when we need to use an unsupported operation…



Casting ~~Spells~~ Types…

- We can convert data from one type to another through a process called *casting*
- Exactly how the data is converted will depend on the types of data involved

- Here, we've *casted* an int to a float and vice-versa
- To cast one data type to another, we use the name of the desired type followed by a pair of brackets with the name of our target variable e.g. `int(…)` and `float(…)`

```
>>> int_number = 100
>>> float_number = 1.75
>>>
>>> int(float_number)
1
>>> float(int_number)
100.0
>>>
```

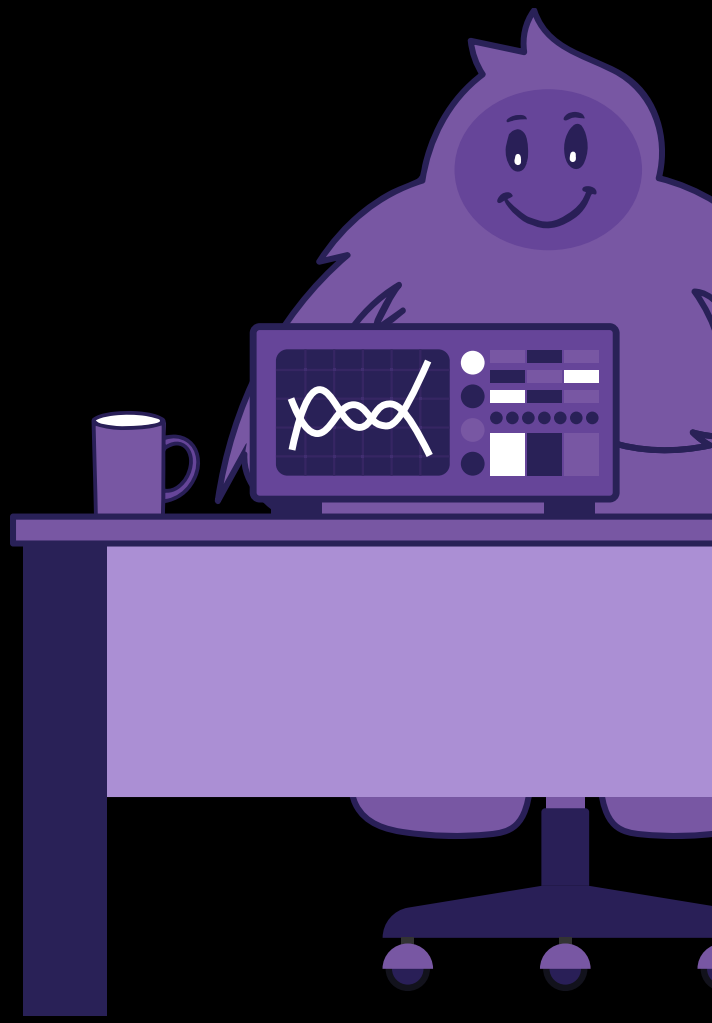
- As you can see, by casting "`float_number`" as an int, we lose the .75 and are left with the 1. Note that Python doesn't round up or down, it simply ignores everything after the decimal point!
- By casting "`int_number`" as a float, we get 100.0. The underlying value of 100 remains unchanged

Challenge Time!

1. Running this code gives us a type error, fix the code to print the correct message:

type_challenge.py > ...

```python
first_bit_of_message = "the yeti is "
second_bit_of_message = "ft tall!"
height = 12

complete_message = first_bit_of_message + height + second_bit_of_message
print(complete_message)
```



Assignment Operators

- Python lets us change the values stored in variables by performing operations on their existing values…

```
>>> x = 10
>>> x = x * 2
>>> print(x)
20
```

- This is a pretty common operation, so Python offers a nice compact way of changing variables in this way:

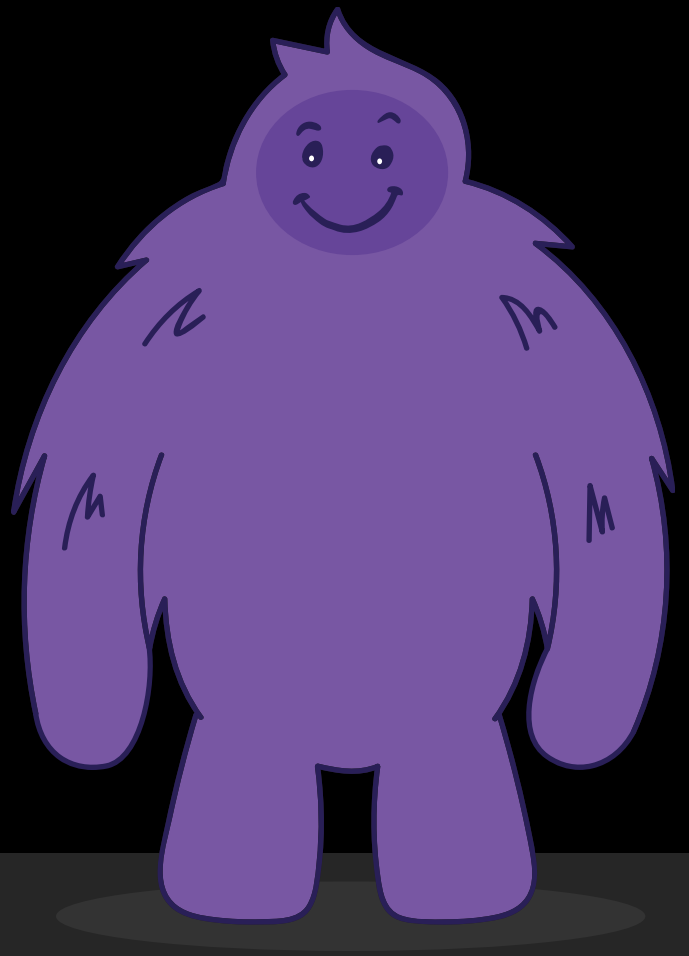
```
>>> x = 10
>>> x *= 2
>>> print(x)
20
```

- As you can see in the examples above, `x = x * 2` and `x *= 2` are fully equivalent

- *Assignment Operators* like this exist for each of the basic arithmetic operators. You'll see some commonly used ones listed on the right, however a full list can be found at: https://www.w3schools.com/python/gloss_python_assignment_operators.asp

Operator	Name
+=	Add and Assign
-=	Subtract and Assign
*=	Multiply and Assign
/=	Divide and Assign
%=	Modulo and Assign
**=	Exponent and Assign
//=	Floor Divide and Assign

Input



Taking Input

- To make our programs interactive, we need a way of taking input from users
- Python provides a really useful function, *`input()`*, that can be used to take input from the terminal
- We provide *`input`* with a *prompt* that is printed to the screen when the program runs
- When the `input()` method is hit, the program will pause execution until the user has typed something on the terminal and pressed enter. The user's data is stored in the program as a *string*
- Let's look at an example:

```
1   name = input("What is your name? ")
2   message = "Hello " + name
3   print(message)
```

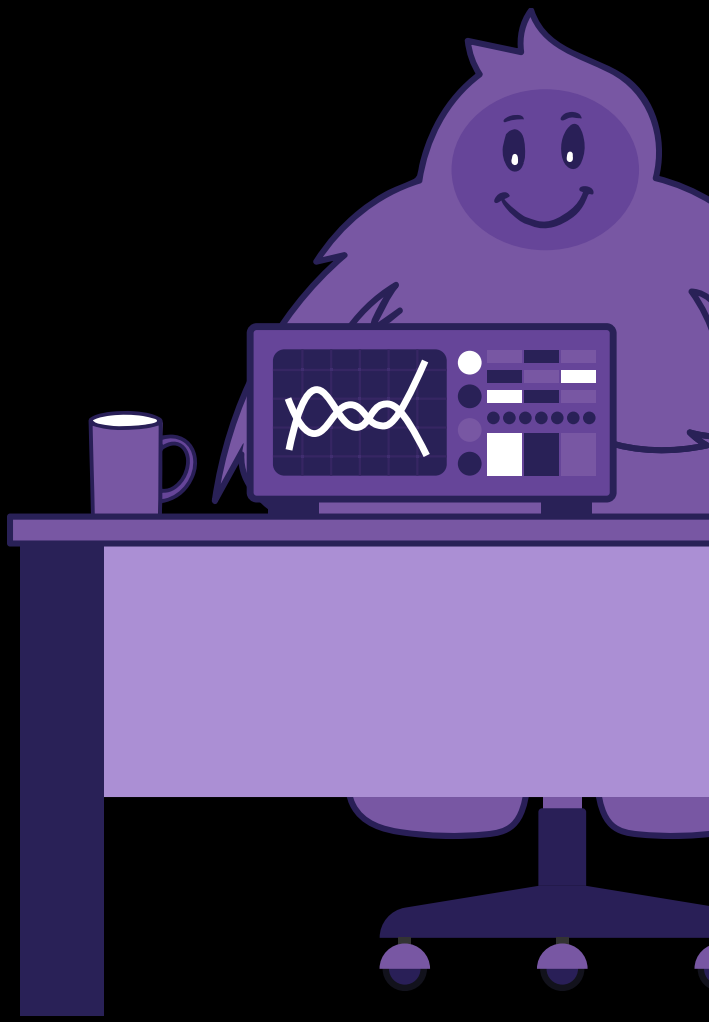
- This program will display the prompt "What is your name? " and pause to accept user input. Any entered data is stored in the "name" variable as a *string*
- Finally, a small message is created and printed to the screen

Another Challenge!

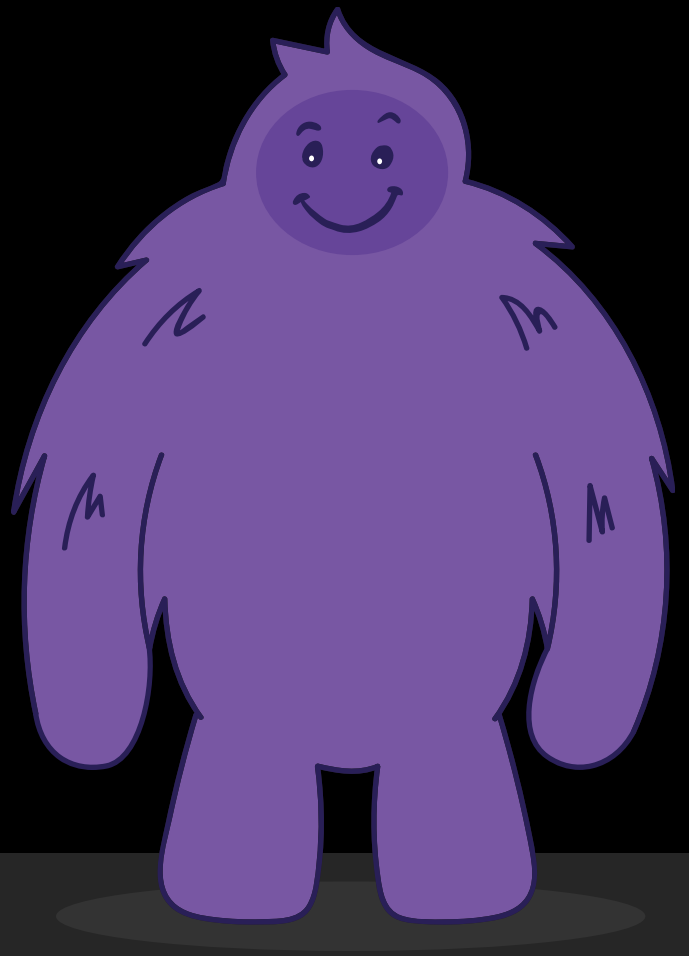
1. Use what you've learnt about the *input* method, *assignment operators* and *data types* to write a program that does the following:
    - Takes the user's name and age as input
    - Prints a hello message that contains the users name and age
    - Calculates what the user's age will be in 5, 10 and 20 years
    - Displays these ages to the screen

    The program output should look similar to this…

```
What is your name? Yeti
How old are you? 100
Hello Yeti, you are 100 years old!
In 5 years, you will be 105 years old!
```



If, elif and else



Boolean Expressions

- We'll get onto `if`, `else` and `elif` statements I promise but first let's look at Boolean expressions
- Boolean expressions are simply expressions that evaluate to either true or false
- Boolean values (`True` and `False`) are the simplest Boolean expressions!

- Some examples of Boolean Expressions:

```
>>> 100 > 10
True
>>> 10 > 1000
False
>>> True and False
False
>>> True or False
True
>>> x = 4
>>> 2*x > 6
True
>>>
```

- Each expression here uses *boolean operators* to compare values, resulting in a `True` or `False` value

Boolean Operators

- Python includes *and*, *or*, and *not* operators that can be used on Boolean values and expressions. Mathematical operators like <, >, <= and >= can also be used in Boolean expressions

- The equality operator is == and checks if two things are equal…

```
>>> 10 == 10
True
>>> 10 == 5
False
```

- Make sure not to use =, this is the assignment operator used for setting variables!

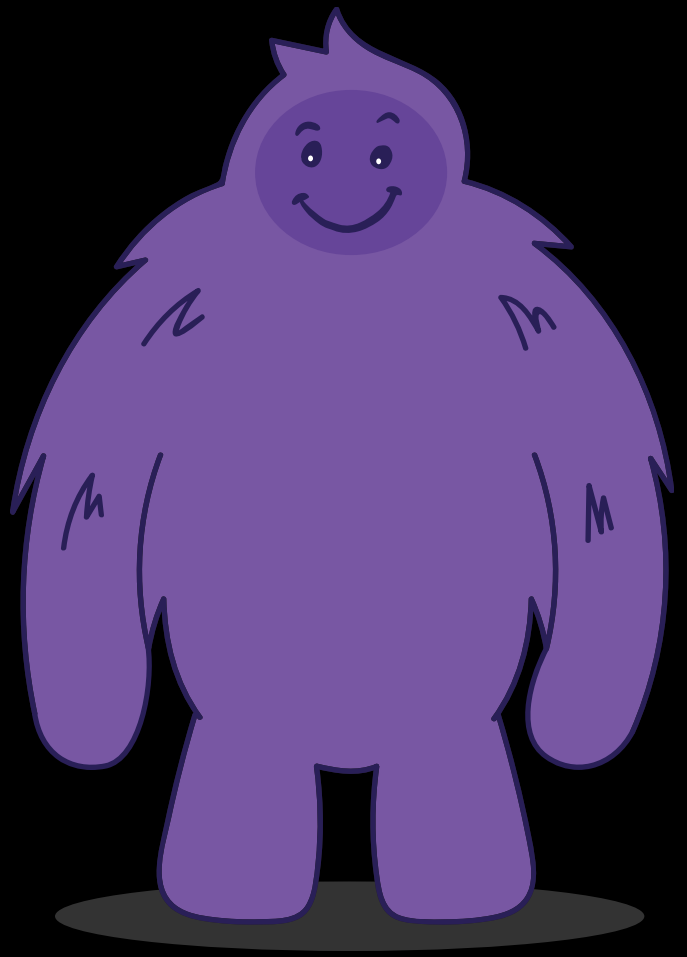
```
>>> 10 = 10
  File "<stdin>", line 1
    10 = 10
    ^^
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of '='?
```

If Statements

- `if` statements let us control which bits of code run based on the outcomes of Boolean Expressions
- Let's look at an example:

```
1  temp = int(input("How many degrees is it today? "))
2
3  if temp > 25:
4      print("Make sure to wear sun screen!")
5  else:
6      print("I hope it warms up!")
```

- This code takes some user input and casts it as an integer
- If the Boolean expression "`temp > 25`" evaluates to true, the code on line 4 will run. If not, the code on line 6 will run



- All if statements follow this pattern, they check if a Boolean expression is true and define some code to run if it is
- `else` statements define some code to run if the statement is false. An `else` statement doesn't always need to be defined; without one, the program will simply do nothing if the Boolean expression is false!

else + if = elif

- We can make our if logic a bit more complex using `elif` statements!
- `elif` is short for *else if*, which means we are adding an else block and then using an if statement to decide if it runs
- Let's look at an example:

elif_statements.py > ...

```
1  temp = int(input("How many degrees is it today? "))
2
3  if temp > 25:
4      print("Make sure to wear sun screen!")
5  elif temp > 15:
6      print("Seems like a nice day out!")
7  else:
8      print("I hope it warms up!")
```

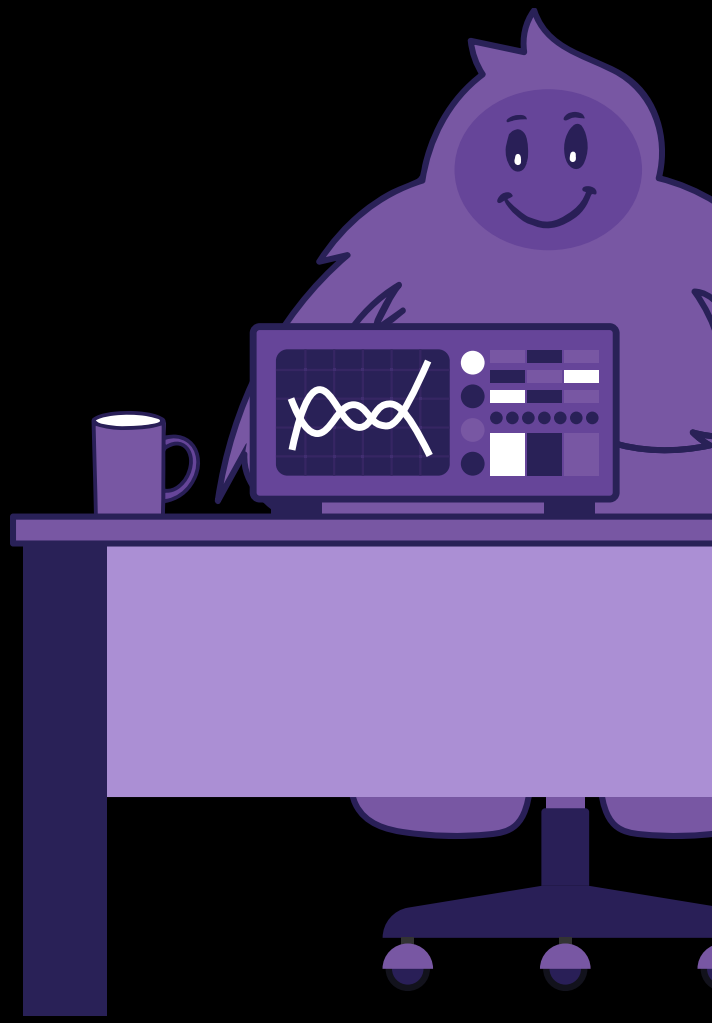
- Here, we've added an `elif` statement to our previous program. This statement will trigger if the previous `if` statement is false and its Boolean expression is true
- This means that if we get a temperature less than or equal to 25 and greater than 15, the new message will be displayed to the screen

If elif elif else

- Another useful feature of `elif` blocks is that we can add as many as we want to an `if` statement

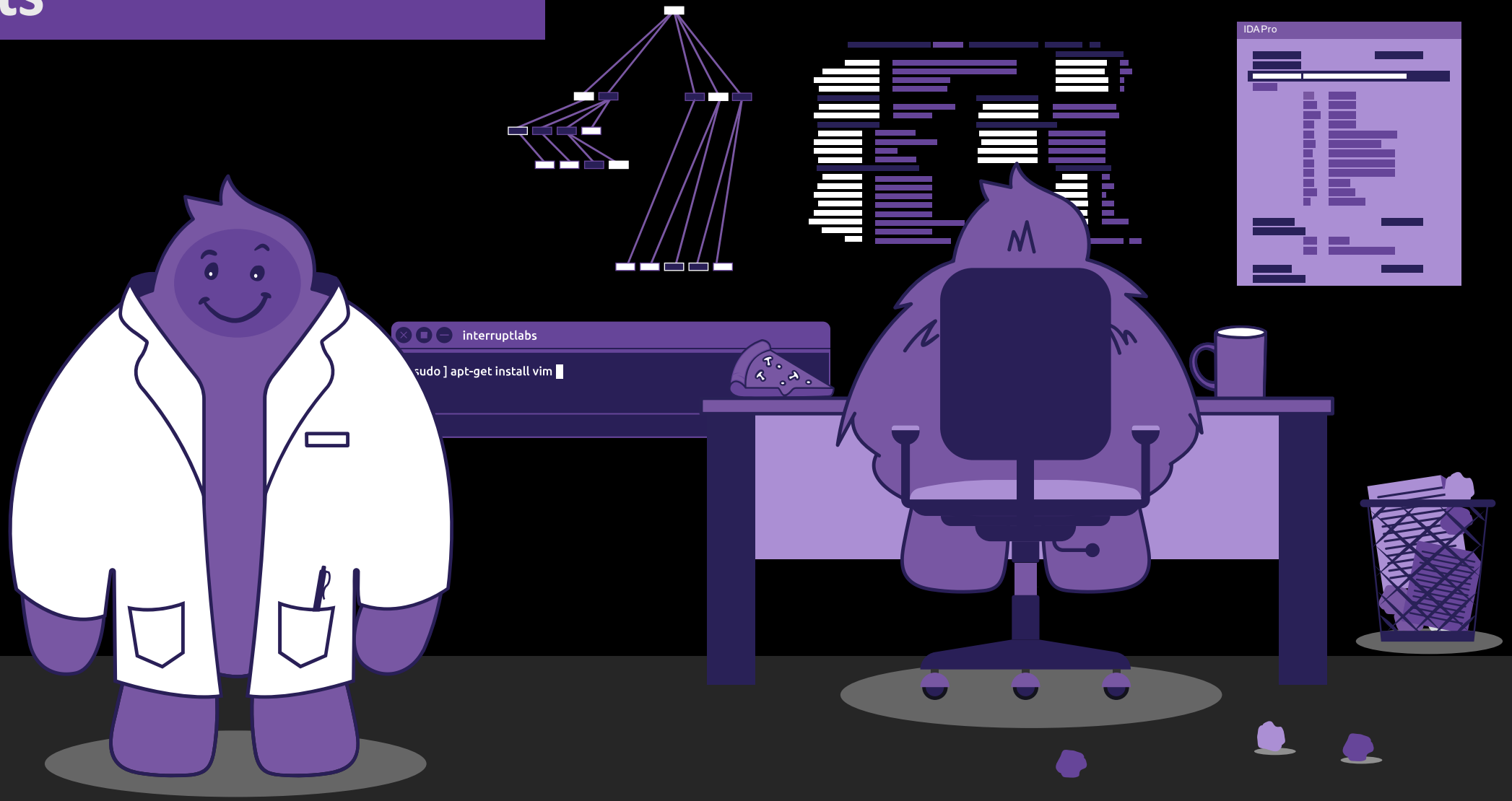
elif_statements.py > ...

```
1   temp = int(input("How many degrees is it today? "))
2
3   if temp > 25:
4       print("Make sure to wear sun screen!")
5   elif temp > 15:
6       print("Seems like a nice day out!")
7   elif temp > 10:
8       print("Maybe wear a hoodie?")
9   else:
10      print("I hope it warms up!")
```



