CODE AVENGERS

Python 1 Instructions

1. Introduction to Python

1.1. The `print` statement

This first lesson introduces the types of things you'll learn in lessons 2-10 such as printing information to the screen, and getting the user to input information to the computer. Let's get started!

The first super cool thing we are going to do in **Python** is print out our name in the **console** using the `print` statement.

A print statement looks like this:

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

This code prints the message "Hello, world!" in the console. OK so maybe this isn't super exciting, but you have to start somewhere!

When typing this code you need to type accurately; your code won't work if you leave out the `( )` or `" "`

So **be careful**!

1. Write a print statement in the code editor that says "Hello, [your name]!" e.g. "Hello, Monty!"
2. Click to test your code.
3. Click to see if you have passed the task.

1.2. Printing numbers

We can print text to the screen using the `print` statement. In programming these chunks of text are called **strings**, but we'll learn more about that later.

We can also print numbers and math using `print` statements:

```
print(5)

print(2 + 2)
```

When we print numbers, we still need the brackets `( )` but we don't need the **quotes** `" "` - they are just for **strings**.

Let's try it!

1. Click to see what the code in the editor prints out.
2. On line 2, change the `print` statement so that it uses the **addition operator** and prints out `8`.
3. On line 3, write a `print` statement that uses the **divided by operator**: `/` and prints out `4`.
4. On line 4, add a `print` statement that uses **multiplication:** `*`.

1.3. The `input` statement

To be honest, only printing stuff on a screen is kind of boring...

We can make our code lots more fun by asking the user questions and printing their answers. To do this we use an `input` statement and a thing called a **variable** to store their response.

The following code asks the user's age and **stores it** (using `=`) in a **variable** called age:

```
age = input("What is your age?")
```

We can print variables just like **strings** and numbers:

```
print(age)
```

We don't need `" "` around a variable name either. More on this later!

1. Write a line of code that asks the user's name and stores it in a variable called `name`.
2. Display the user's name using the `print` command.

1.4. Joining data in print statements

We can do more than just printing the variable name. For example, we might want to ask this: `age = input("How old are you?")`

But print out:

"You are 16" or **"16 is a great age to be!"**

There are lots of ways to do this, but let's start easy by 'adding' the parts together to get the sentences above:
```
print("You are " + age)
print(age + " is a great age to be!")
```

The `+` does not add any spaces, so make sure you **look closely** at your output to check there are spaces in all the right places!

1. Edit your code from the last task so it prints out **"Hello [name]"**
2. Add a print statement on line 3 that says **"[name] is a nice name!"**

Note: [name] means whatever name the user types in, e.g. if the user types in "Bob" it should print "Hello Bob"; if they type "Gertrude" it should print "Hello Gertrude".

1.5. Review quiz!

The final task in some lessons is a quick multi-choice review quiz so you can see how you have understood the content.

Welcome to the first one!

Review Quiz Questions:

1. Which of these is the Python command that lets the user type in some text?
2. What is the correct symbol for multiplication in Python?
3. Which of these print statements will print out the number 7?
4. The word "Sally" is saved in a variable called `name`. Which of these print statements will correctly print out: "That snake is named Sally"

2. Strings and print statements

2.1. The length of a piece of string

This lesson will give you more practice at using the `print` statement with **strings**.

Lesson 3 will focus more on numbers and calculations

So, what is a string? A string is literally just a bunch of letter, number or punctuation characters all "strung" together.