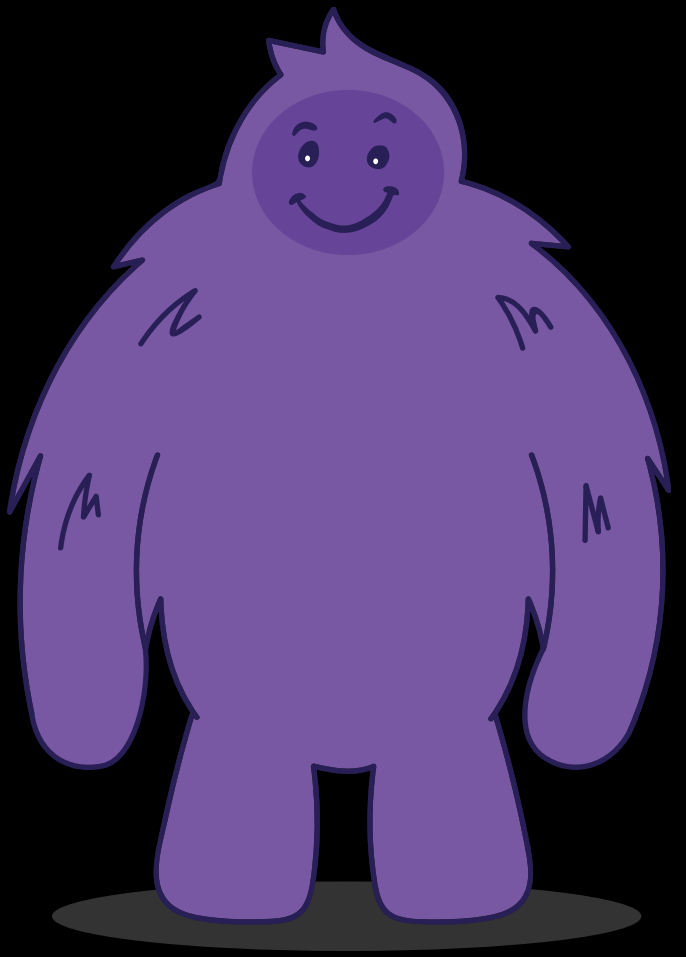
- We've modified the previous program to add another `elif` block. This time, our new block will run if *all previous* `if`/`elif` statements don't and its Boolean expression is true!
- This means our new message will be displayed if the user inputs a temperature value smaller than 25, smaller than or equal to 15, and larger than 10

Lists



Variables, but More

- Lists are another key data type you'll use again and again in Python
- However, unlike ints, floats and strings, lists are special in that they can contain *multiple values*
- Lists can be used to house collections of values, variables, even other lists! The items we store in a list are called its *elements* or its *members*



- Let's look at an example:

lists.py > ...

```
1    a = 10
2
3    my_list = [1, 2, "string", a]
```

- Here, we've created a list containing, ints, a string and even a variable!*
- Now let's try accessing some of these stored elements!

** When we store variables in a list, we actually only store the value currently stored in that variable. If we change the value of a variable after placing it in a list, the value stored in the list will remain the same*

Indexing and Accessing List Elements

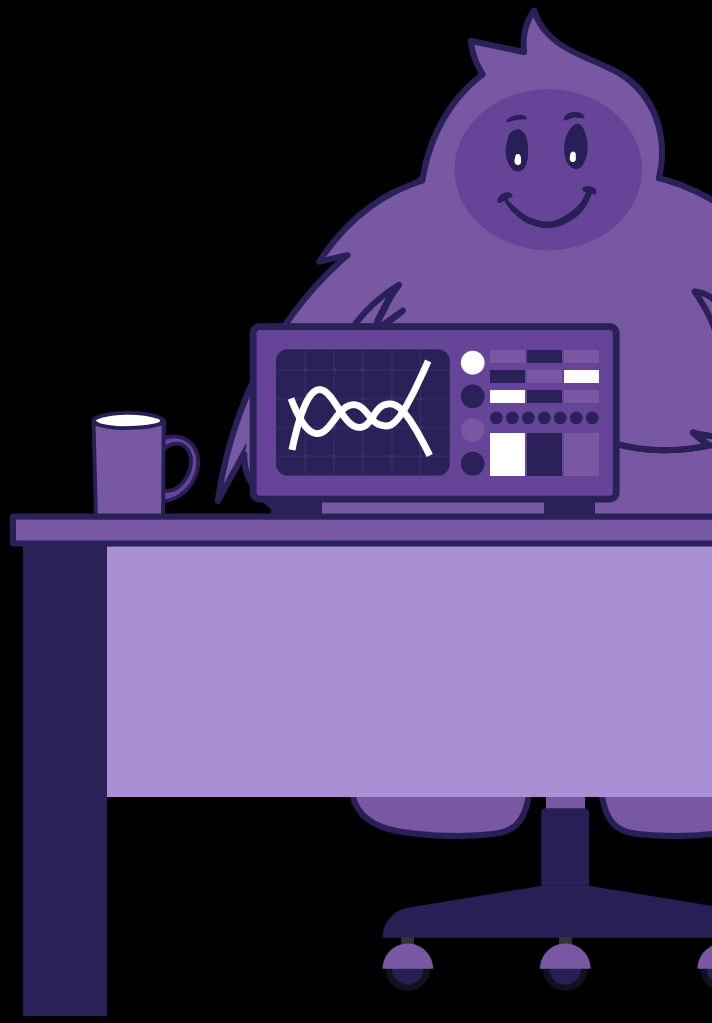
- We access elements of a list based on their *index*. The *index* of an element is simply it's position within a list
- An important thing to note is that we start counting/indexing our list elements from **0**
- Going back to our previous example, let's look at the indexes of each element:

0	1	2	3
1	2	"string"	a

- To access an element of a list, we use *square brackets* containing the *index* of the desired element
- Try running this code, what elements should be printed to the screen?

lists.py > ...

```
1   a = 10
2
3   my_list = [1, 2, "string", a]
4
5   b = my_list[0]
6   c = my_list[2]
7
8   print(b)
9   print(c)
```



Range Against the Machine

- Sometimes we'll just want a big list of numbers, Python's in-built `range()` function is perfect for that!
- The `range()` method can be used in a couple of different ways:

Usage	Explanation
range(100)	Provides a range of numbers from 0 to 100
range(50,100)	Provides a range of numbers from 50 to 100
range(0,100,2)	Provides a range of numbers from 0 to 100, counting in 2s (0, 2, 4, 6…)

- The `range()` method doesn't give us back a list however, it returns a range *object.* We'll cover *objects* in depth in a later section, but for now just know that we have to *cast* ranges to lists in order to use them as lists

```
>>> my_range = range(10)
>>> type(my_range)
<class 'range'>
>>>
>>> my_list = list(my_range)
>>> print(my_list)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

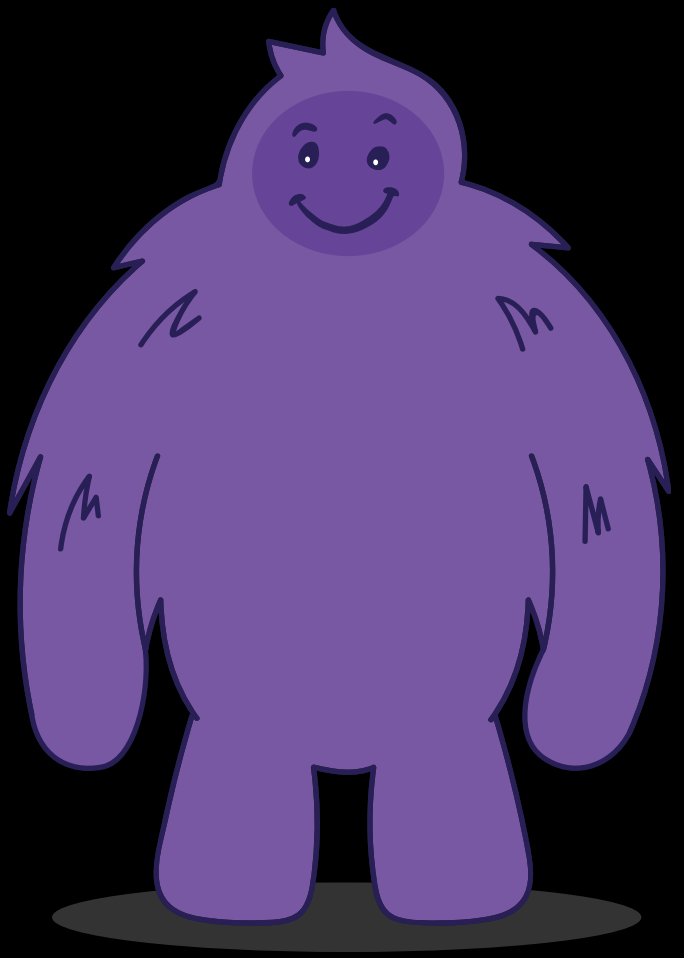
Slicing

- We know how to access individual elements of a list, but what if we want to access a sub-section of a list?
- For this we use *slicing!* We'll use the square brackets again to select elements from a list but this time we'll put in some extra values…

```
>>> my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> my_list[0:3]
[0, 1, 2]
>>>
```

- *Slicing* uses the syntax [<start index> : <stop index>] to select portions of a list
- An important thing to note is that the element at the start index is included, yet the element at the stop index is excluded!
- Python uses the start of the list as the default value of <start index> and the end of the list as the default value of <stop index>, you can use these default values by leaving <start index> or <stop index> blank

```
>>> my_list[:5]
[0, 1, 2, 3, 4]
>>> my_list[5:]
[5, 6, 7, 8, 9]
```



More Slicing!

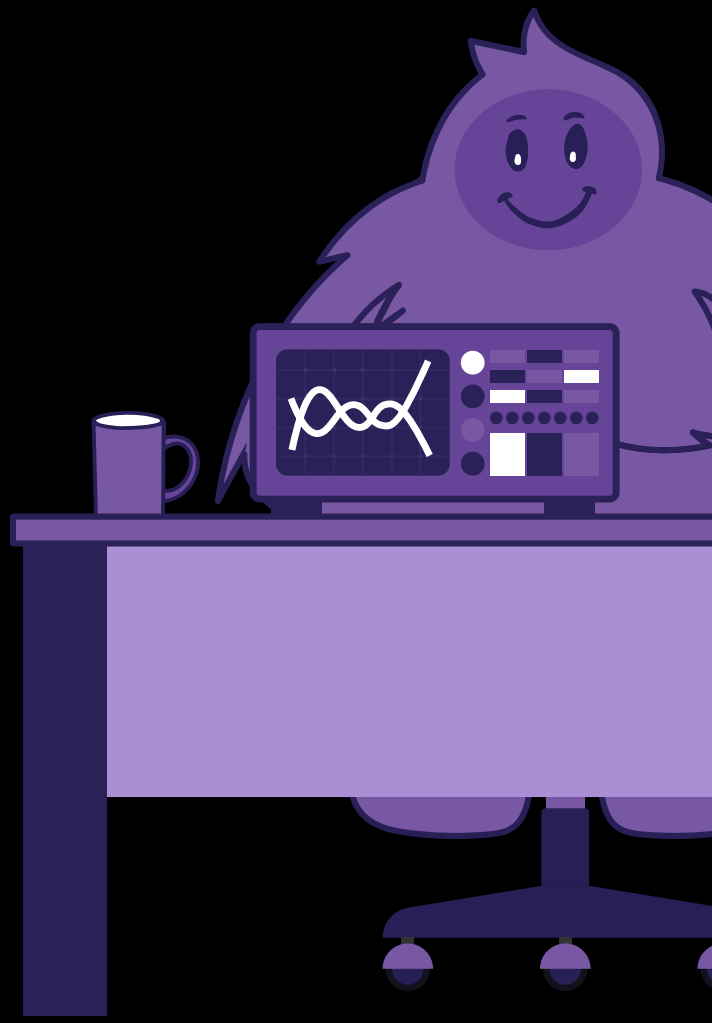
- Python has another powerful trick we can use when slicing up lists!
- We can use *steps* to select every n^{th} element from a sub-list
- To do this, we use the syntax [<start index> : <stop index> : <step>]
- For example:

```
>>> my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> my_list[5::2]
[5, 7, 9]
>>>
```

- Here, we've set <start index> to 5, left <stop index> as it's default variable and set <step> to 2
- This means we select every 2^{nd} element between the element at index 5 and the end of the list. In this case, 5, 7 and 9

Challenge!

1. Use the range method to create a list containing every number between 0 and 10,000
2. Create two new lists by slicing this list, the first should contain every *even* number, the second should contain every *odd number*



Negative Indices

- Rather weirdly, you can actually use *negative numbers* to access list elements!
- Elements in a list have both a positive and a corresponding negative index value
- Let's look at an example list to demonstrate:

```
[ "cat", "dog", "hamster", "gerbil", "goldfish", "parrot" ]
```

- Positive Indices:

0	1	2	3	4	5
cat	dog	hamster	gerbil	goldfish	parrot

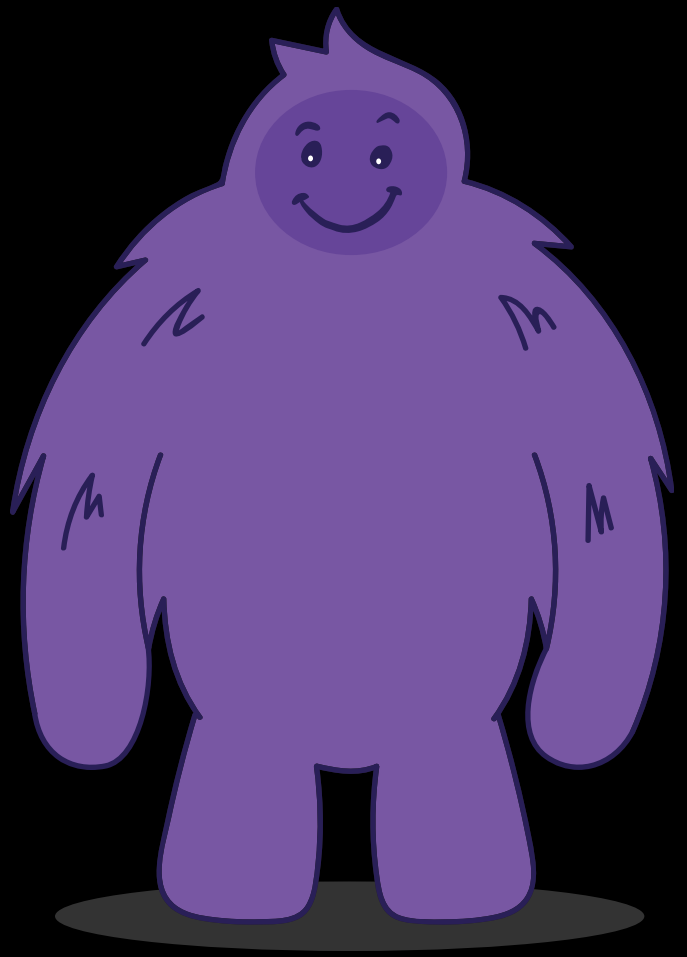
- Negative Indices:

-6	-5	-4	-3	-2	-1
cat	dog	hamster	gerbil	goldfish	parrot

A List of Useful Methods

- Python comes packed with a bunch of really useful methods for working with lists!
- It's worth spending some time familiarising yourself with these methods as they are incredibly useful for data manipulation:

Methods	Explanation
append()	Add an element to the end of a list
insert()	Add an element to a certain position in the list
extend()	Add the elements of another list to the end of the list
pop()	Get the last element from a list, removing it
remove()	Remove the first occurrence of an element in the list
index()	Get the index of the first occurrence of an element in the list
…	…



- You can find a full list of these functions here:
  https://docs.python.org/3/tutorial/datastructures.html
  https://www.w3schools.com/python/python_ref_list.asp

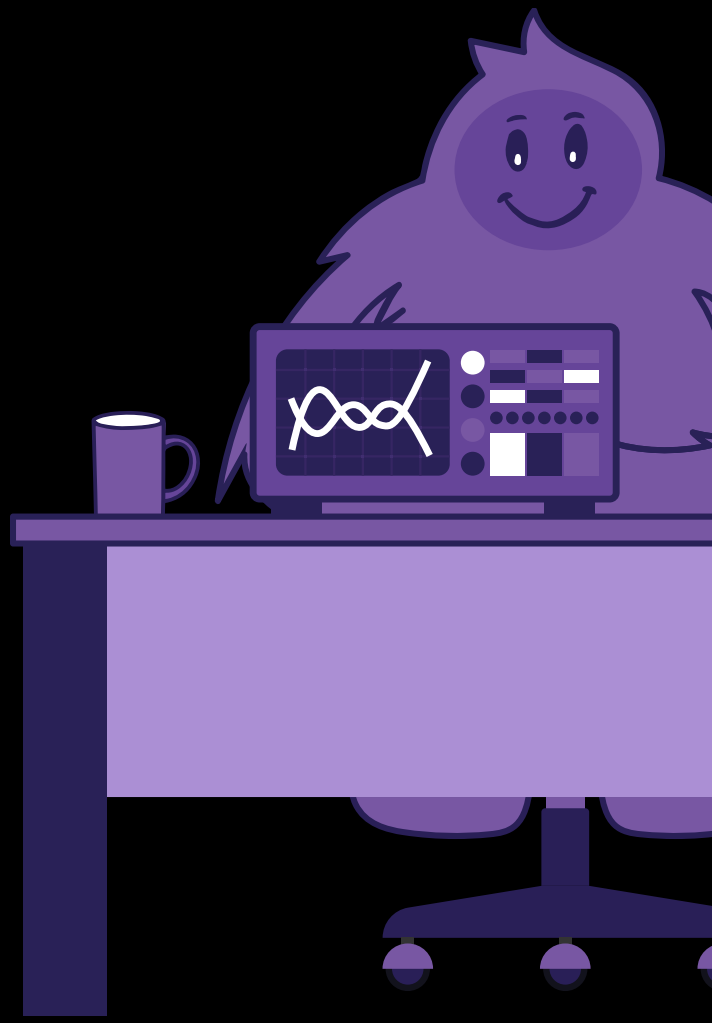
Lists Within Lists

- Lists can contain other lists as elements. These lists can contain lists and so on and so on…
- Python lets us directly access the members of lists within lists and even perform operations on them without removing them from the original list
- We access members by simply using another pair of *square brackets*…

```
>>> x = [1, 2, ["cat", "dog", "fish"]]
>>> x[2][0]
'cat'
>>>
```

- Here, the first pair of square brackets indicates we want the element of list ‘x’ with element 2. The second pair of square brackets indicates we want the element from that list with index 0, giving us "cat"
- So long as we index our lists correctly, we can use lists within lists just as we would any other list

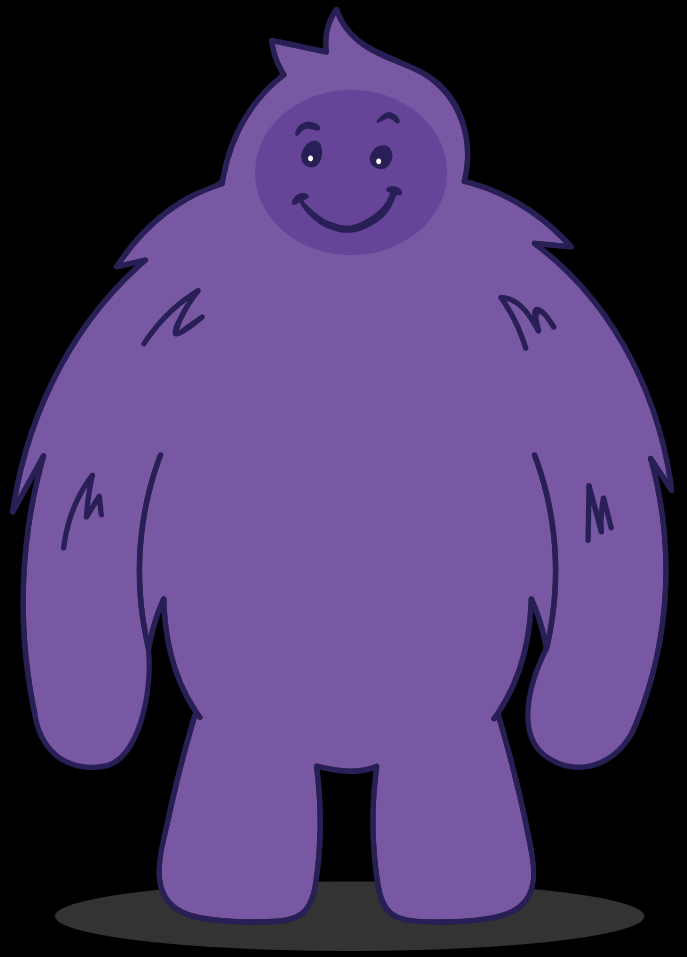
```
>>> x[2].append("another fish")
>>> print(x)
[1, 2, ['cat', 'dog', 'fish', 'another fish']]
>>>
```



INTERRUPT LABS

Challenge!

1. Create a program that accepts 5 numbers from a user and stores them in a list. Use list methods to sort these numbers from small to large and print them to the screen
2. Create a list of numbers from a range and print it to the screen. Let a user enter some data and choose where to put it. Print the list to the screen after adding the user data



Tuples

Less Than Lists, More Than Variables…

- Lists are *mutable*, this means that we can change the values in them without having to *re-assign* the variable
- Sometimes, we'll want collections of values that are *immutable*, collections of values that *don't change*
- This is where tuples come in!
- We declare tuples using regular brackets *( )*

```
1    my_tuple = (1, 2, 3)
```

- Since tuples are immutable, they support less functionality than lists. This makes them more efficient and lightweight however for most uses the difference is pretty negligible…

Dictionaries

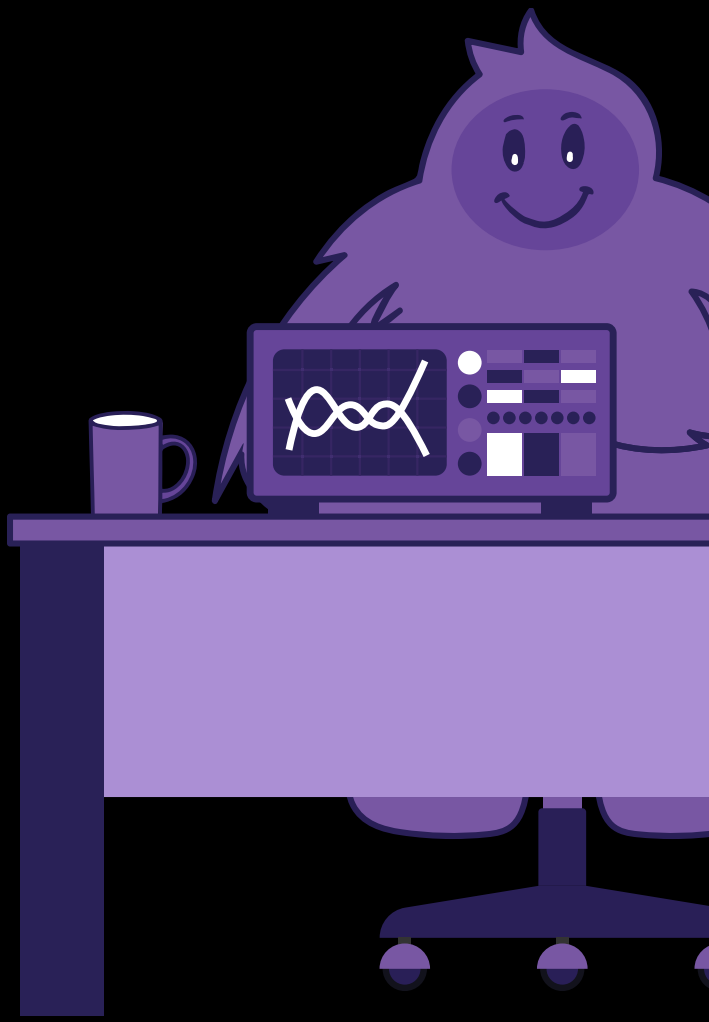
Dictionaries

- Dictionaries are another really useful way of storing multiple bits of data
- We store values in a dictionary along with an associated value, a key, that we can use to access that value later on
- Dictionaries can be declared using *curly brackets { }*, let's look at an example:

my_dictionary.py > ...

```
1    my_dictionary = {"key_one": "value one", 2: "value two", "key_three": 3}
```

- Here, we declare a dictionary "my_dictionary" with three key-value pairs
- Declaring a dictionary is done with the following syntax:
  dictionary = { <key>: <value>, <key 2>: <value 2>, … }
- Since we use *keys* to look up values in the dictionary, each key **must** be unique
- We can use any data type as a key so long as it is *hashable*, explaining this fully is outside the scope of this tutorial, but for now just know that a *hashable* type is a data type the interpreter can verify the *uniqueness* of
- In practice, this means we can use strings, ints and even floats as keys!



Storing and Retrieving Data

- We can store and retrieve data from dictionaries using *square brackets*
- The format is pretty similar to that used with lists, we just use our key values instead of indexes!

my_dictionary.py > ...

```
1   my_dictionary = {"key_one": "value one", 2: "value two", "key_three": 3}
2
3   print(my_dictionary["key_one"])
4
5   my_dictionary["key_one"] = "new value"
6
7   print(my_dictionary["key_one"])
```

- We access dictionary values using the following syntax:
  dictionary[<key value>]
- In this code, we access the value associated with the key 'key_one' and print it to the screen. Then, we associate/assign this key a new value, "new value", before displaying it to the screen again
- Note that we can use the assignment operator '=' to add new key/value pairs to a dictionary!

```
9   my_dictionary["new key"] = "even newer value"
10  print(my_dictionary["new key"])
```

Useful Dictionary Methods

- Python dictionaries have several useful in-built methods
- Here's a short list of a few useful functions:

Method	Explanation
keys()	Return a list of all keys in a dictionary
values()	Return a list of all values in a dictionary
items()	Return a list of all key-value pairs in a dictionary
pop()	Remove a matching key-value pair from a dictionary
clear()	Remove all key-value pairs from a dictionary
update()	Add a new key-value to the dictionary
get()	Get a value from a dictionary given a key

- A full list can be found here:
  https://www.w3schools.com/python/python_ref_dictionary.asp

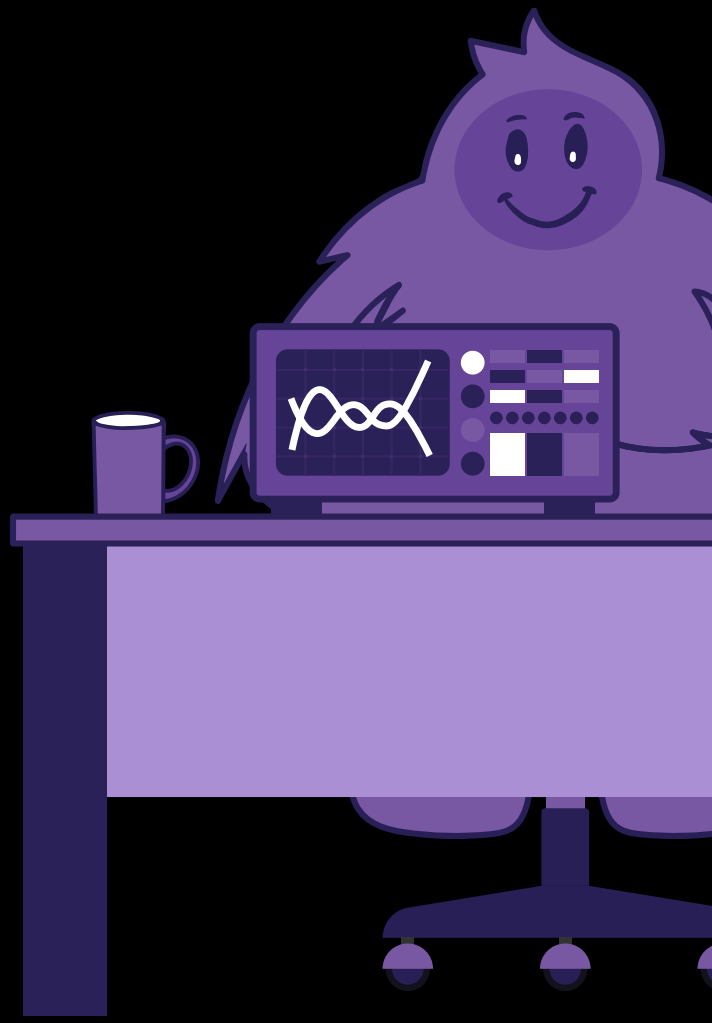
Challenge Time!

1. Create a program that stores a unique message for every day of the week in a dictionary. This program should then ask the user what day of the week it is and print the associated message

2. Modify this program so that different “moods” of message are available for each day, e.g. happy messages, grumpy messages, etc.. Store these messages in a *dictionary that contains dictionaries!* The program should ask the user for their mood and for the day of the week before displaying the correct message to the screen…

```
What's your mood? happy
What day is it? thursday
awesome, it's almost friday!
```

```
What's your mood? angry
What day is it? thursday
can't this week be over already? >:(
```

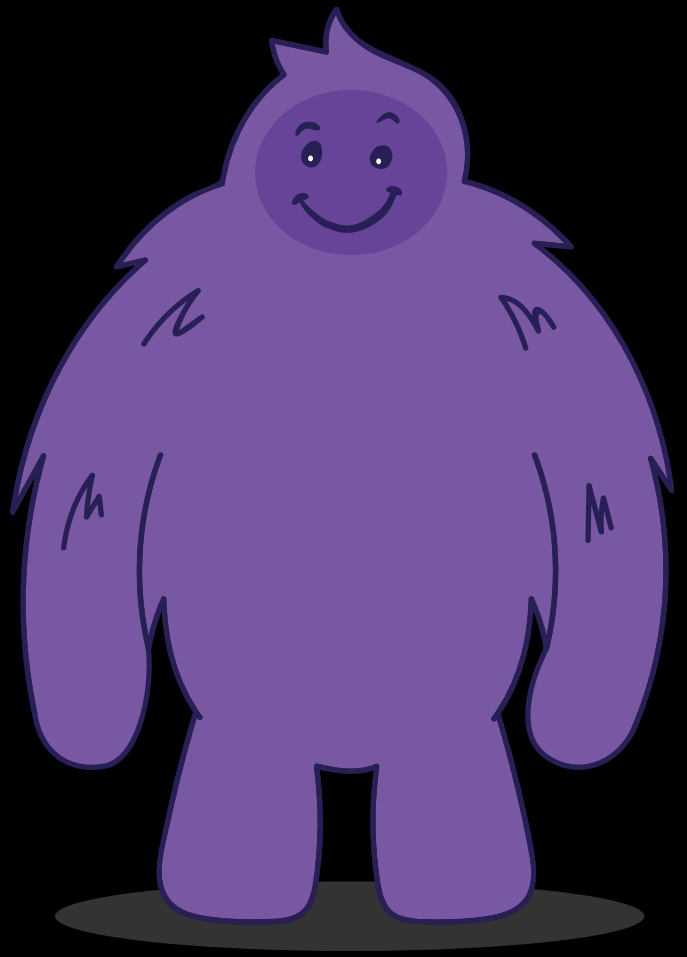


Loops

INTERRUPT LABS

Going Loopy

- Sometimes, we'll want to repeat code a set number of times, repeat until a condition is met or carry out some task for each element in a list. We use *loops* to accomplish this
- Python has two types of loops we can use, `while` loops and `for` loops
- `while` loops are particularly useful when we want to repeat code until a condition is met, `for` loops are useful when working with *iterables*



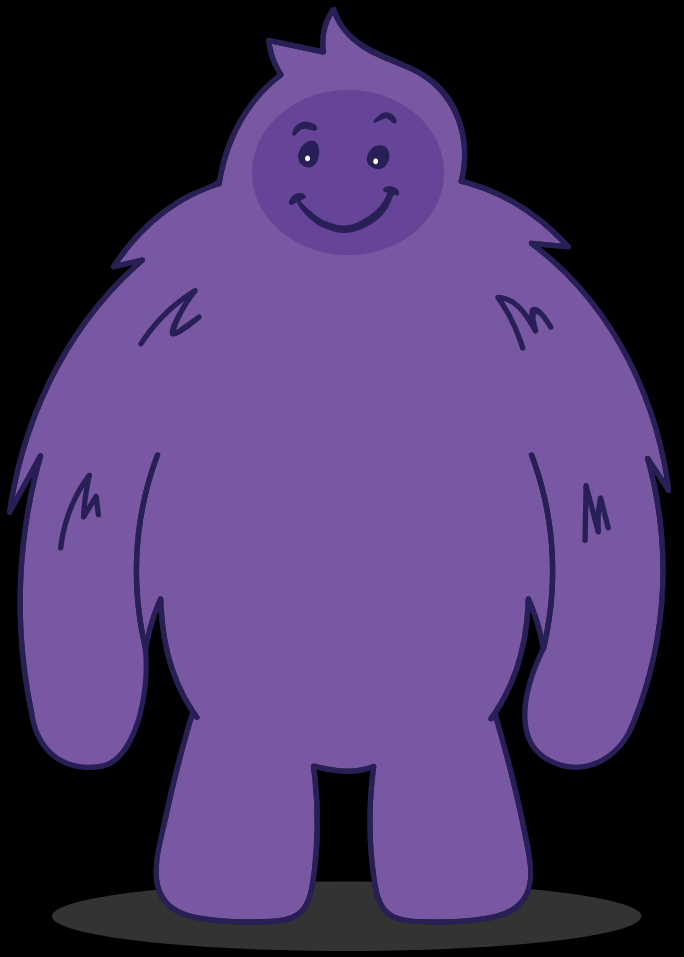
While Loops

- `while` loops are the simplest kind of loop. They repeat a section of code so long as a condition is true…

```
1  count = 0
2  while count < 10:
3      print("Hello")
4      count += 1
5
6  print("Finished!")
```

- In this example, the `while` loop will check if the variable count is less than 10. If so, lines 3 and 4 will execute before the `while` loop begins again
- When count is no longer less than 10, the while loops condition becomes false, and the loop stops repeating. Lines 3 and 4 are skipped and the program resumes execution on line 6

- Try copying and running this code, what happens when you change the `while` loops condition? What happens if the condition is always true?



Breaking Free

- Sometimes we'll want to exit a loop early or under some special circumstance
- We can accomplish this using the `break` keyword
- When Python hits the `break` keyword, it will immediately exit whatever loop it's currently in and resume execution at the next non-loop line of code
- The `break` keyword works in both `for` and `while` loops but throws and error if used outside of a loop!

```
1  count = 0
2  while count < 10:
3      print("Hello")
4      if count == 5:
5          break
6      count += 1
7
8  print("Finished!")
```

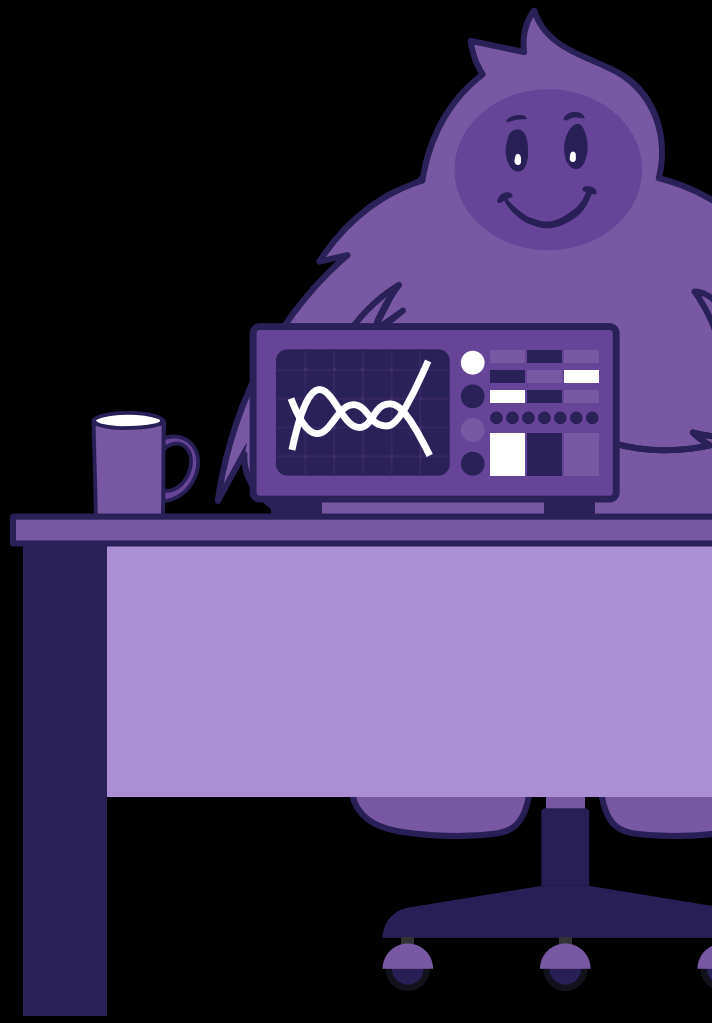
```
1  count = 0
2  while count < 10:
3      break
4      print("Hello")
5      count += 1
6
7  print("Finished!")
```

- How many times will these loops print "Hello" to the screen?

Challenge Time!

1. Write a program that takes two inputs from the user, an initial message and a number. Then, use a while loop to display this message to the screen that number of times

2. Write a program that repeatedly takes numbers as input from the user, only stopping when the sum of these numbers is greater than 100

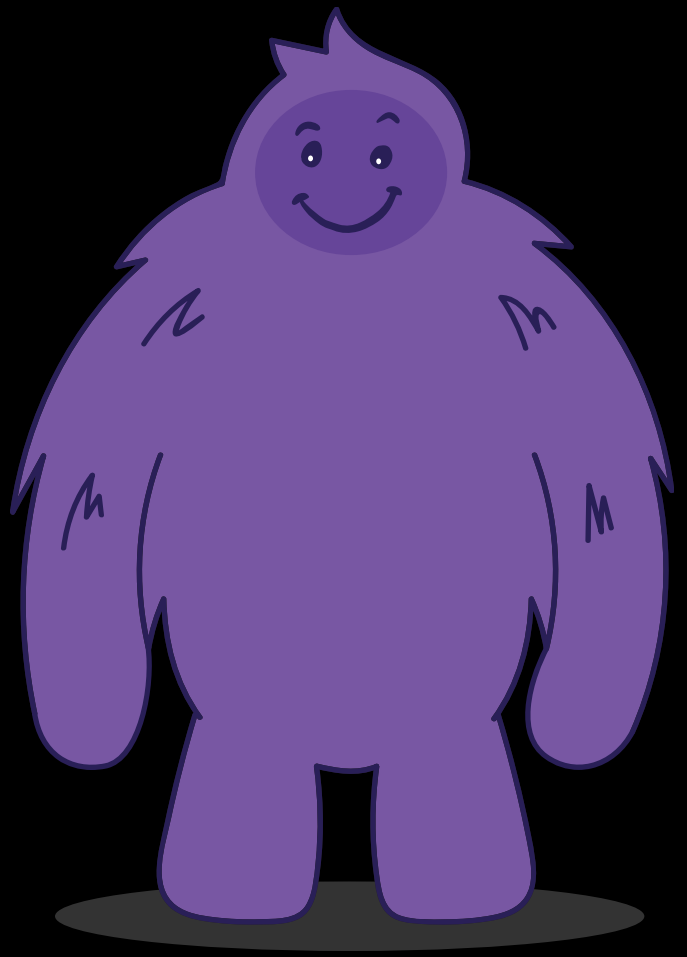


May the For-ce be With You

- *For each member of iterable X, do Y…*
- We use `for` loops when working with iterables
- An iterable is any type/object that is capable of returning its values *one chunk at a time*
- Lists, tuples, dictionaries and ranges are all examples of iterables
- A `for` loop will run its associated code *for each member* of an iterable
- Let's look at an example:

for_example.py > ...

```
1   my_list = [1, 2, 3, 4, 5]
2
3   for i in my_list:
4       print(i ** 2)
5
6   print("Finished!")
```



- The `for` loop on line 3 *iterates* through 'my_list', repeating for each value in the list. The variable *i* is assigned to each value of the list as the loop repeats. In this case *i* is first set to 1, then 2, then 3 etc…
- Note that we can give our temporary value any valid name, through *i, j, x, y, z* are common choices
- When no more values can be drawn from the list, the loop completes and line 6 runs

More Examples, so You Don’t For-get ;)

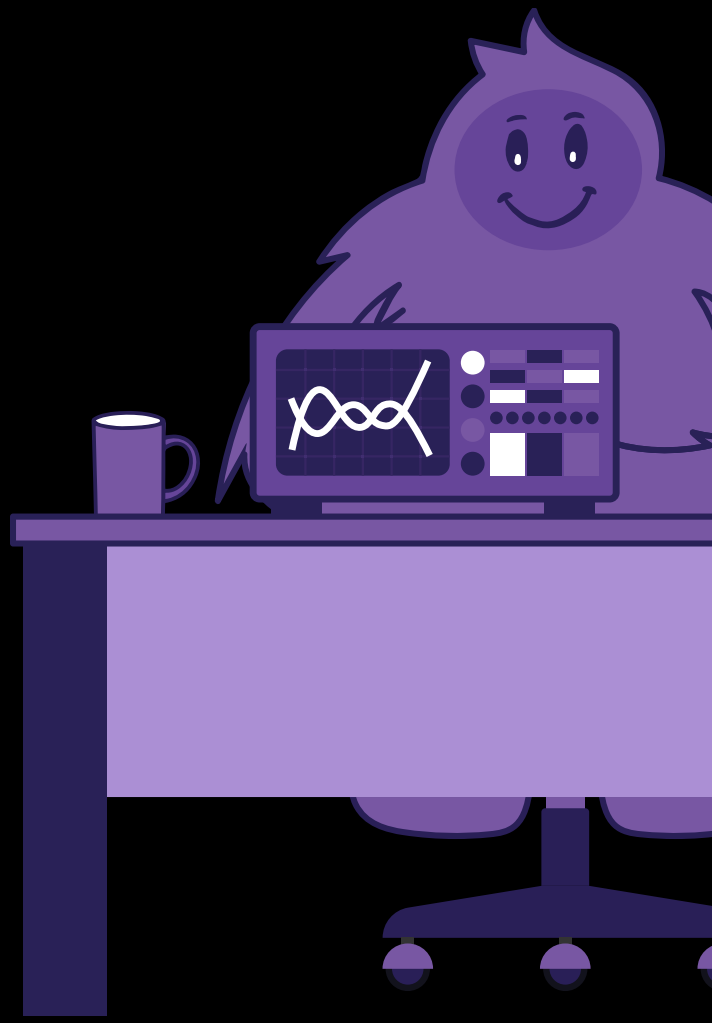
- Let’s hammer the point home with some more `for` loops!

more_for_loops.py > ...