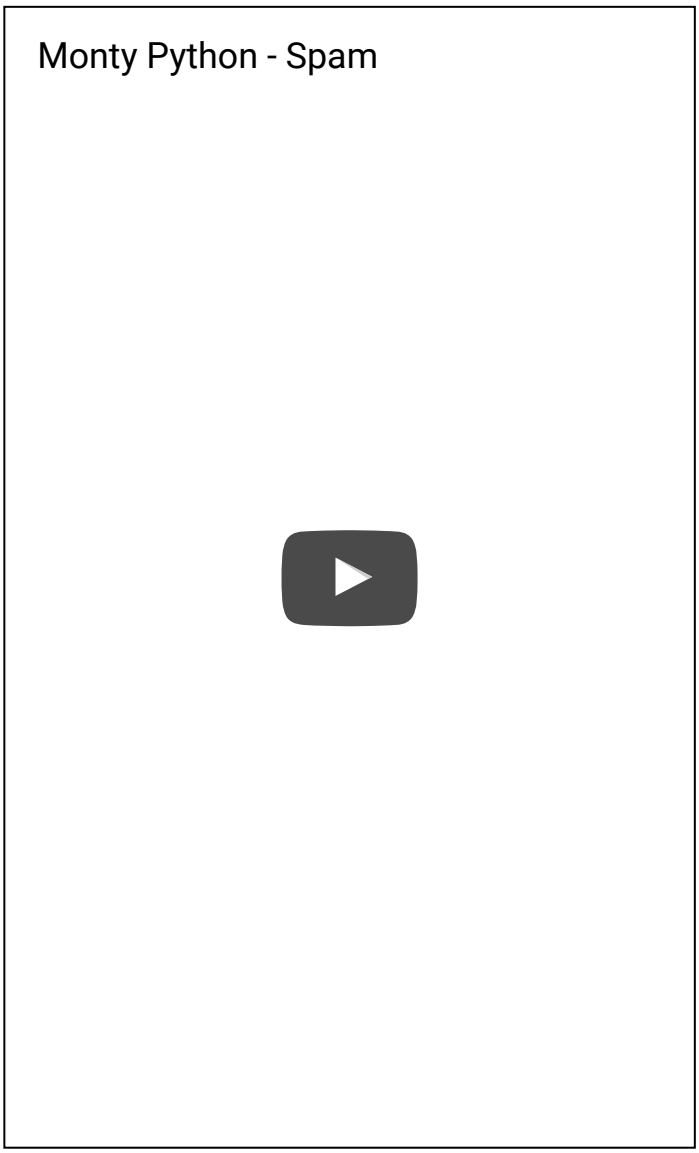
In Python, strings are written inside **quotation marks**: `" "`. You can use double or single quotes (`' '`), but you should stick to one type throughout the same program (with a couple of exceptions). Let's use `" "` for now.

We can add strings to join them: `print("Hello " + "Bob")`

And we can also multiply them! `print("Hello" * 5)`

1. Click to see what the code does.
2. Delete the existing code and print your name on line 1.
3. On line 2, combine these two strings into one print statement: "Monty Python ", "and the Holy Grail".
4. On line 3 `print` the word "Spam!" 8 times.
5. and your code!

Fun fact: The word **spam** is used to describe annoying emails we don't want because of this scene from Monty Python!



2.2. Choose the right quotes!

What happens if we want to print a sentence that has **quotes** in it? If it's `'` it will be fine:

```
print("I'm programming in Python!")
```

Strings with a `"` in them give us an error because Python thinks the second `"` is the end of the string:

```
print("Python was named after "Monty Python"")
```

We can use `' '` in this situation to fix the problem:

```
print('Python was named after "Monty Python"')
```

What if we have both `'` and `"`? We can put one type of quotes around the outside of the string, and tell the computer to **ignore any more** of these by putting a backslash `\` before them. This is called **escaping**:

```
print("And the computer said: \"I'm a computer, silly!\"")
```

Whenever possible you should use the other quote character around the outside of a string to avoid the need to `\` **escape** within the string.

Use the **guidelines above** to complete the following tasks.

1. Choose the right quotes for the quote in the `print` statement on line 1 in the code editor.
2. Choose the right quotes and print the sentence: **We are no longer the knights who say "ni", we are now the knights who say "ekki-ekki-ekki-pitang-zoom-boing!"**
3. Write one more `print` statement that says: **There are 10 types of people in this world, those who know "binary" and those who don't.**

The backslash key,`\`, which also has the pipe symbol `|` is above `Enter` on US keyboards, and next to `Z` for UK keyboards. The forward slash `/` is beneath the `?`.

2.3. Printing on more than one line

Sometimes we want to print text on more than one line. If you try to hit enter in the middle of a print statement (on line 1) you'll see that it just breaks your code.

There are 2 ways we can print multi-line statements. The first is by using triple `"` quotes: `print("""Hitting enter here will now work properly""")`

The second is to use a special **newline** character `\n`. You might recognize the backslash, which also tells the program to treat the `n` differently: `print("Here is text \n on two lines")`

The code above works fine, but is often written like this to make it easier to read:

```
print("Here is text \n"
      "on two lines")
```

1. Click to see what happens.
2. Fix the `print` statement on line 1 using triple quotes.
3. Now that it works, change it to say "does work"!
4. Use `\n` to write a multi-line print statement on line 3.
5. your code!

2.4. String variables

We can use what we have learned about strings to store data inside **variables** too:

```
name = 'Fred'
```

Right now, the value stored inside the variable called `name` is a string that says `'Fred'`. So we can say that `name` is a **string variable**.

Variables are called variables because the values they store can change during a program.