```
1   for i in range(10):
2       print(i)
3
4   my_dictionary = {"key one": "value 1", "key two": "value two"}
5
6   for i in my_dictionary:
7       print(my_dictionary[i])
8
9   my_tuple = (1, 2, 3, 4)
10  for i in my_tuple: print("Value: " + str(i))
```

- Try copying and running this code, what values are being assigned to `i` when we iterate through `my_dictionary`?

- As you can see on line 10, sometimes we can write `for` loops on a single line to make them more compact. We can do this when the code we want to repeat is only a single line. You can also use this trick on `if` statements that don’t have any `elif`/`else` statements!

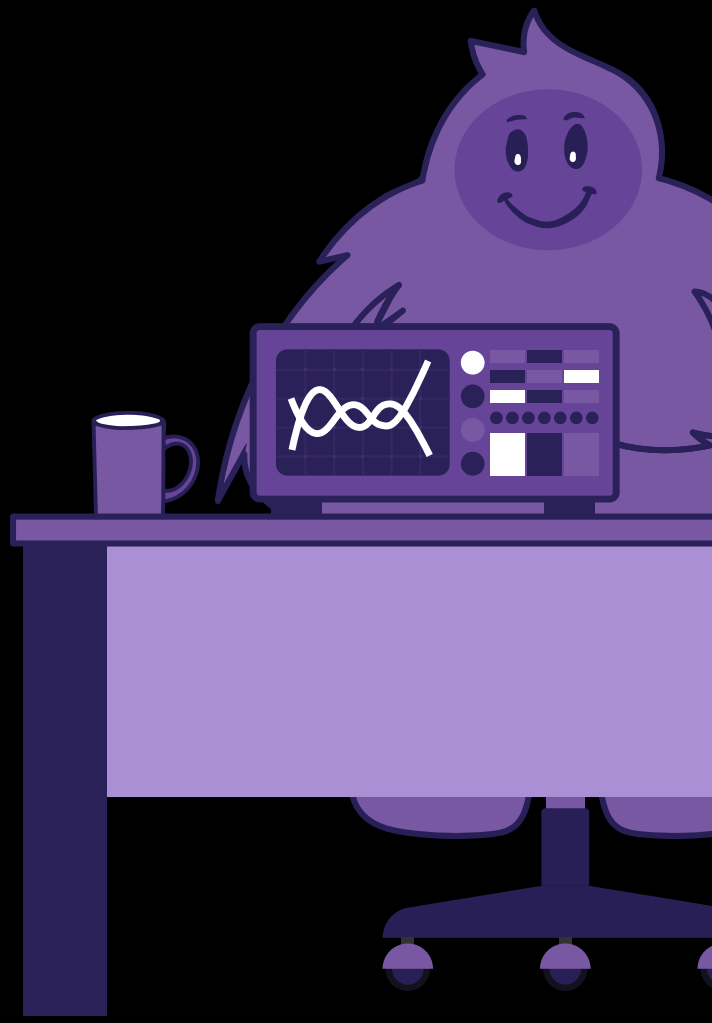


A Challenge *for* You!

1. Write a program that uses a for loop to iterate through all numbers between 1 and 100, appending all numbers divisible by a user-supplied integer value to a list

2. Write a program uses a for loop to store ten user-supplied integers to a list. This program should then use a second for loop to determine which of these values is the greatest

```
>>: 0
>>: 100
>>: 150
>>: 3
>>: 4
>>: 5
>>: 20
>>: 11
>>: 19
>>: 30
Largest value is: 150
```



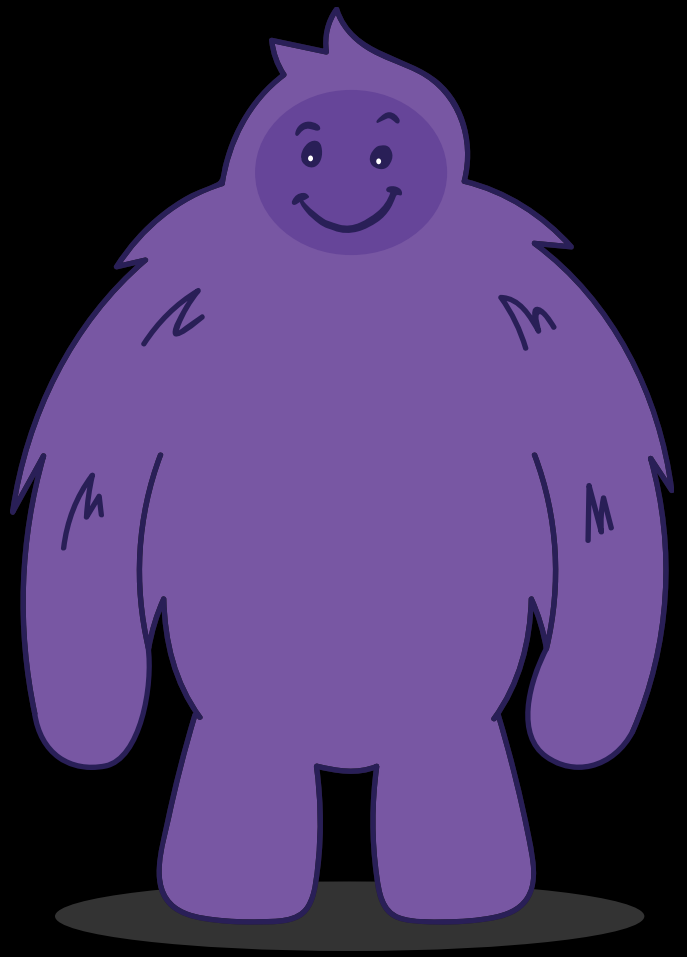
INTERRUPT LABS

One More Thing Be-for-e We Move On!

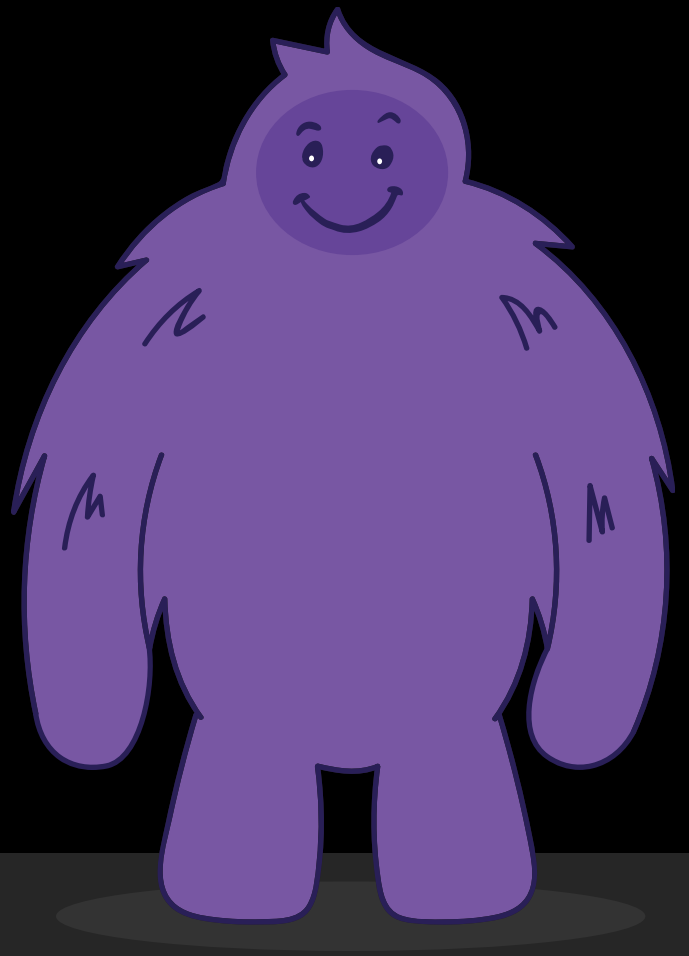
- Some iterables will return *multiple values at a time!*
- Imagine a list of co-ordinates, we'll have to deal with an X and a Y value on each loop
- If we know how many values we'll be dealing with, we can use *unpacking* to assign multiple temporary values in each iteration

```
>>> my_coords = [(1, 3), (4, 5), (6, 7)]
>>> for x, y in my_coords:
...     print(str(x) + " : " + str(y))
...
1 : 3
4 : 5
6 : 7
```

```
>>> more_coords = [(1, 2, 3), (3, 1, 3), (7, 4, 7)]
>>> for x, y, z in more_coords:
...     print(str(x) + " : " + str(y) + " : " + str(z))
...
1 : 2 : 3
3 : 1 : 3
7 : 4 : 7
```



Random-ness



The *random* Module

- All our programs so far have been *deterministic*, that is to say that given the same inputs they'll do the same thing each and every time…
- We can use the *random* module to introduce *non-deterministic (random)* behaviors to our programs! This module provides many in-built methods to support operations on lists, ranges, numbers, bytes, etc.
- You can find the full documentation here:
  https://docs.python.org/3/library/random.html

- [Note]: This library provides us with *pseudo-random* numbers, this means that the *random* module is not random enough to be secure when used for cryptographic purposes!

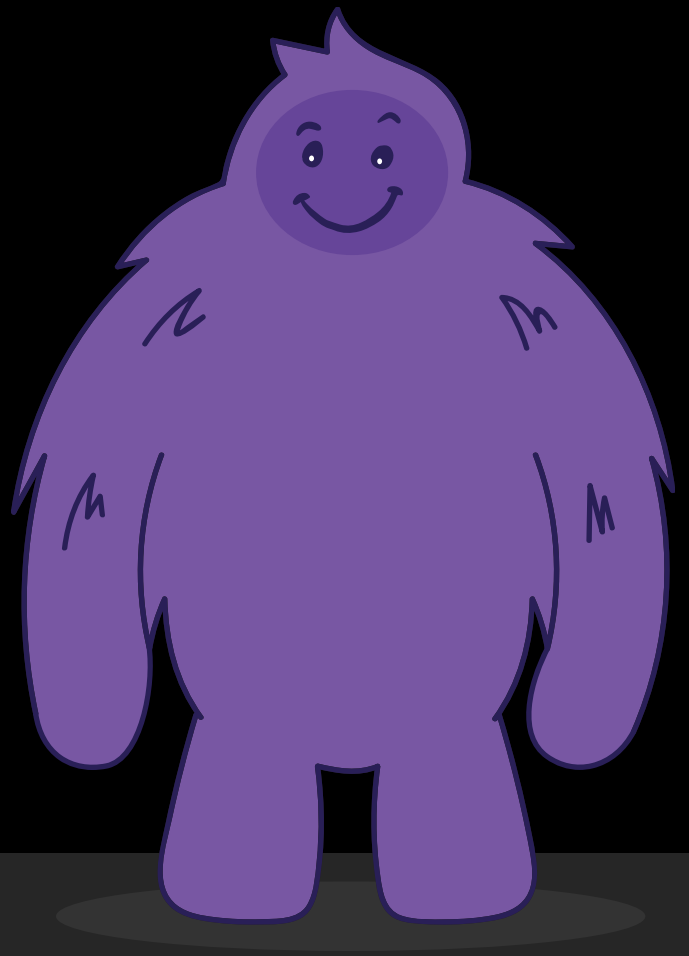
Some Useful Functions

- Here's some examples of *random*'s functionality…
- Remember to view the documentation to get the full picture!

randomness.py > ...

```python
# Import the random module
import random

# Pick a random integer between 10 and 20
x = random.randint(10, 20)

# Pick a random integer from a range
y = random.randrange(0, 100, 2)

# Generate some random bytes
z = random.randbytes(50)

# Pick a random element from a list
my_list = ["Shrek", "Fiona", "Donkey", "Farquad"]
my_choice = random.choice(my_list)

# Pick multiple random elements from a list
my_choices = random.choices(my_list, k=2)

# Pick a random floating-point number from a gaussian distribution (bell curve)
mean = 10
standard_deviation = 3
a = random.gauss(10, 3)
```

Working with Numbers



Basic Operations

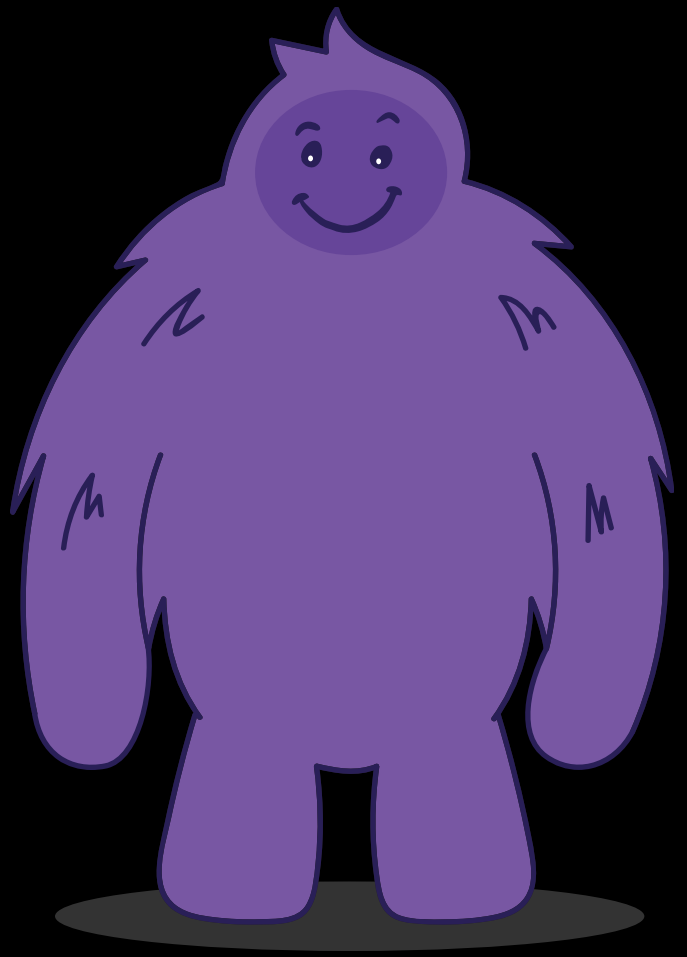
- Numbers can be added, subtracted, multiplied together etc. pretty easily using operators. However we should make a note of some quirks of these simple operations

- When we use two ints, the result will usually be another int. However, dividing one int by another will produce a float

```
>>> x = 10
>>> y = 2
>>> z = x / y
>>> type(z)
<class 'float'>
```

- When we use an int and a float, the resulting value will be a float

```
>>> x = 10
>>> y = 5.5
>>> z = x + y
>>> type(z)
<class 'float'>
```



Hex and Binary Numbers

- You're likely to encounter a lot of numbers represented using binary and hexadecimal notation; thankfully, Python provides easy ways of working with both!
- We can assign integer variables using hex and binary notation by prefixing our hex/binary numbers with 0x/0b respectively

```
1   hex_value = 0xff
2   binary_value = 0b1010
3
4   print(hex_value)
5   print(binary_value)
```

- If you run the code above, you'll see Python prints out 255 and 10. This is because Python treats "hex_value" and "binary_value" as regular decimal int values once we've assigned them
- The conversion that happens here between hex/binary and decimal occurs implicitly, let's try converting some decimal values to hex and binary!

Converting Hex and Binary Numbers

- We can use the `hex()` and `bin()` methods that come with Python to convert integer values from decimal to hex and binary…

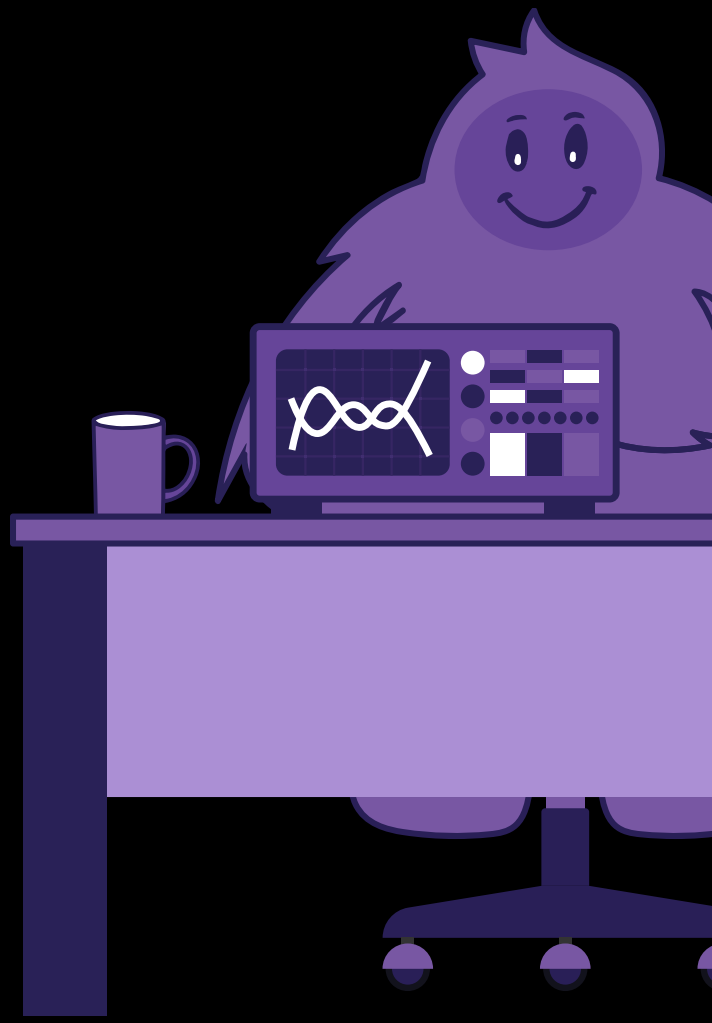
```
>>> number = 100
>>> hex(number)
'0x64'
>>> bin(number)
'0b1100100'
```

- Rather confusingly, `hex()` and `bin()` output strings
- Thankfully, we can *cast* these strings back to integers pretty easily!
- All we have to do is pass another argument to the `int()` method we used earlier! We provide the *base* of the number system we are converting from. Hexadecimal values are base-16. Binary values are base-2…

```
>>> number = 100
>>> hex_str = hex(number)
>>> bin_str = bin(number)
>>>
>>> print(int(hex_str, 16))
100
>>> print(int(bin_str, 2))
100
```

- We can also pass `int` a *base* of 0 if we want Python to take its best guess at what it should use…

```
>>> print(int(hex_str, 0))
100
```

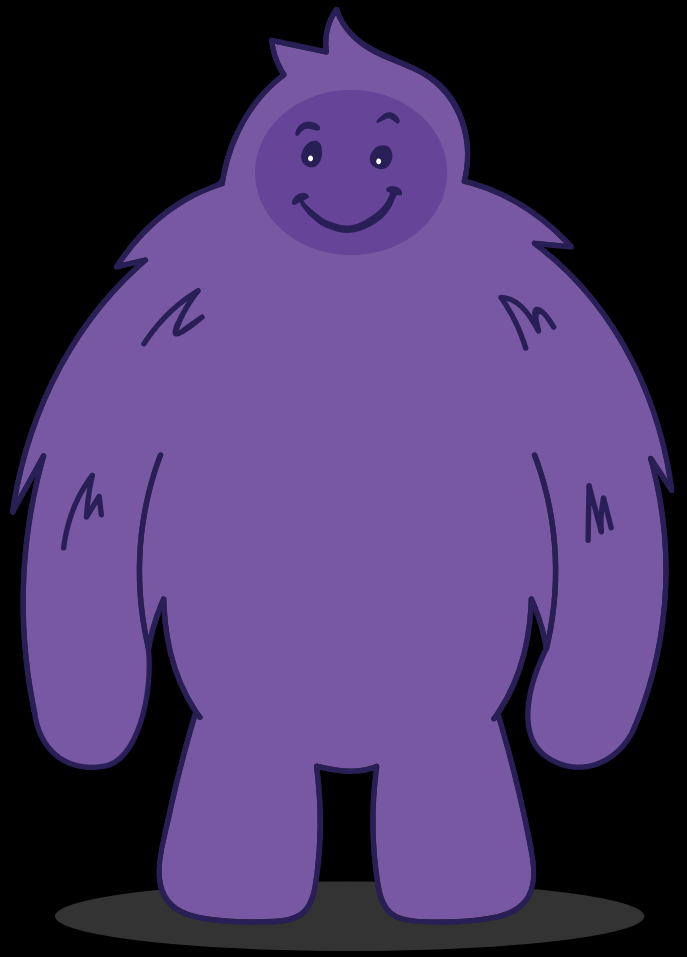


Using the *math* Library

- Sometimes we'll want to go beyond basic operators and start using things like trigonometric functions, complex numbers etc.
- Python includes the `math` library which defines these functions for us!
- To use the math library, we start off our Python script with an `import` statement followed by the name of the library. We'll cover imports in greater depth later on but for now just know that imports allow us to add extra functionality to our programs
- Once the library is imported, we access its functionality using the "." operator…

```
1  import math
2
3  angle = math.pi
4  cos_angle = math.cos(angle)
5
6  message = "cos of " + str(angle) + " is " + str(cos_angle)
7  print(message)
```

- If you find yourself using complex numbers, Python includes a `cmath` library which defines complex versions of these functions



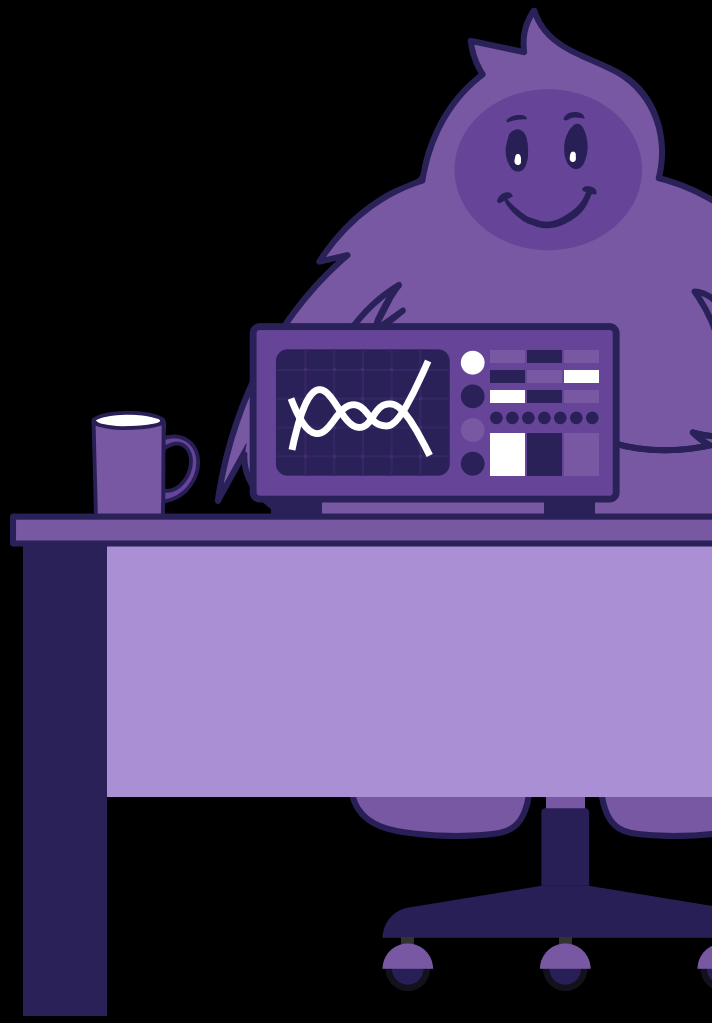
Challenge Time!

1. Use the math library to finish this program and calculate the length of the hypotenuse…

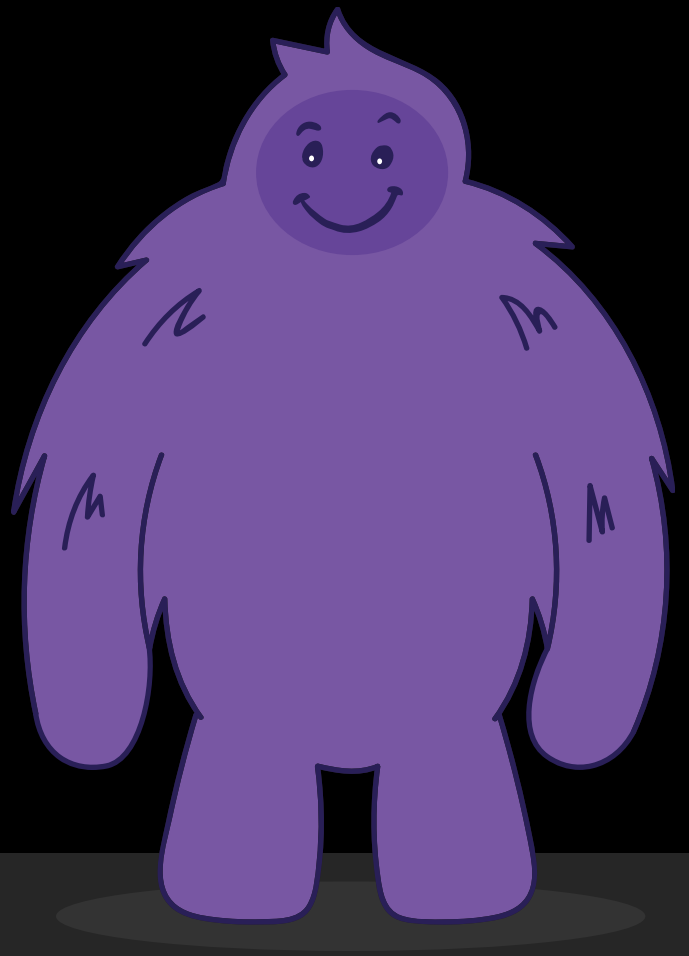
```python
import math

opposite_length = 10
adjacent_length = 20

hypotenuse_length = ????

print("The length of the hypotenuse is " + str(hypotenuse_length))
```

[Hint] Look at the *documentation* for the math library, are there any useful methods?
https://docs.python.org/3/library/math.html

2. Write a program to calculate the area of circle given a user-supplied diameter value



Working with Strings



Getting Stringy with it

- You'll find yourself using strings a ***lot.*** Fortunately, Python excels at handling strings, providing loads of default functionality for us to use
- Strings are really just lists of individual characters, we can index and splice into them the same way we would any other list!

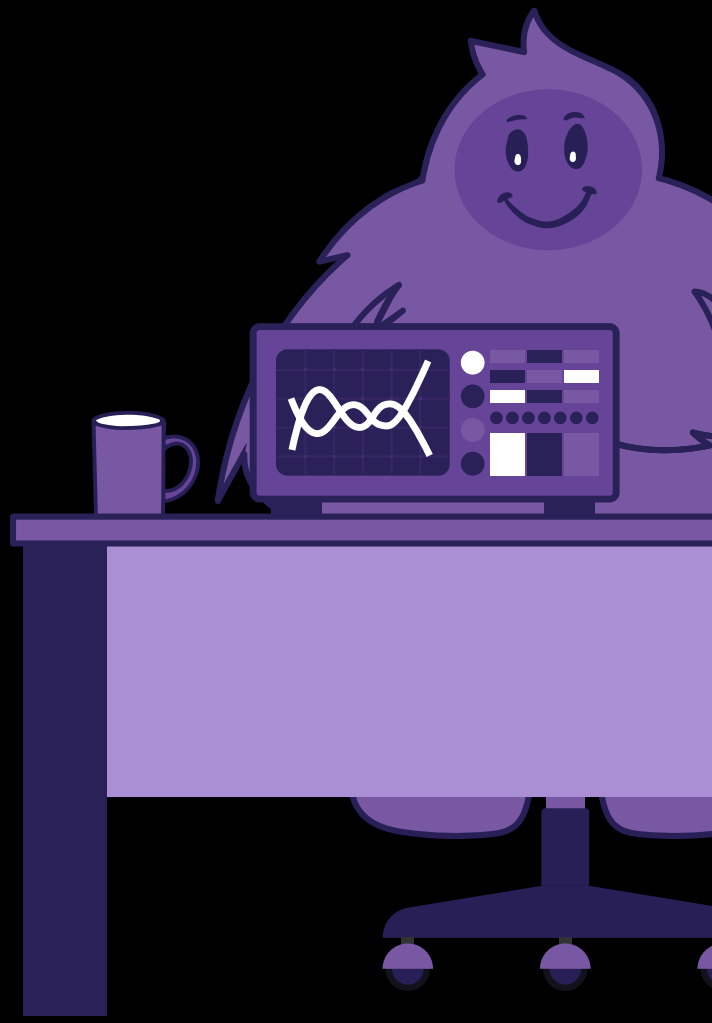
```
1   my_string = "Hello, World!"
2   print(my_string[:5])
3   print(my_string[::-1])
```

```
Hello
!dlroW ,olleH
```

- We can also *iterate* through strings as we would any other *iterable* type!

```
6   # Iterating through a string
7   for character in my_string:
8       print(character)
```

```
H
e
l
l
o
,

W
o
r
l
d
!
```



String Maths?

- We can use some arithmetic operators on strings, namely + and *
- + is used to *concatenate* strings, as you can likely guess, string concatenation is simply combining strings together

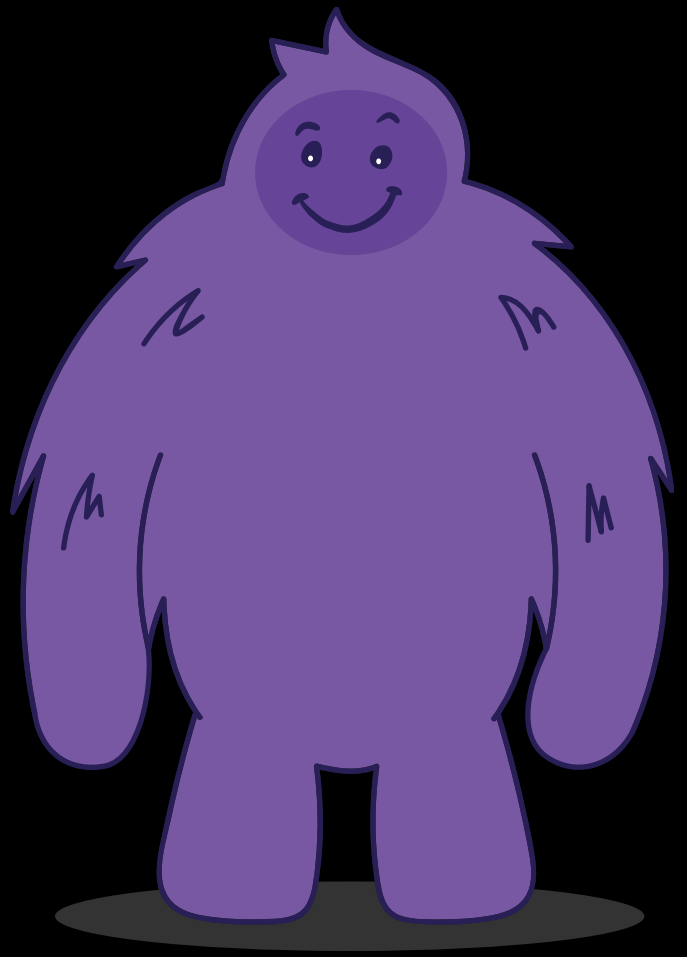
```
10   # Concatenation
11   print("Hello, " + "World!")
```

```
Hello, World!
```

- * is a simple way of repeating (*multiplying)* strings

```
13   # 'Multiplying' Strings
14   print("Hello, " + "World!"*10)
```

```
Hello, World!World!World!World!World!World!World!World!World!World!
```



Escape Characters

- We use escape characters to represent things like newlines, carriage returns, tab spacing and even alerts!
- We must use '\' to *escape* the string before the escape character can be displayed on screen
- This means that to print the actual character '\' we have to write '\\'!
- Here's a short list of common *special characters*:

Escape Character	Purpose
\n	Newline
\t	Tab
\r	Carriage Return
\a	Alert
\b	Backspace

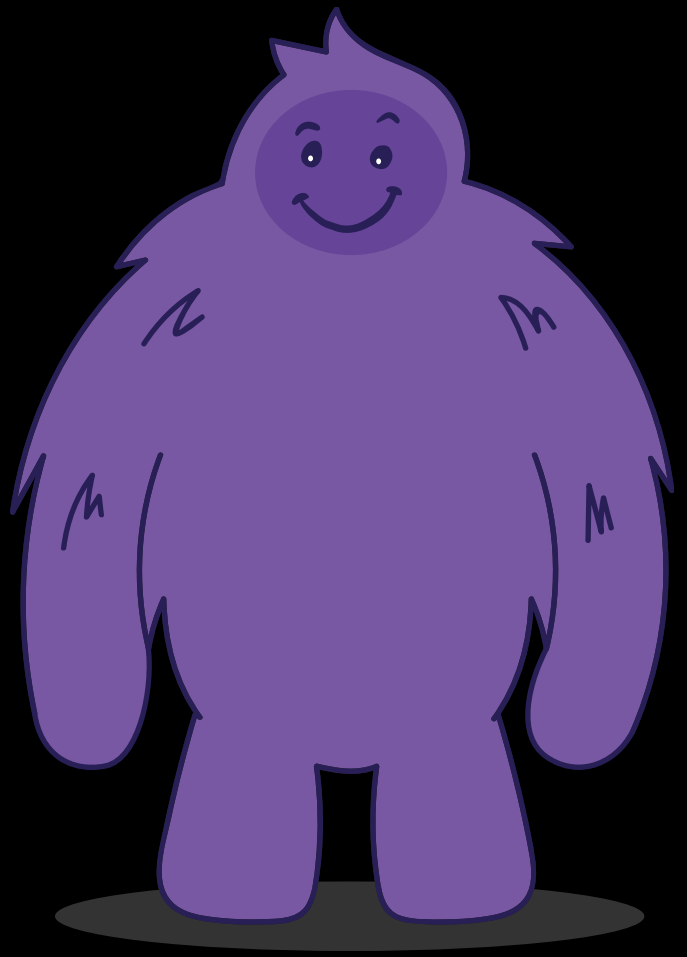
- A longer list of escape characters can be found here:
  https://en.wikipedia.org/wiki/Escape_character
- Try experimenting with escape characters by printing them to the screen and adding them to other strings!

Raw Strings

- By prefacing strings with the character ‘r’, we create a *raw string*
- This means that Python will ignore escape characters within this string, displaying it exactly as typed…

```
>>> raw_string = r"\t\t\t\t\tHello, World"
>>> print(raw_string)
\t\t\t\t\tHello, World
```

- Raw strings are particularly useful when dealing with data containing a lot of backslashes that might accidentally be interpreted as escape characters!



Format Strings

- Format strings or “f-strings” allow us to get really specific about how our strings are displayed

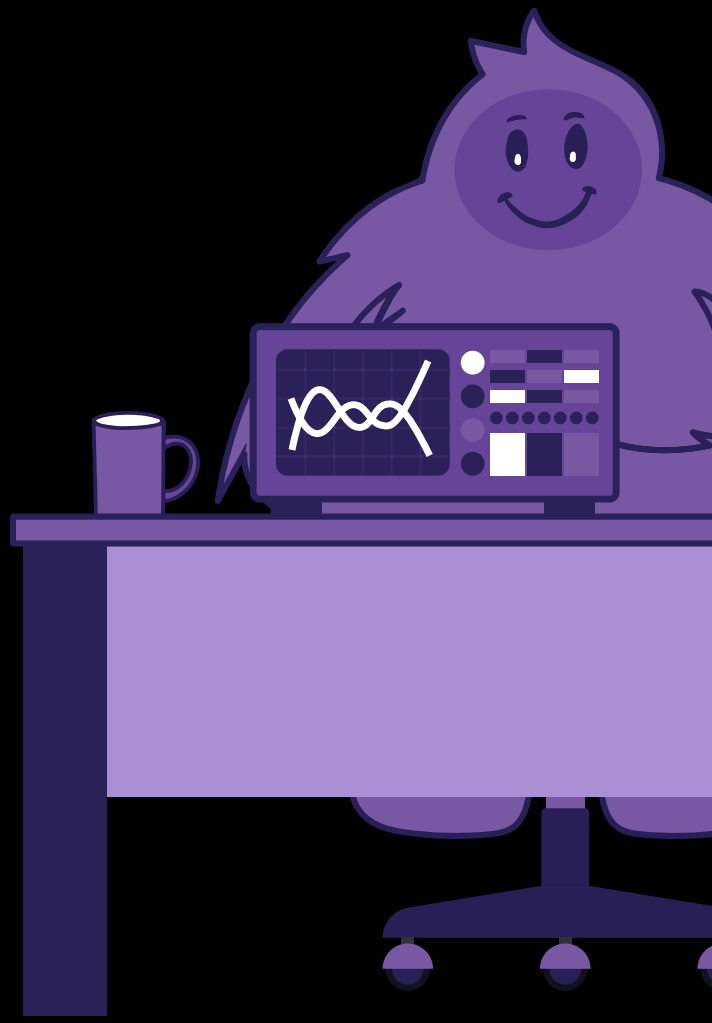
```
17  # Format strings
18  name = input("What is your name? ")
19  colour = input("What is your favourite colour? ")
20  message = "Hello {}! Your favourite colour is {}".format(name, colour)
21  print(message)
```

- The `.format()` method will replace the curly brackets in a string with the values given to it. This is done in order, so the first set of brackets is replaced by *name* and the second with *colour*
- We can be more concise and write values directly into the brackets using *format strings*
- We declare a format string by prepending the character ‘f’ to the front of a string

```
23  # F-strings
24  message = f"Hello {name}!"
25  print(message)
```

- Format strings even allow you to write code directly inside the brackets. So long as the resulting value can be represented as a string, it will automatically be placed into the final format string!

```
27  # Code in F-strings
28  print(f"One Hundred Squared is {100**2}")
```



Format Specifiers - Alignment

- *Format specifiers* are small bits of syntax we can include with format strings to specify how data is displayed on-screen
- We can use *format specifiers* to align our values when we display them
- Format specifiers are placed after a colon within the curly brackets of a format string, {value : format specifier}

- We can align values to the left using the < operator. The number following the < denotes how many spaces should be used to align the value.

```
>>> name = "YETI"
>>> message = f"# Hello {name:<20} #"
>>> print(message)
# Hello YETI                #
```

- We can center values using the ^ specifier

```
>>> message = f"# Hello {name:^20} #"
>>> print(message)
# Hello         YETI         #
```

- Values can be aligned to the right using the > specifier

```
>>> message = f"# Hello {name:>20} #"
>>> print(message)
# Hello                YETI #
```

Format Specifiers - Conversions

- When it comes to printing numbers, format specifiers can be used to convert between hex, binary, decimal and float!
- This can be done by introducing an `x`, `b`, `d`, or `f` character to our format specifier

```
1   my_number = 123
2
3   message = f"{'My number:':<20}{my_number:>20}"
4   binary_message = f"{'My number in binary:':<20}{my_number:>20b}"
5   hex_message = f"{'My number in hex:':<20}{my_number:>20x}"
6
7   print(message)
8   print(binary_message)
9   print(hex_message)
```

```
My number:                          123
My number in binary:               1111011
My number in hex:                       7b
```

- By using the # operator in our format-specifier, we can instruct python to display the type of data being displayed

```
3   message = f"{'My number:':<20}{my_number:>#20}"
4   binary_message = f"{'My number in binary:':<20}{my_number:>#20b}"
5   hex_message = f"{'My number in hex:':<20}{my_number:>#20x}"
```

```
My number:                          123
My number in binary:           0b1111011
My number in hex:                  0x7b
```

Format Specifiers - Floats

- Format specifiers can be used to display floats to *n* significant figures or decimal places (i.e. show *n* values after the decimal point)
- These format specifiers are incredibly useful as they correctly round the value to our desired length!

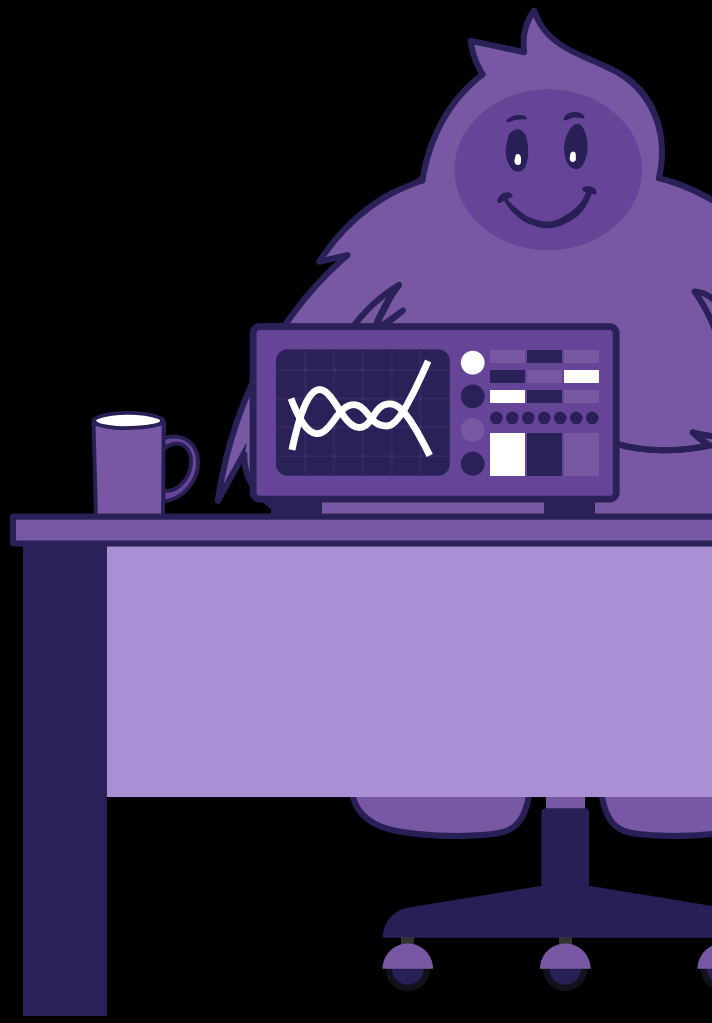
```
1   import math
2
3   two_decimal_places = f"Pi to 2 decimal places: {math.pi:.2f}"
4   twenty_decimal_places = f"Pi to 20 decimal places: {math.pi:.20f}"
5   two_sig_fig = f"Pi to 2 significant figures: {math.pi:.2}"
6
7   print(two_decimal_places)
8   print(twenty_decimal_places)
9   print(two_sig_fig)
```