1. Have a look at the existing code to see how multi-line strings can be stored inside variables. Print out the `poem` variable on line 6.
2. On line 10, create a variable called `mood` and set it equal to `"Happy"`
3. Print out the `mood` variable
4. Use the `input` statement to ask the user how they are feeling today, store their answer in the `mood` variable
5. Print `mood` again
6. **(Optional)** Use the `+` operator to print `phrase` and `mood` together in another print statement

Programs often have variables with different **datatypes** because like we have already seen, sometimes we work with text, sometimes with numbers and other types of data.

2.5. Review quiz!

Let's see what you have learned in lesson 2.

Review Quiz Questions:

1. What characters go around text to make it a string?
2. Which of these is the 'new line' character for putting `print` statements on more than one line?
3. What character would be best to use around a **string** that says: **Don't click the wrong answer!**
4. What character would be best to use around a string that says: **A "string" is a sequence of characters**

3. Math calculations and operators

3.1. Math operators

Let's take a closer look at math in Python. Here are our main math operators:

`+` Addition

`-` Subtraction

`*` Multiplication

`/` Division

And we also have `**` for powers or **exponents**. For example, 2^3 would be written as `2 ** 3`

1. Write a print statement that calculates: **2 x 5 + 3**.
2. Write a print statement that calculates: **5^4** .
3. Write any `print` statement that has a result of 12.

3.2. BEDMAS - Order of operations

You might know from math that when you have different operations (`+, -, /, *`) in one equation, they have to be done in a certain order. One common rule is **BEDMAS**.

This means that anything inside brackets `( )` is calculated first, followed by exponents (or powers, `**`), followed by division `/` and multiplication `*`, then addition `+` and subtraction `-`: `print(3 + 4 * 2)`

In the above `print` statement the computer multiplies `4 * 2` first to get `8`, and then adds `3`. This gives a result of `11`.

If we want to do `3 + 4` first, we need to use brackets `( )`: `print((3 + 4) * 2)`

It's **very important** to check you have closed all of the brackets you have opened once you put brackets inside a print statement!

1. Add brackets to the print statement on line 1 so that the output is 5.
2. Edit the print statement on line 2 so that the power `(**)` is calculated before the subtraction (the result should be `16`).
3. Write a print statement on line 3 that adds 5 and 2 and then multiplies the result by 10.

Important note: Multiplication and division have equal priority, so if an equation just has those you can do it left to right as it doesn't matter which order you do them in e.g.

```
print(3 * 4 / 2)  # Equals 6
print(4 / 2 * 3)  # Equals 6
```

This is also true for addition and subtraction.

3.3. Identify different types of numbers

If sequences of characters are called **strings**, then what do we call numbers? There are two main types of numbers we'll encounter at this level: **integers** and **floats**.

Integers are whole numbers like 1, 56, -32 and so on.

Floating point numbers or **floats** have a decimal point in them like 3.2 or 0.1.

We identify what the **datatype** of a variable is by what is stored inside it; if it stores an integer, it's an **integer variable**, if it stores a float, it's a **float variable**.

One important thing to know is that if you divide in a calculation, you'll always get a float back, so `8 / 4` will give you `2.0`.

1. In the code editor you will see 4 variables that each store the result of a calculation. Print out each variable in a separate print statement. Look closely at the results.
2. Which variable is an **integer**? ▼

   `result_3` and `result_4` show us something interesting. In `result_3`, for example, `0.2 + 0.1` should be `0.3`, but it isn't exactly. This is because of how computers deal with **floats**. You don't need to know why at this level, it's just good to know that this happens, and it's not a bug in your code!

3. Create a variable called `result_5` and store in it a calculation that uses at least one float.
4. Print `result_5` in another print statement.

But why? It's important to know that these are treated as 2 different types of data because sometimes we need to do different things with them in a program. You will learn more about this in later lessons.

3.4. Calculations with variables

Let's try using what we've learned about numbers, `print` statements and calculations to do something useful!

A common calculation a program might do is **calculate someone's pay** (wages) based on their hourly rate and how many hours they have worked that week.

1. Create a variable under the relevant **comment** (see note below) called `hourly_rate` and store the number `15.50` in it.
2. Create a variable called `hours_worked` and store the number `40` in it.
3. Set a variable called `wages` equal to the hourly rate multiplied by the number of hours.
4. Print the result.
5. What type of variable is `hourly_rate`? ▼
6. What type of variable is `wages`? ▼
7. What type of variable is `hours_worked`? ▼

#Hashtag: The lines you see in the editor that start with `#` are called **comments**. These are 'notes' that explain what the code does. They are just for the humans that are reading the code so the computer ignores them. More on that in **lesson 5**!