```
Pi to 2 decimal places:  3.14
Pi to 20 decimal places:  3.14159265358979311600
Pi to 2 significant figures:  3.1
```

- We can combine the float specifier with the alignment specifiers as follows:

```
11    two_decimal_places = f"Pi to 2 decimal places: {math.pi:>20.2f}"
12    print(two_decimal_places)
```

```
Pi to 2 decimal places:                3.14
```



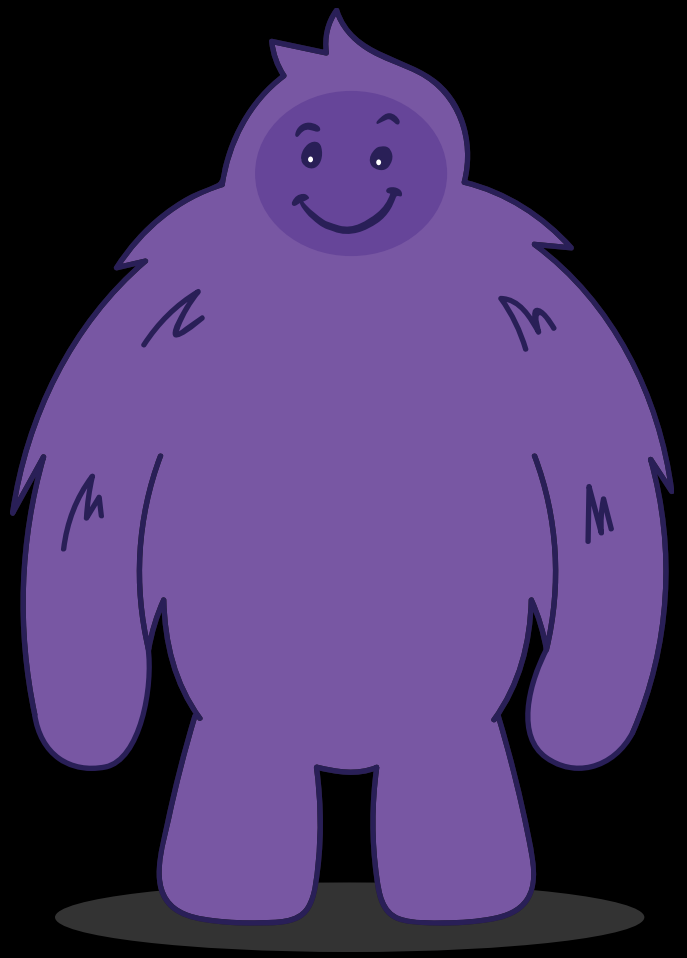
Out of Ord-er

- We can derive the Unicode value of a character by using the `ord()` method
- `ord()` only works on one character at a time…

```
>>> my_character = 'A'
>>> ord(my_character)
65
>>>
```

- The `chr()` method can be used to convert integers to their corresponding Unicode character

```
>>> chr(65)
'A'
>>>
```

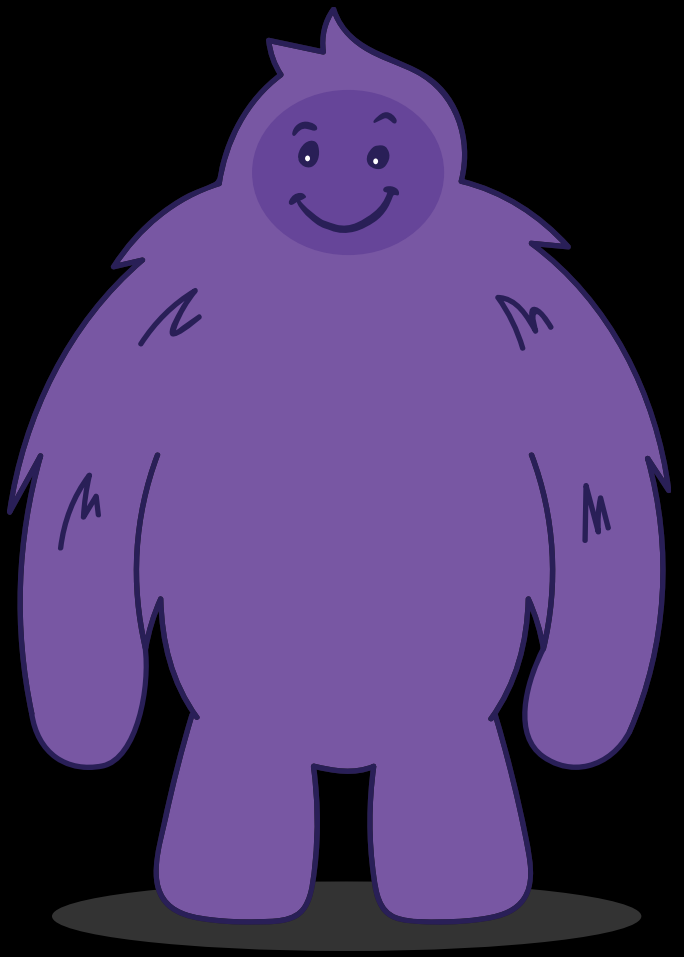


Useful Methods

- Python contains *a lot* of methods for working with strings, when you find yourself processing lots of strings, make sure to reference the documentation and look online. Don't spend time re-writing functions that are already included for you!

- You can find a big list of useful methods here:
  https://www.w3schools.com/python/python_ref_string.asp

- Open up the interpreter and try using some of these functions!
- You'll use strings a lot when scripting with Python, so being familiar with the available methods is *invaluable*!



Working with Files

Opening and Closing Files

- Python makes working with files super easy!
- Before we read/write to a file, we must open it, this can be done using the `open()` method
- The `open()` method takes two arguments, the *name* of the file being opened and the *access mode*
- The *access mode* tells python how we'll be using the file

- Here we'll open a file called *file.txt* for reading
- The `open()` method is given the name of the file *file.txt* and the *read access mode, "r"*
- After we're done with a file, we must close it so other programs can use it. Files are closed using the `close()` method

```
file = open("file.txt", "r")
# Do stuff
file.close()
```

- We can use a `with` statement to automatically close our files once done

```
with open("file.txt", "r") as file:
    # Do stuff
```

Access Modes

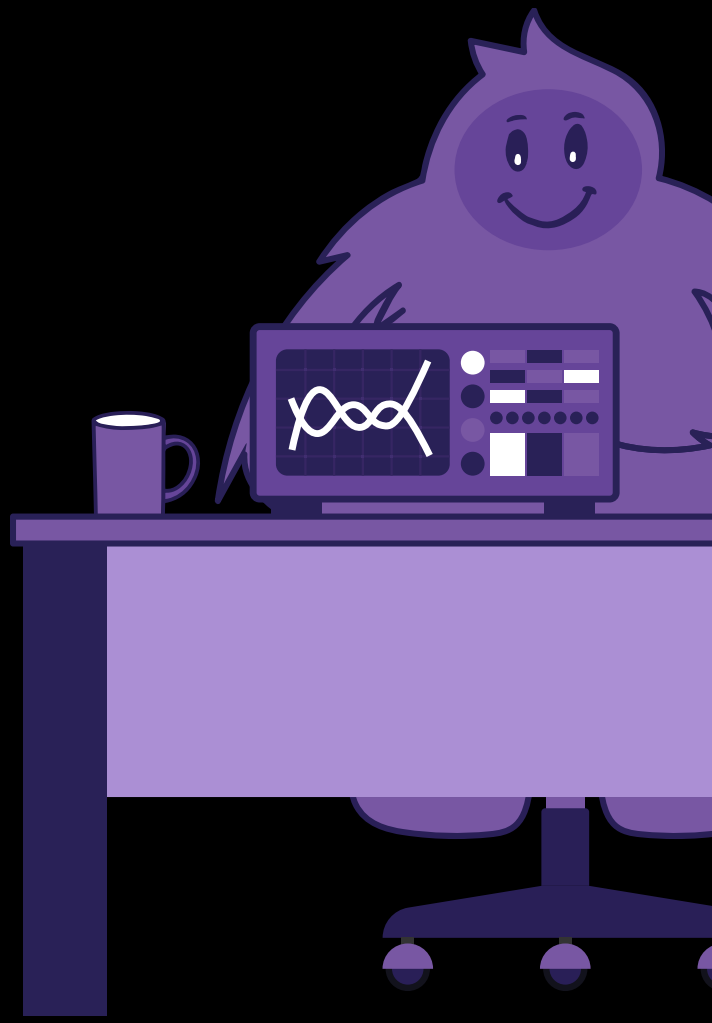
- Access modes are used when opening a file to tell Python how we'll be using the file
- Here's a handy table of all the access modes you can pass to `open()`:

Access mode	Description
"w"	Write mode, if the file doesn't exist, it'll be created
"r"	Read mode, if the file doesn't exist, an error will be thrown
"a"	Append mode, if the file doesn't exist, it'll be created
"w+"	Write + Read mode, if the file doesn't exist, it'll be created
"r+"	Read + Write mode, if the file doesn't exist, an error will be thrown
"a+"	Append + Read mode, if the file doesn't exist, it'll be created

Reading and Writing

- Python has some useful in-built methods for reading and writing to/from files!
- We can read/write character by character, all at once, or we can read/write line-by-line…

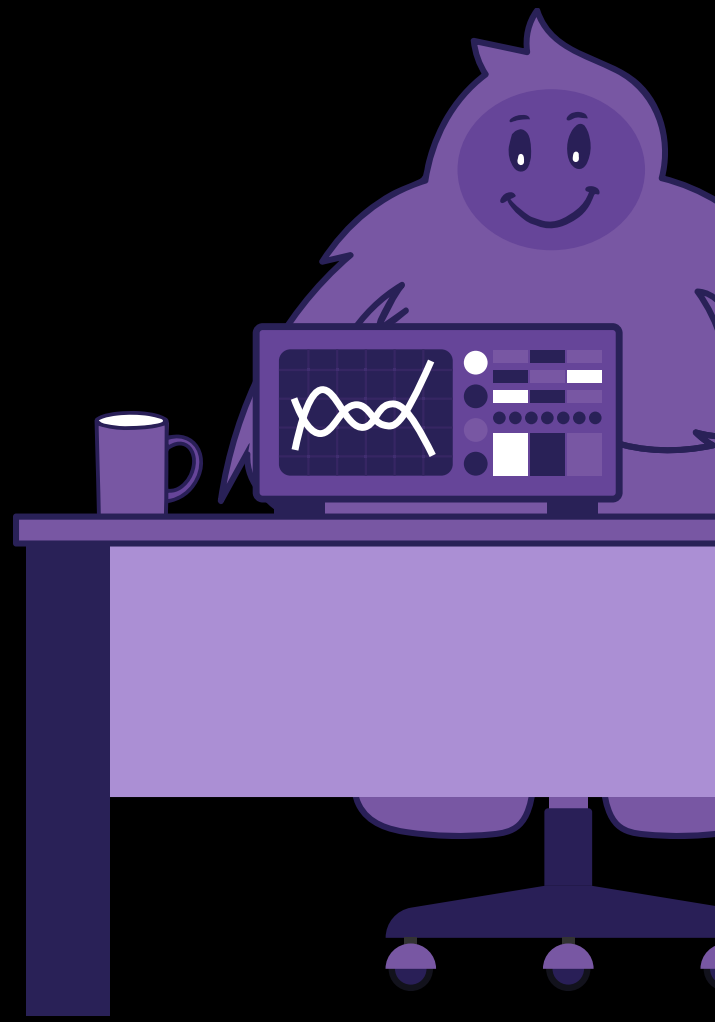
```python
file = open("read_file.txt", "r")

# Read everything
file.read()
# Read 20 characters
file.read(20)

# Read 10 lines
file.readlines(10)
# Read each line into a list, with every line being a member of the list
file.readlines()

file.close()


file = open("write_file", "w")

# Write a string to a file
file.write("Hello, world!")
# Write each member of a list to a file, with each member on a separate line
file.writelines(["line 1", "line 2", "line 3", "line 4"])

file.close()
```



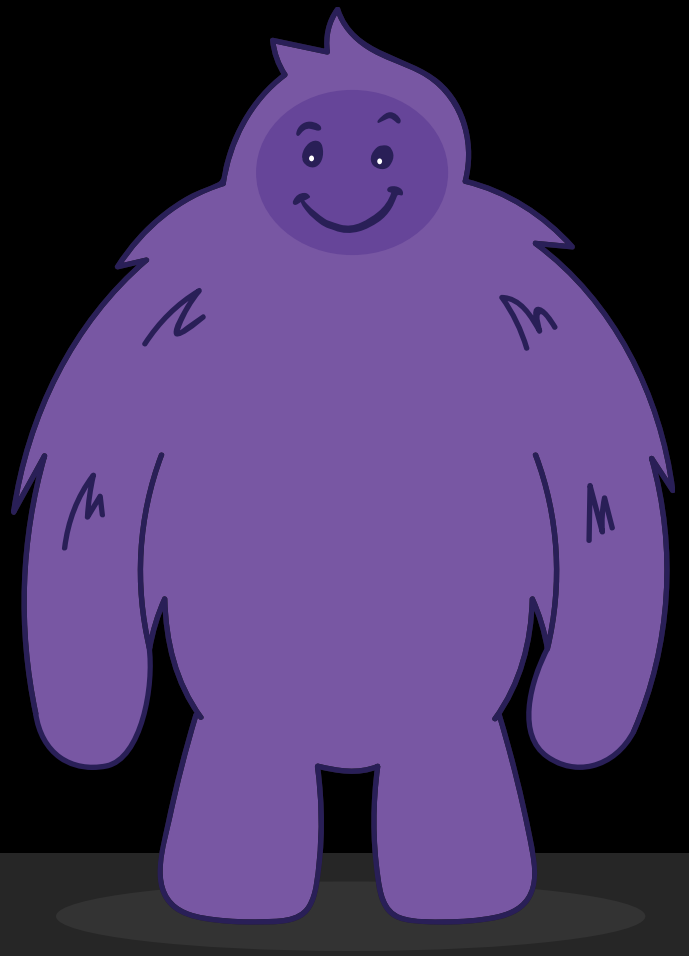
Binary Files

- By default, Python expects to be reading or writing text data to and from files, however you'll often want to read/write the *raw bytes* contained in a file
- This can be done by putting a "b" character next to the *access mode* we used to open the file…

```
1   # Write to a file
2   with open("byte_file.bin", "wb") as file:
3       file.write(b"\xca\xfe\x12\xab")
4
5   # Open a file
6   with open("byte_file.bin", "rb") as file:
7       # Read the first 10 bytes
8       x = file.read(10)
```



Functions



What is a Function?

- A function is just a way of defining code that can be *re-used.* We've actually been using functions throughout this tutorial, `print()`, `ord()` and `char()` are all functions!
- Functions take arguments/parameters and *return* a value
- Let's define a function

```
1   # Add numbers
2   def add_numbers(number_one, number_two):
3       total = number_one + number_two
4       return total
```

- We declare functions using the syntax:

def FUNCTION_NAME(argument_name_1, argument_name_2, …):
…code…
return value1, value2, …

- Functions can have as many arguments as necessary, they can also have zero arguments!
- Functions can return as many values as needed but return *None* by default, so if no return value is set, `None` will be returned!

```
def say_hello():
    print("Hello!")

say_hello()
x = say_hello()
print(x)
```

```
Hello!
None
```

Using Functions

- To use a function, we *call* it
- This is done by invoking the name of the function and specifying values for each of its arguments inside a pair of brackets

```
a = add_numbers(100, 102)
```

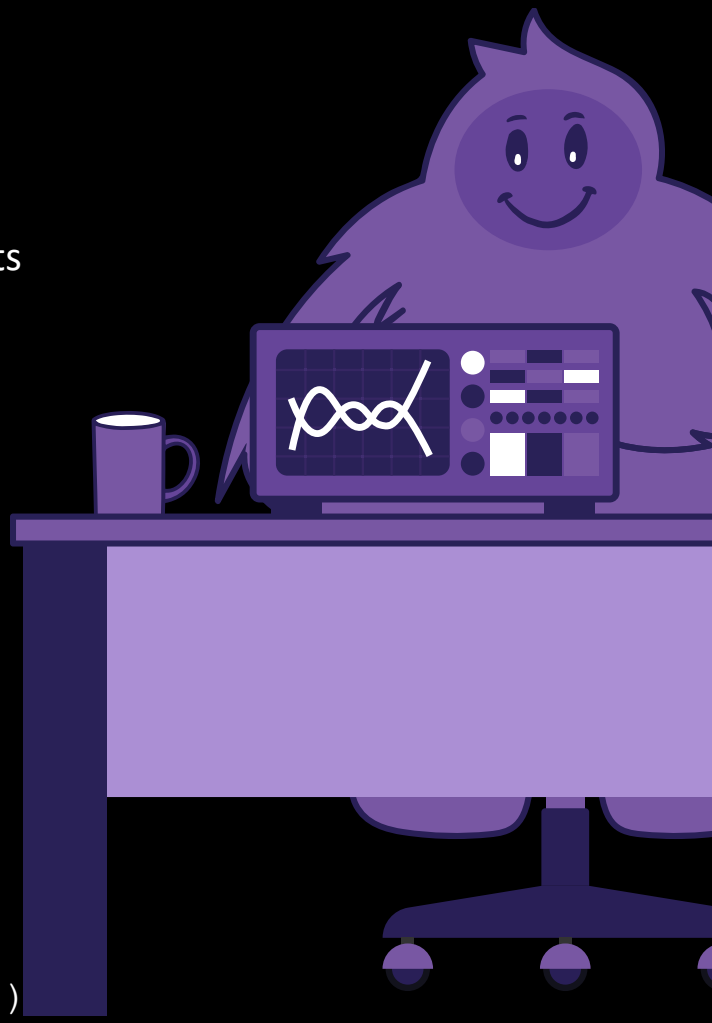
- Here, we call the function `add_numbers` with the arguments *100* and *102*, the return value is stored in variable `a`
- The arguments of `add_numbers` are assigned based on their order, with `argument_one` receiving the value of 100, and `argument_two` receiving the value of 102. We can assign values to function arguments directly too, using the = operator as we would with other variables…

```
a = add_numbers(number_one=100, number_two=102)
```

- We can even call functions using return values from other functions!

```
a = add_numbers(add_numbers(100, 100), 102)
```

- Here, argument `number_one` is assigned the return value of another call to `add_numbers()`



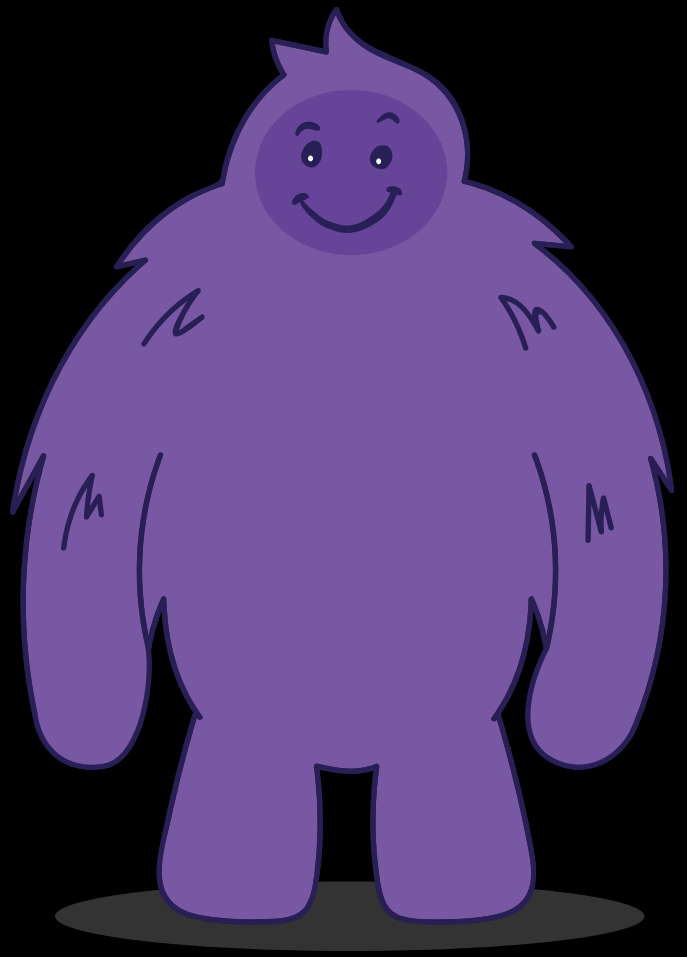
Arguments & Parameters

- Function arguments are sometimes called *parameters*
- Functions can have any number of associated parameters of any value
- Python uses *pass by assignment* to give parameters values. This means that parameters are assigned values in the same way as regular variables, the difference being that parameters don't exist *outside* of their associated function!

- Functions can give their parameters *default* values, which are assigned to values unless the caller explicitly gives the functions another value!

```
def show_message(message="Hello World!"):
    print(message)

show_message()
show_message("Goodbye World!")
```

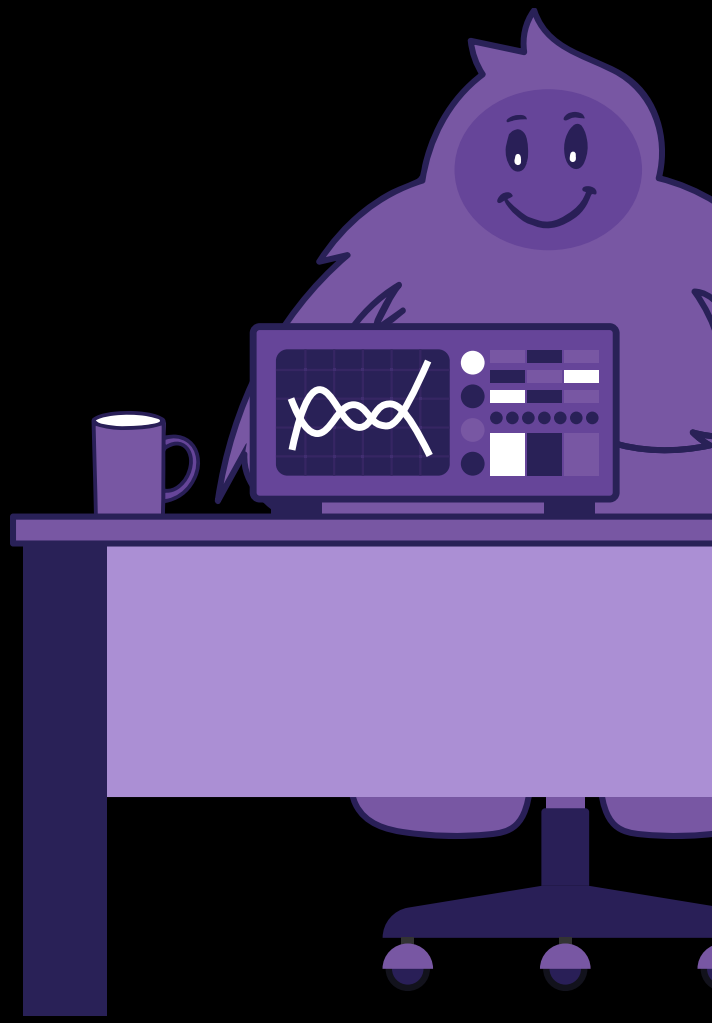
```
Hello World!
Goodbye World!
```



Challenge!

1. Write a function that calculates speed given distance and time, call it multiple times with different values and display the results

2. Write a function that returns a *True* value if a string contains the sub-string “QUIT”. Use this function to write a program that won’t quit until the user inputs a string containing the “QUIT” sub-string



Packing & Unpacking

- We can *unpack* values from a list to supply them as individual arguments to a function. Let's take an example function which has 3 parameters:

```
def print_info(name, age, hair_colour):
    print(f"Your name is {name}")
    print(f"You are {age} years old")
    print(f"You have {hair_colour} coloured hair")
```

- If we have a list that stores 3 values, we can *unpack* it using the *operator and call `print_info` without having to explicitly assign values to `name`, `age` and `hair_colour`!

```
values = ["Yeti", 100, "white"]
print_info(*values)
```

```
Your name is Yeti
You are 100 years old
You have white coloured hair
```

- We can also use the * operator in function definitions when we want to accept varying numbers of arguments! For example:

```
def add_together(*args):
    total = 0
    for i in args:
        total += i
    return total

print(add_together(1, 2, 3))
print(add_together(1, 2, 3, 4, 5, 6, 7, 8, 9))
print(add_together(*[1, 2, 3, 4]))
```

```
6
45
10
```

Unpacking Dictionaries

- In the example above, we unpack values and assign them to parameters based on their *position.* To be a bit more explicit, we can unpack *dictionaries* instead!
- To unpack a dictionary, we use the `**`syntax, let's try calling our `print_info` function from earlier

```
values = {"name": "Orangutan", "hair_colour": "Orange", "age": 10}
print_info(**values)
```

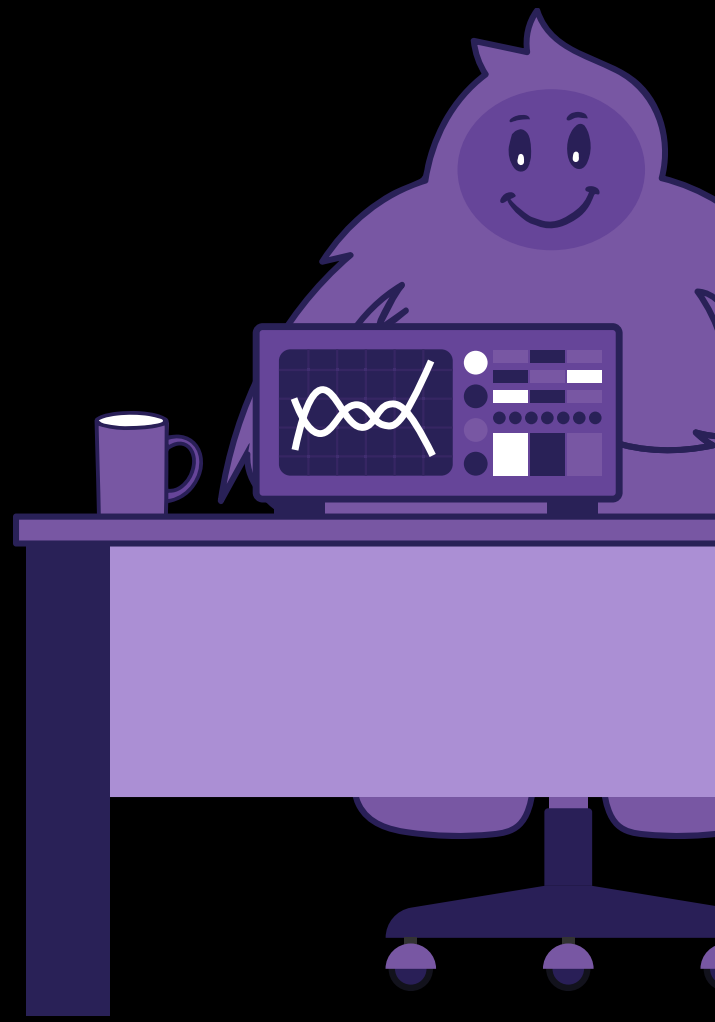
```
Your name is Orangutan
You are 10 years old
You have Orange coloured hair
```

- When a dictionary is unpacked, parameters are assigned the values associated with their matching *keys*, for example, the parameter name was assigned the value *Orangutan* as values[“name”] = “Orangutan”

- For dictionary unpacking to work, a functions parameters must be keys in the dictionary being used!

Scope

- Any variables assigned within a function belong to that functions *scope*, variables defined outside of a function belong to the *global* scope
- This means that they only exist *whilst their function is running*
- Let's look at an example:



- `scoped_var` is defined within the function, meaning we'll get an error when we try to access it outside of the function!

```
NameError: name scoped_var is not defined
```

```
def scope_example():
    scoped_var = 100

scope_example()
print(scoped_var)
```

- Here, when we access `scoped_var` we'll get the value 0 instead of 100! Even though `scoped_var` is re-assigned in function!

```
scoped_var = 0

def scope_example():
    scoped_var = 100

scope_example()
print(scoped_var)
```

Globals

- We can access variables defined outside of functions, yet we can't re-assign them, since that would only add them to our function's scope

```
scoped_var = 0
def scope_example():
    # Print the globally defined 'scoped_var'
    print(scoped_var)
```

```
def scope_example_2():
    # Assign a 'scoped_var' in the function scope
    scoped_var = 100
```

- If we want to change the values of functions defined in the *global* scope, we'll use Python's `global` keyword to tell the Interpreter that certain variables should remain as part of the global scope when we change them inside a function

- Here, we initially declare `scoped_var` globally and define *`scope_example_3()`*. This function assigns 100 to the globally defined variable *`scoped_var`* when called

  We print *`scoped_var`*, call *`scope_example_3()`* and then print `scoped_var` again, allowing us to confirm that calling `scope_example_3()` did change its value!

```
scoped_var = 0
def scope_example_3():
    global scoped_var
    scoped_var = 100

print(scoped_var)
scope_example_3()
print(scoped_var)
```

```
0
100
```

Func-ception

- We can define functions within functions!
- Just like before, these functions will only exist inside the *scope* of the original function! So this code will throw an error…

```
1  def outer_func():
2      def inner_func():
3          print("I'm the inner func!")
4
5  inner_func()
```

- Functions can also *return* other functions and take other functions as *arguments*! By returning `inner_func` we can store it in a variable and use it in the global scope!

```
1  def outer_func():
2      def inner_func():
3          print("I'm the inner func!")
4
5      return inner_func
6
7  x = outer_func()
8  x()
```

- Note that we return `inner_func` and not *`inner_func()`*. This would call `inner_func`, which returns None, and would return None from *`outer_func`*!
- We store the return value of `outer_func` in the variable x. Since this return value is a function, we can call x as we would any other function!

Func-ception 2

- When we define an inner function, it can access the values of variables defined in the outer function. This means that we can have variable function definitions!
- Let's look at an example…

```
1   def power_function_maker(power):
2       def power_function(value):
3           return value ** power
4       return power_function
5
6   square = power_function_maker(2)
7   cube = power_function_maker(3)
8
9   print(square(2))
10  print(cube(2))
```

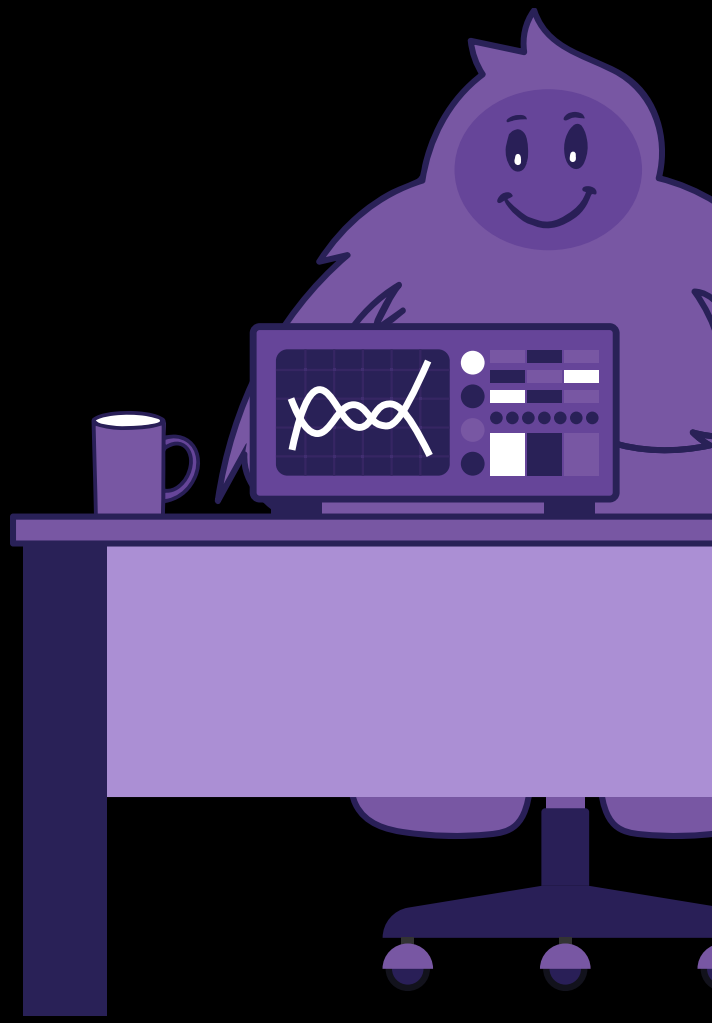
- Here, we define an outer function, `power_function_maker` which takes a single argument, `power`. *`power_function_maker`* defines an inner function *`power_function`* that raises a value to a power. Calling `power_function_maker` with an argument of 2, will return a function that raises a value to the power of 2
- We use `power_function_maker` twice to create functions for squaring and cubing, try copying the code above and see what functions you can create with *`power_function_maker`*

Challenge Time!

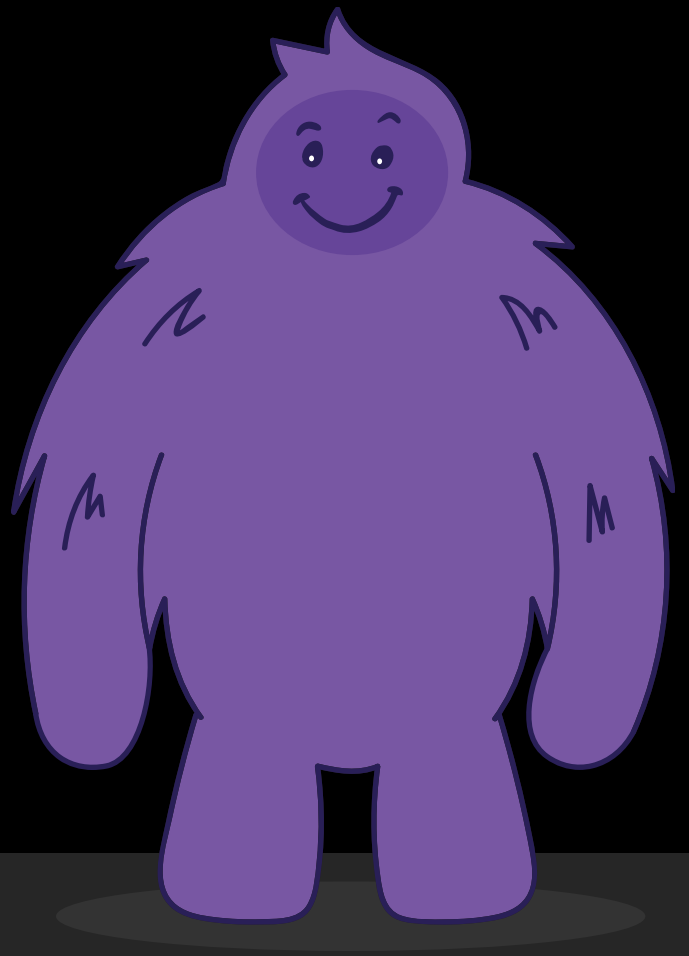
1. Use global values to write a function that will print out how many times it has been called

2. Write a function that takes a single number as an argument and returns a function that will repeat a given phrase that many times

```
>>> repeat_twice = repeater_function_maker(2)
>>> repeat_twice("Hello")
Hello
Hello
>>> repeat_thrice = repeater_function_maker(3)
>>> repeat_thrice("Goodbye")
Goodbye
Goodbye
Goodbye
```



Classes and Objects



Objects

- Objects represent collections of associated properties and behaviours. We use objects to encapsulate more complicated data types than our primitive strings, floats, lists, etc.

- Python objects are particularly useful at modelling real world objects. Think of a cow, it has properties, like its name, age, sex, weight, etc. and behaviours like mooing, eating grass, walking, breathing, etc.

- If we were to represent this cow in code, we could store these properties in variables and write functions that simulate each unique behaviour of the cow

- We use objects to represent pretty much everything in Python, each string, list, int, float, etc. you create is an object with associated properties and behaviours!

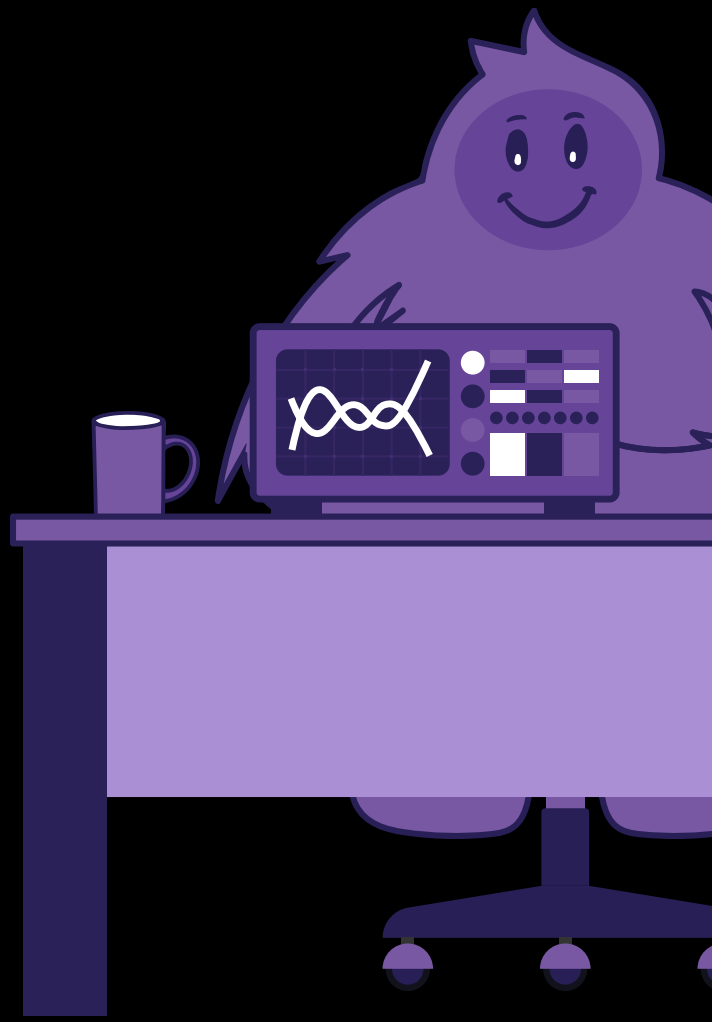
Classes

- In Python, we create objects based on definitions of *classes*
- Classes define the properties and methods that will be available to an object
- We can create many objects from one class, but each class must be unique
- Let's create a simple class for our cow example:

```
class Cow:
    name = "Daisy"
    age = 10
    sex = "Female"
    weight = 500

    def moo(self):
        print("moo")

    def eat_grass(self):
        print("CHOMP")

my_cow = Cow()
print(my_cow.name)
my_cow.moo()
```

- Here, we define a class, *cow*, using the *class* keyword. We then define some *class variables* and *class methods* for our *cow* class
- To use this class, we *instantiate* it, calling it like a function. This creates a *cow* object which gets stored in the *my_cow* variable
- This object is referred to as an *instance* of the *cow* class
- We access methods and properties of an object by using the . operator placed after the name of the object
- Using the . operator, we print out the name of *my_cow* and call its *moo()* method!

```
Daisy
moo
```



- Try copying this code and instantiating some more cows, do you notice any issues?

Being *Self*-ish

- As you may have noticed, if we create multiple cows, they'll all have the same name!
- This is because we defined *class variables* which are the same for each object created from a class! When we want variables to change per-instance of a class, we'll use *instance variables*

- To use instance variables, we define a special function *__init__*
- This function runs whenever we instantiate a class
- *__init__* has a special parameter, *self*
- *self* is a reference to the individual object that has been created from a class. Python passes *self* as an argument to each *class method* automatically. This means that we need to list it as an argument in the *moo()* and *eat_grass()* method definitions!

- When we instantiate objects from a class with an *__init__* method, we must provide values for each of its parameters!

```
class Cow:
    def __init__(self, name, age, sex, weight):
        self.name = name
        self.age = age
        self.sex = sex
        self.weight = weight

    def moo(self):
        print("moo")

    def eat_grass(self):
        print("CHOMP")

daisy = Cow("Daisy", 10, "Female", 500)
bob = Cow("Bob", 8, "Male", 510)
```

Using *self*

- *methods* can make use of the *self* variable to update the instance
- Let's upgrade our cows `eat_grass()` method to demonstrate…

- We'll declare an *attribute,* `hunger_level` and set it to 30
- Next, we modify `eat_grass` to reduce `hunger_level` each time it's called and stop the cow from eating when it's not hungry
- We access `hunger_level` from `eat_grass` as a property of the *self* parameter

- Try copying this new code and creating a couple of different cows

```
class Cow:
    def __init__(self, name, age, sex, weight):
        self.hunger_level = 30
        self.name = name
        self.age = age
        self.sex = sex
        self.weight = weight

    def moo(self):
        print("moo")

    def eat_grass(self):
        if self.hunger_level > 0:
            print("CHOMP")
            self.hunger_level -= 10
        else:
            print("NO MOO-RE")

daisy = Cow("Daisy", 10, "Female", 500)
daisy.eat_grass()
daisy.eat_grass()
daisy.eat_grass()
daisy.eat_grass()
```

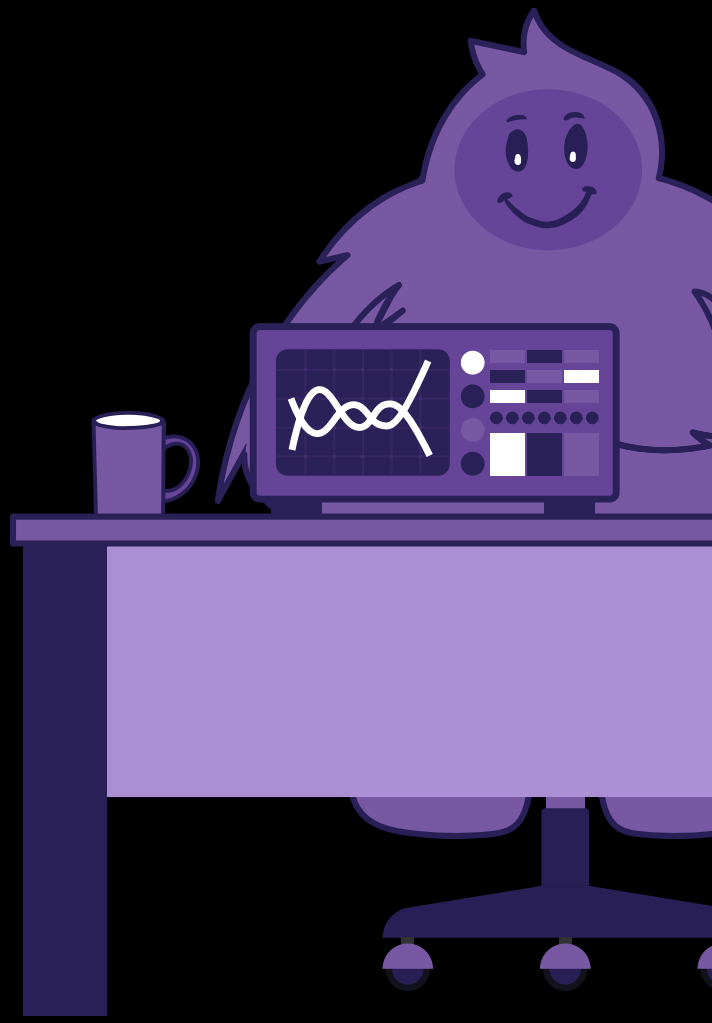
```
CHOMP
CHOMP
CHOMP
NO MOO-RE
```

Challenge

1. Create a class to represent a cat. This class should store information about each cat's name, age, happiness level and fur colour! Include a `pet()` function that increases the happiness level of each cat

2. Write a function that takes a *cat* object as a parameter and displays its information to the screen in addition to calling the pet() method