3.5. Python math review quiz

Let's review what we've learned about math in Python!

Review Quiz Questions:

1. Which of these is the correct symbol for doing powers such as 2^3?
2. Which of the following print statements prints 7?
3. Which of these variables is a float?
4. In which of these calculations does the addition `(+)` happen first?

4. Combining things in print statements

4.1. Joining things in a print statement

As you have seen we can add two strings together with the `+` operator:
`print("Hello, " + "world")`

We can also add variables that have strings stored in them: `print("Hello " + name)`

```
print(greeting + name)
```

What happens with numbers?

1. Add the `wages` variable to the print statement on line 6.
2. Click to see what happens. You will see an error–**you don't need to fix this.**

   We get an error with `+` because in Python strings and numbers can't be added together in this way. We'll learn how to deal with this in the next task.

3. Click to move on to the next task.

4.2. Joining strings with numbers

One simple way that we can combine strings with numbers or calculations is to use a comma - `,` like this:

```
print("This week you have earned: $", wages)
```

This will print a space between the two parts automatically.

1. Change the `+` in the print statement to a `,` (comma).
2. Click to see the output.

4.3. Joining strings with calculations

This method can also be used to join calculations with strings:

```
print("5 + 4 =", 5 + 4)
```

Which will print out:

```
5 + 4 = 9
```

Remember we can use variables that store numbers **anywhere** that we would use numbers:

```
print("The area of the rectangle is: ", length * width)
```

1. Add a calculation on to the end of the `print` statement on line 2.
2. Do the same for line 3.
3. Add a **string** to the start of the `print` statement on line 4 so that it makes more sense.
4. Use the variables from lines 7-8 to add a calculation into the `print` statement on line 11 for the circumference of a circle.

Notes:

- The **comment** starting with `#` on line 10 has a hint for the circumference calculation.
- `PI` is a **constant** value (it doesn't change) so it is written in `UPPERCASE`. Make sure you write it the same way in your code. More on this later!

4.4. Format print statements using the built-in `.format()` function

A comma (`,`) is not always the best way to combine strings and numbers in a print statement, because there is always a space between them and sometimes (like in the wages example) we don't want a space.

Python has a built-in function called `.format()` which gives us heaps more control over how our stuff is printed on the screen.

Here is how `.format()` is used, and you can **click here for an explanation**:

```
print("9 * 3 = {}".format(9 * 3))
```

1. Use `{}` and `.format()` to include `wages` in the print statement on line 8.
2. `.format()` the print statement on line 11.
3. `.format()` the print statement on line 12.
4. `.format()` the print statement on line 13.

4.5. Print more interesting statements with `.format()`

The best part about `.format()` is that it works with **anything**, and you can put the missing information **anywhere** in the string:

```
print("Hello {}, nice to meet you!".format("Bob")) print("The circumference is: {}".format(PI * diameter))
```

We can also put in more than one bit of data by separating each part with a `,`:

```
print("'Hi {}, you earned ${} this week!".format(name, wages))
print("The area of a {} by {} rectangle is: {}".format(length, width, length * width))
```

1. Use `input()` to ask the user for their name and store it in a variable called `name`.
2. Ask the user for their age and store it in a variable called `age`.
3. Print out both variables in a string using `.format()`.
4. Click to test your code.

It's **really important** to make sure you have closed the `print` brackets **AND** the `.format()` ones!

5. Comments and variable names

5.1. Write comments for the humans that read your code!

Back in lesson 3 we took a quick look at **comments**. In this lesson we are going to take a closer look at this and some other **conventions** for writing Python.

Conventions in coding are basically a set of **guidelines** that everyone agrees on to make it **easy to understand** code, especially if it was written by somebody else!

Comments are designed to help **humans** understand what is going on and why:

```
#Store the user's chosen name for use throughout the game
user_name = input("And by what name shall this brave adventurer be known?")
```

Usually we try to keep comments short, simple and necessary. Here's an example of an **unnecessary** comment: `#Set hourly_rate equal to 15`
`hourly_rate = 15`

1. Take a look at the code in the code editor. Add a `#` before each line you think should be a comment then click to check.
2. There is one comment in there that doesn't need to be there, delete the one that isn't needed.

5.2. Choose good variable names!

A **variable** can be named almost anything. Python variable names can have letters, numbers and underscores _ in them.

There are some guidelines to choosing **good** variable names:

- Avoid **single letter** variable names (**most of the time**) e.g.: `a = 15`
`b = 20`
`c = a * b` It is hard to understand what this code does, compared to:
`length = 15`
`width = 20`
`area = length * width`

- Use **decent, descriptive** names e.g.