```
>>> jeb = cat("jeb", 5, "black")
>>> cat_info(jeb)
This cat's name is: jeb
This cat is 5 years old
This cat has black coloured fur
Just petted jeb!
```



Inheritance

- Sometimes we'll want to copy certain properties and methods from previously defined classes. We can do this using *inheritance*
- Let's look at an example, we'll write a class to represent a soldier. We'll define some properties like name and rank in addition to some common soldier-ey behaviours like marching and reporting for duty. Next, we'll use this class to define a new *commander* class….

- The class we defined, *Soldier*, is referred to as a *base* class or a *parent* class. This is because we'll write a class that *inherits* from it
- Next, we define our *Commander* class, specifying the *Soldier* class in a pair of brackets in the first line of our definition. This class is a *child* of the *Soldier* class
- We give our commander class a new method, `give_orders`, that won't be available to regular soldiers

- Finally, we create an instance of our *Commander* class, instantiating it as we would any other soldier
- We can call methods and access properties defined in the *Soldier* class in addition to calling the `give_orders()` function

```python
class Soldier:
    def __init__(self, name, rank) -> None:
        self.name = name
        self.rank = rank

    def report(self):
        print(f"{self.rank} {self.name} reporting for duty!")

    def march(self):
        print("LEFT RIGHT LEFT RIGHT LEFT...!")

class Commander(Soldier):
    def give_orders(self):
        print("FIRE!")

patton = Commander("Patton", "General")
patton.report()
patton.march()
patton.give_orders()
```

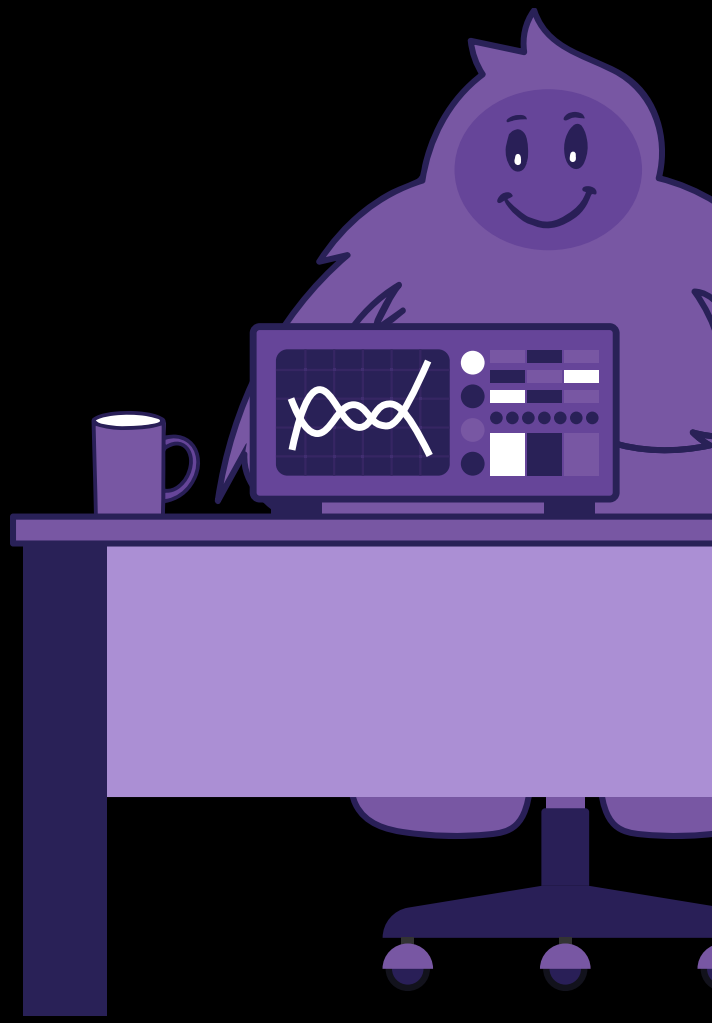
```
General Patton reporting for duty!
LEFT RIGHT LEFT RIGHT LEFT...!
FIRE!
```

Super() Powers

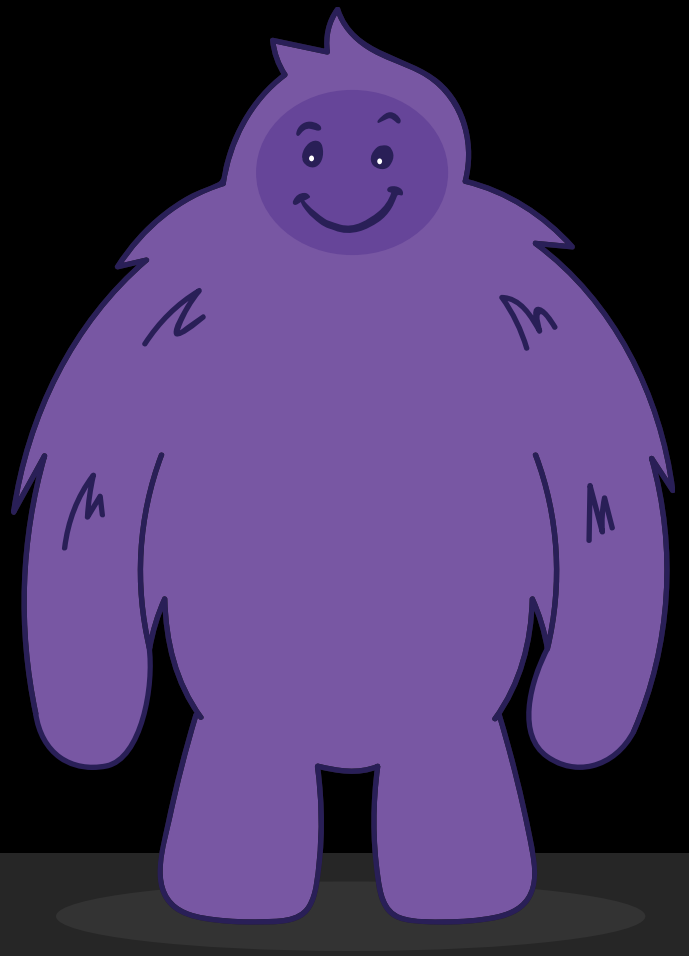
- When we're defining a child class, sometimes we'll want to call methods and access properties defined in the parent class
- We accomplish this using the `super()` method, which provides a reference to a classes *parent class*
- Let's look at an example:
- Here, we define a *base class* called *Dog* and a *child class* called *AnnoyingDog*
- This *child class* re-defines the woof method it inherits, using the `super()` method to access and call the woof method defined in the parent multiple times, achieving a more annoying effect

```
bob Woofed!
bob Woofed!
bob Woofed!
```

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

    def woof(self):
        print(self.name + " Woofed!")

class AnnoyingDog(Dog):
    def woof(self):
        super().woof()
        super().woof()
        super().woof()


my_dog = AnnoyingDog("bob")
my_dog.woof()
```



Exceptions



Exceptional Errors

- When our programs experience a problem or an error, they *throw* an Error object, also referred to as an *Exception*
- There are many different types of *Exception*, all for many types of Error!

```
>>> 5 + "5"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
>>> my_function_spelled_wrong()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'my_function_spelled_wrong' is not defined
```

```
>>> my_list = [1, 2, 3]
>>> my_list[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- Errors in Python are *objects* that are created when a problem occurs. Different classes are defined to describe different types of Error. All Errors inherit from the `Exception` base class that is defined by Python

- Errors *thrown* by problematic code can be *caught* by dedicated error handling code. If no code exists to *catch* an error when it gets *thrown*, the program crashes!

Trying and Catching

- In Python, we *catch* errors that get thrown by code using *try-except* blocks
- Unlike *if-else* blocks, each *try* must be accompanied by at least one *except* block
- Each *except* block should specify the *type* of error it will catch…

```
1  try:
2      user_number = int(input("Please enter a number! "))
3      result = user_number * 10
4      print("10 * your_number = " + str(result))
5  except ValueError:
6      print("Please put in a number!")
```

```
Please enter a number! 20
10 * your_number = 200
```

```
Please enter a number! Potato
Please put in a number!
```

- Here, we use a *try-except* block to *catch* the `ValueError` that gets thrown when a user enters a value that can't be converted to a number

- We *could* use the `Exception` base class to catch any and all errors, but this is generally a bad idea. We want to be specific when handling errors, as overly general except blocks might hide new and unexpected error types!

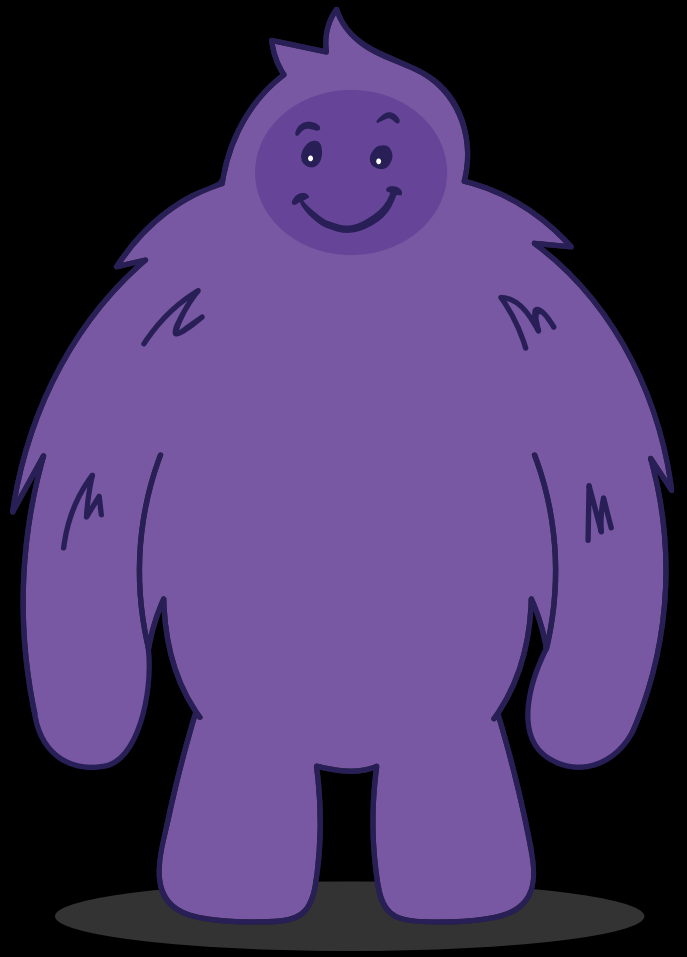
Showing Error Information

- We can use the *as* keyword to store error objects in a variable, we can then use this variable

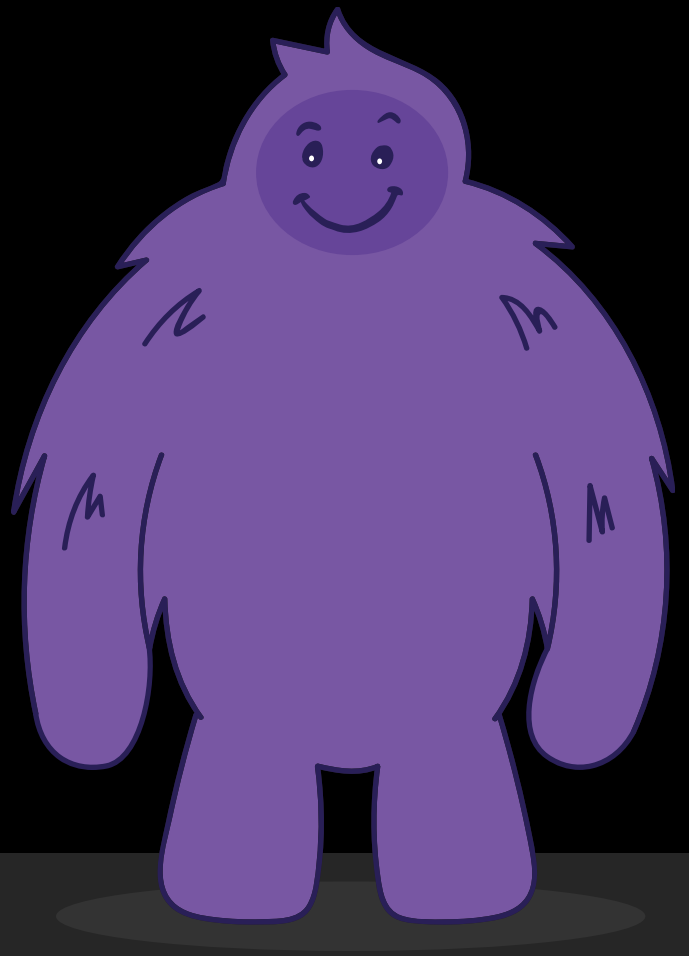
```
try:
    user_number = int(input("Please enter a number! "))
    result = user_number * 10
    print("10 * your_number = " + str(result))
except ValueError as error:
    print("Please put in a number!")
    print("\nError")
    print("---------------")
    print(error)
    print("---------------")
```

- Here, we store the `ValueError` object that gets thrown when a user inputs a dodgy value in the *error* variable
- We can then access this Error object like any other object, here, we print it to the screen to provide the user with more info…

```
Please enter a number! Potato
Please put in a number!

Error
---------------
invalid literal for int() with base 10: 'Potato'
---------------
```



Using Pip



Pip

- Python has a huge library of external modules that can be installed to provide different functionality
- Many of these modules are listed on PyPi, the Python Package index, which is maintained by the Python team
- You can search PyPi here:
  https://pypi.org/

- *pip* is a command line tool that allows us to install modules from PyPi and use them in our code
- Try invoking *pip –V* from the command-line to check that it is installed…

```
root@DESKTOP-021GAMA:~# pip -V
pip 22.0.2 from /usr/lib/python3/dist-packages/pip (python 3.10)
```

- If this command fails, try using *pip3 –V* instead. If this also fails, *pip* is likely not accessible via the command line PATH

Pip Install

- With a working install of *pip*, installing Python modules from PyPi is really easy…
- We just use the *pip install* command with the name of our desired module!

```
root@DESKTOP-021GAMA:~# pip install scapy
Collecting scapy
  Downloading scapy-2.5.0.tar.gz (1.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 13.4 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: scapy
  Building wheel for scapy (setup.py) ... done
  Created wheel for scapy: filename=scapy-2.5.0-py2.py3-none-any.whl size=1444349 sha256=6c6fe0aa11e1bb844144941aaa25506
dff68473335c28f15454462f2e613511f
  Stored in directory: /root/.cache/pip/wheels/82/b7/03/8344d8cf6695624746311bc0d389e9d05535ca83c35f90241d
Successfully built scapy
Installing collected packages: scapy
Successfully installed scapy-2.5.0
```

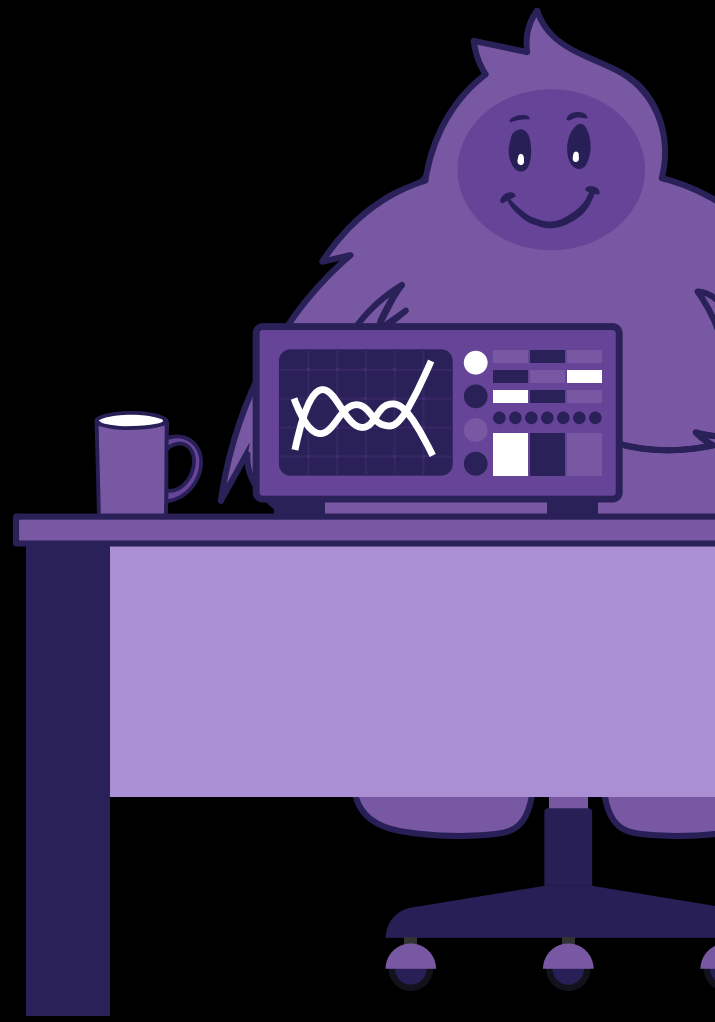
- Here, I've installed scapy, a module for manipulating network packets
- We can use the *pip uninstall* command in the same way to uninstall modules we no longer want

Venv

- One problem with Python modules is that pip will install them globally, which can cause issues if two modules conflict with one another
- A solution to this is to create a *virtual environment* for our projects
- When using a virtual environment, installing a module with pip will only install it to that environment!

- We create a virtual environment using the command:
  python –m venv [VIRTUAL_ENVIRONMENT_NAME]

- This will create a folder that houses the virtual environment in the current directory



Using Virtual Environments

- After creating a virtual environment, we need to *activate* or *source* it in order to actually use it
- Some code editors like VSCode will detect and use virtual environments automatically, however we'll manually set them up here so you can see the process…

- Linux/Mac:
- First, we navigate to [VIRTUAL_ENVIRONMENT_NAME]/bin/ , if you list the contents of this folder, you should see several *"activate"* files, e.g. *activate, activate.csh, Activate.ps1,* activate.fish…
- We then use the *source* command, supplying the *activate* file as an argument:
  source [VIRTUAL_ENVIRONMENT_NAME]/bin/activate
- To stop using the virtual environment, we use the 'deactivate' command

- Windows:
- Navigate to [VIRTUAL_ENVIRONMENT_NAME]/Scripts
- Type 'activate' whilst in this directory to begin using the virtual environment
- Type 'deactivate' to stop using the virtual environment

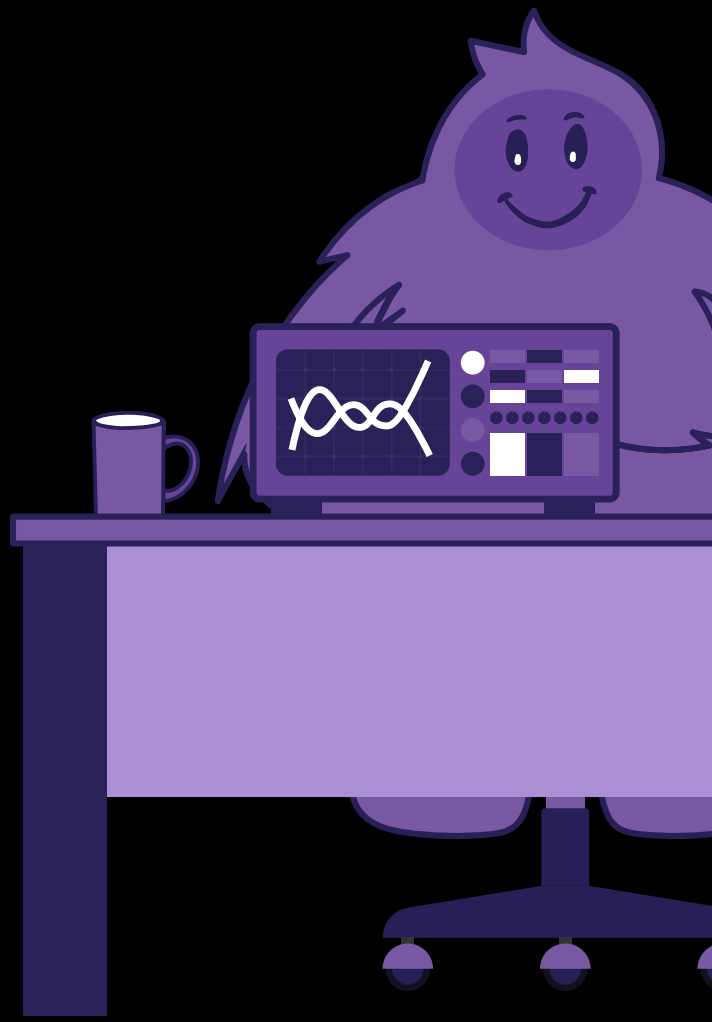
requirements.txt

- When using other people's Python tools/projects you'll likely see a file called *requirements.txt*
- This file lists all the external packages that need to be installed to run the project
- We can also list the *version number* we want to install for each package

requirements.txt

```
1   scikit-learn==1.4.1
2   numpy=1.26.4
3   colorama==0.4.6
```

- You can install all the packages in a *requirements.txt* file using pip:
  pip install –r requirements.txt



www.interruptlabs.co.uk