`hourly_rate = 15` can be better than

`rate = 15`.

Use words from the comments and code in the editor to help you choose better variable names for these variables:

1. The **n** variable should be replaced by ▼
2. The **s** variable should be replaced by ▼
3. The **a** variable should be replaced by ▼

When you follow these guidelines, your code is **easy to understand** which also means you need **less comments**.

5.3. Choose even better variable names

Here are some more guidelines for well-named variables:

- Variable names can't start with a **number**.
- Variable names should be written as `lower_with_under` which means all **lower case**, with **underscores** _ separating words. This is compared to `camelCase` which is used in other languages:

```
#Python
first_name = "Bob"

#JavaScript
firstName = "Bob"
```

1. Change the variable name on line 2 from `camelCase` to `lower_with_under`.
2. Do the same for the variable on line 3.
3. Do the same for the variable on line 4.
4. Now fix up the variables in the print statement on line 7.

When developers follow the guidelines for a language it makes it much easier to **work in teams** where many different people work on the **same code**.

5.4. Write all that punctuation so it's easy to read

Python has guidelines for all that **punctuation** too! Here's a few tips to make your code more readable:

- **Don't** put spaces before or after brackets: `print ( "Hello, world!" )` is hard to read compared to: `print("Hello, world!")`
- **Do** put spaces between numbers and operators in calculations, on either side of equals`=`, and after commas`,`: `print("2 + 2 - 1 =", 2 + 2 - 1)` is easier to read than `print("2+2-1=",2+2-1)`
- **Don't** use brackets when you don't need them: `print((3 * 21) + 4)` - the `*` already happens first.

1. Fix up the calculation in the variable on line 1 in the editor.
2. Fix up the `print` statement on line 2.

My hovercraft is full of eels

5.5. Python conventions quiz!

Let's see how much you have absorbed about how to write good Python code!

Review Quiz Questions:

1. Which symbol is the symbol for writing comments in Python?
2. If you wanted to store the number of students present in your class today, which would be the best 'Pythonic' variable name for it?
3. Which of these `print` statements is written most correctly?
4. Which of these is a valid Python variable name?

6. Turtle drawing basics

6.1. Meet Tia the Turtle

We're going to take a break from `print`, `input` and math for a little while to play with some **Turtles**!

Turtle graphics are a bit of a programming tradition and they really help to understand the idea of **sequence** in programming.

The example code shows you how to create a turtle called `tom` by first **importing** the `turtle` module.

1. In the code editor `import` the `turtle` module.
2. On line 2 create a turtle called `tia`.
3. Type `tia.forward(50)` on line 3.
4. Click to see what happens.

6.2. Move and turn Tia

We can **move** a turtle forwards or backwards using: `tom.forward(50)`
`tom.backward(100)`

The number in the brackets is how many **pixels** (px), or how far, the turtle will move.

We can also **turn** a turtle using: `tom.left(90)`
`tom.right(45)`

Here, the number is the **angle** it will turn, in degrees. So `90` will make it turn a right angle, `180` will turn it to face the opposite direction and `360` will turn it all the way around.

1. Click to see what the code in the editor does.
2. Complete the code to make `tia` draw a **square**.
3. Modify the code so that `tia` draws a square that is **200px x 200px**.

6.3. Change the color and size

Great! So we're drawing with our turtle. We can **change the color and size** of the line that gets drawn using: `tom.color("red")`
`tom.pensize(5)`

1. On line 3 set the `pensize()` to `10`.
2. Edit your code for the square so that each side is a different color, in the order: **red, green, yellow, blue**.
3. Click to see the image!

These commands happen in **sequence** or we can say they are **sequential**.

It's very important in programming to be able to break down a problem into **steps that happen in order**. And then we have to **make sure** that those things **do** happen in that exact order!

If you change the color or turn in the wrong place, you will end up with a different result.

6.4. Draw some different shapes

You can draw all kinds of things with turtles, but you have to think carefully about how to do it.

In this task there are comments to show you where to write the code for each shape, and these: `.goto(x, y)`, `.penup()` **and** `.pendown()` have been used so that each shape can be drawn on a different part of the screen, without messy extra lines connecting them.

Check the reference section for more detailed explanations of these functions.

NOTE: Use `.left()` for all of the shapes in this exercise to keep them on the canvas!

1. Under the correct comment, from line 11, draw a **purple rectangle** that is `120` wide and `50` high.
2. Under the correct comment write the code to draw an **orange triangle** with sides that are `60`px.

6.5. Filling shapes with colors

We can also **fill shapes** with a color. There are 3 commands we need for this: `tina.fillcolor()` sets the color for filling. `tina.begin_fill()` goes just before the line of code that starts drawing your shape. `tina.end_fill()` goes right after the last line of code used to draw the shape.

1. Click to see what the code in the editor does.
2. Set the pen size on line 5 to `8`.
3. Set the `fillcolor()` to yellow on line 8.
4. Add `begin_fill()` on line 9 - we don't need anything in these brackets.
5. Add `end_fill()` on line 18.
6. Click again if you closed your canvas.

7. Introduction to variables

7.1. Calculate donut orders... Mmmm... Donuts....

Variables are a really important part of programming, because they make it easy to **change your code** later on.

Imagine if you had a big pay program with the workers' hourly rate of $15/hr written in there 50 times **as a number**:
```
worker1_pay = 15 * 40
worker2_pay = 15 * 35
worker3_pay = 15 * 27...
```

If their pay was increased to $17/hr you would have to change *all 50 of them individually!*
```
worker1_pay = 17 * 40
worker2_pay = 17 * 35
worker3_pay = 17 * 27...
```

If we use a variable, we only have to change the number **once**, then anywhere its name is written, the new value will be "plugged in":
```
hourly_rate = 17
worker1_pay = hourly_rate * 40
worker2_pay = hourly_rate * 35
worker3_pay = hourly_rate * 27...
```

1. Read the comments in the code, then fill in each `??` in the calculation on line 4 with the correct number of each donut to work out what **order #1** costs.
2. Write the correct calculation for **order #2** on line 7.
3. Use `{}` and `.format()` to output the cost of order 1 on line 10.
4. Add a formatted `print` statement for **order #2** after order #1.

7.2. Use variables to make code more flexible

It works fine if we just write the numbers in there to do the calculations, but it's not very **flexible**. Programmers call this **hard coding**.

If the price of the donuts **changes**, we have to change lots of numbers for our code to work.

If we make a **mistake** while doing this, it's a lot harder to see where the mistake is... *Do we have the wrong number of glazed or filled donuts?!*

Let's **refactor** this code by creating some variables to store the price of each type of donut!

1. Which names would be best for these variables:
   ▼
2. On lines 2-4, create variables to store the price of each type of donut using the names you have selected.
3. In each calculation, change the prices to the appropriate variables.
4. **(Optional)** Try changing the order summary output to a single multiline print statement with both order costs.

7.3. Write a program for game purchases using variables

Now it's your turn! You've had some good practice at **creating variables**, **doing calculations** with them, and **outputting the result** to the screen, so let's try another example.

C.A. Games Sale!

The local gaming store **C.A. Games** is having a crazy sale on Playstation games! The cost of games is as follows:

- Playstation 3 games: $20
- Playstation 4 games: $45

Let's get coding! Use the comments to help you.

1. **Create variables** for each price you need to store. Use `ps3_game` and `ps4_game` as the names for the variables.
2. Write a calculation for each of orders 1-2. Store the costs in variables called `order1_price` and `order2_price`.
3. Print out each **order total** along with which order it is e.g. "Order 1 costs: $10". Use a separate print statement for each order.
4. **(Optional)** As an added bonus, you get an **extra $10 off your total order** if you buy **3 or more** of the same type of game (e.g. PS3). This discount can only be applied once to an order. Include the discount in the calculation for any order that is eligible.

7.4. Ask for user input to make the program interactive

Now our program is more **flexible** - we can update the price of any game type by changing just **one number**.

It would be nicer if we could **type in** the number of each game instead of leaving it hard coded though, e.g.: `num_ps2_games = input("How many PS2 games?: ")`

There's one **important thing** here - if you use `input` you always get a **string**. We're going to need the `int()` function we saw earlier: `num_ps2_games = int(input("How many PS2 games?: "))`

And then our calculations can use both variables: `ps2_total = num_ps2_games * ps2_game`

1. Use `int()` and `input()` to **ask the user how many** of each type of game have been purchased, store those values in variables called `num_ps3_games` and `num_ps4_games`.
2. For each type of game, calculate the **total cost** and store in a variable. Use `ps3_total` and `ps4_total` as the variable names.
3. Calculate the **total cost** for the order and store in a variable called `total_cost`.
4. Print out the total cost for the order in the sentence: "Your order costs: ${}"
5. **(Optional)** Try print out a summary of each game type after the total cost - you can choose which data to include.

Watch your brackets!

7.5.

Review Quiz Questions:

1. Which of the following input statements will correctly take in a number that can be used in a calculation?
2. Why are well-named variables important in programming?
3. If `book_price = 5.50` and the deal is "buy 2 get the 3rd half price", which calculation will give the correct `total_price`?
4. In the following code, what would be the best names for the variables labelled `a`, `b` and the number **15** (in that order)?

8. Numbers and variables

8.1. Increment (increase) an integer variable using +=

We've actually learned a lot about working with **string** variables, so now let's take a closer look at working with **integer** and **float** variables.

If we have a number stored in a variable there are lots of ways we might want to change it, but a common way is that we might want to **increment**, or add to it: `my_var = 10`
`my_var = my_var + 10 #Now my_var = 20`

So we're **adding 10** to the number and storing it back in `my_var`. In Python there is a nice shorthand for this: `my_var += 10` where the `+=` operator does **exactly the same thing**.

1. Create a variable called `num_pies` and set it equal to `0`.
2. Print `num_pies` in the sentence: "Lisa has {} pies".
3. Use `+=` to give Lisa 5 pies.
4. Print the `num_pies` sentence again.

Easy as pie right?

8.2. Increment a variable, using another variable

Let's get the user in on the action this time.

Remember we have to use `int()` if we want to use their input as an **integer** rather than a **string**. There is also a function called `float()` which you can use if you think their input might have a decimal point in it (like if it's money).

```
start_value = 10
amount_to_add = int(input("How much should I add to
  the number 10? "))
start_value += amount_to add
```

1. Ask the user how old they are and store it in an appropriately named variable.
2. Ask them how many years they would like to add.
3. **Increment** the age variable using their input.
4. Print out the result in the format "In {} years you will be {}".
5. Add in a few **concise comments** where you think they are most needed.

Important note about int() and float().

8.3. Use other operators to change a variable

Awesomely enough, Python doesn't just come with `+=` to easily change a variable, we also get `-=`, `*=` and `/=`:

`my_var = 15` **stores the value 15** in a variable called `my_var`

`my_var -= 5` **subtracts 5** and stores the result back in `my_var`

`my_var /= 2` **divides by 2** and stores it back in `my_var`

`my_var *= 4` **multiplies by 4** and stores back in `my_var`

1. What is the value of `my_var` in the example instruction code above after all 3 changes?
2. In the code editor, **subtract** 10 from `number1` and print the result.
3. **Multiply** `number2` by 4 and print the result.
4. **Divide** `number3` by 8 and print the result.

8.4. Calculate discounts using shorthand operators

Let's try this with some context by going back to our **C.A. Games Sale**.

These operators are useful if we want to work out something like a **discount**, say **$5 off**: `price = 15`
`price -= 5 #Takes off $5`

Or **half price**: `price = 15`
`price /= 2 #Divide by 2 and store it back in price`

Or even a **10% discount** *(see the note below the tasks for how this is done if you're unsure about percentages)*: `price = 15`
`price *= 0.9 #10% discount`

1. **PS3 games** have an **extra $2 off** for **one day only**, so change the price using `-=` on line 6.
2. All customers also get **15% off** their total order, so change the `total_cost` on line 20 before it is printed out.

Calculating a % Discount: One easy way to work out a price after a % discount is:

1) Write the percentage **as a decimal** (either in your head or by dividing by 100) e.g. `10% = 10/100 = 0.1`

2) **Subtract that from 1** e.g. a 10 percent discount is `1 - 0.1 = 0.9`

3) **Multiply the original price** by that number e.g. a 10% discount on $30 is `30 * 0.9`

So in our code we can write `price *= 0.9` for a **10% discount**.

8.5.

Review Quiz Questions:

1. Which operator **increments** a variable?
2. Which decimal number would calculate the price after a 15% discount if you multiplied the original price by it?
3. If `var_2 = 4`, which of these results in `var_2` being 1?
4. What is the value of `var_1` at the end of the following code?

9. Debugging print statements

9.1. Hunting for bugs

Another important skill to have in programming is the ability to find mistakes in your code, or **debugging**.

You've probably discovered this already if you've had code that has given you error messages!

So let's go bug hunting...

1. **Quotes** can be tricky to get right, fix the bug on line 1.
2. See if you can remember how to fix the bug on line 3.
3. There's another error with the quotes on line 5.
4. Good luck finding the problem with line 7!
5. Click to check that all of the code works now.

9.2. Debug problems with brackets

Brackets were easy when it was just: `print("Hello, world!")`

But now we have brackets inside brackets inside brackets, so you'll need to get good at spotting bracket bugs!

1. Find the bug in the `print` statement on line 1.
2. There's a really common bug on line 3.
3. There are several bugs on line 5 to fix up!
4. Click to check that all of the code works now.

9.3. Debug variables

Variables are **case sensitive**, which means that `my_name`, `My_Name` and `MY_NAME` are all different variables.

That's why we try to make them all `lower_with_under` - it makes the chances of a mistake lower.

We also have to spell them exactly the same, with underscores in the same place, any time we use them.

1. Something is wrong with the variable name on line 4, fix it up!
2. Check all the other variable names for bugs and fix them too.
3. Click to check that the code works correctly.

And here's another **Monty Python** treat - the **Knights who say Ni!**:

Ekke Ekke Ekke Ekke P'tang Zoo Boing!

9.4. Debug code that uses numbers

Not even computers like it when they can't calculate a math problem!

When writing python code that works with numbers, we have to make sure we have our **order of operations** correct.

We also have to make sure that if a number is coming from an `input` that we convert it using `int()` or `float()`

1. Click run to see what happens
2. Something is going wrong with the calculation. This **TypeError** means we're trying to do maths with the wrong kind of data. See if you can fix this!
3. Click again. Hmm, that price looks a bit strange!
4. The calculation does not calculate what it is supposed to. Fix the bug in the calculation on line 8.

9.5. Put it all together for some real-world debugging

Now to put your skills to the test. The code in the editor has bugs of all the types you just learned about, and it is your job to get the code working!

1. Click to see what happens.
2. Find ALL the bugs!
3. Click again to check it's all working.

10. Review lessons 1-9

10.1. Review `print` and `input` statements

In **Lessons 1-9** we have covered all the basics of **sequential** programming in Python. We can now write code that completes a set task **step by step** and know the importance of getting these steps **accurate** and in the **right order!**

This lesson will review what you've learned one more time before moving on to more complex stuff.

A `print` statement **outputs** text to the screen. It can contain **strings**, numbers, calculations or **variables**.

An `input` statement lets the user **type in** text so that we can **store and use it**.

1. Ask the user for their **name** and store it in a variable.
2. **Print** just their name to the screen.
3. Ask the user what **year** they were born in and store it in a variable.
4. **Subtract** this number from 2017 and store it in another variable.
5. Print the result in the sentence "You will be {} years old this year!"

10.2. Review all that maths!

You've also learned a lot about how **math** is done in Python, including the **order of operations** or **BEDMAS/PEMDAS**.

Calculations can be **printed** in a `print` statement:

```
print(3 * 4 - 7 ** (4 / 2))
```

Or **stored** in a variable:

```
result = 10 - 4 * 2
```

And we can use variables that contain numbers to do **calculations**:
```
num1 = 7
num2 = 5
print(num1 * num2)
```

1. **Print** the calculation described in the comment on line 1.
2. On line 6, use a shorthand **operator** to **add 10** to the score variable, then **print it** on line 7.
3. Use the variables on lines 10-11 to calculate the **area of a rectangle** inside the print statement on line 13.
4. Ask the user for **2 numbers** and print out **the sum** (+) of them. Use the comments to lay out your code.

10.3. Review working with strings

We looked at how to work with **strings**, including **joining them** together and getting the **speech marks/quotes** right.

`print("Your name is " + name)` is used for just **2 parts** when they are **strings or variables containing strings**.

`print("3 x 4 =", 3 * 4)` works if you need to join a **number or calculation** to the end of a **string**, but it always **adds a space**.

```
print("Here's a number: {}. Did you know 3 x 3 = {}? And
here's a word: {}".format(7, 3 * 3, "spaghetti"))
```

We can use `.format()` for **everything**, but it's particularly useful when we need to put something in the **middle** of a string, or **more than one** thing in there.

1. **Combine** the two string variables in the code in one `print` statement on line 5 using `+`.
2. **Print** out the string "Coding is fun!" 8 times using 1 `print` statement on line 8.
3. **Include the calculation** in the print statement on line 11 using `,`.
4. Put the `phone` variable into the print statement using `.format()`.

10.4. Write good code that works!

Finally, we also learned about **writing good Python code** by following the **conventions**, and how to look for **bugs** to make sure code works. This included:

- **Good** variable names - `lower_with_under`.
- **No spaces** around brackets e.g. `print ( "hello" )`.
- Getting the **quotes** right!
- Checking **brackets, quotes, spelling** of variable names and that we have **used `int()`** if we need a number from an input.

1. Correctly print out: **"Python" is a cool programming language** on line 1.
2. Tidy up the code on line 3.
3. Rename the variables on line 6-7 with more **Pythonic** names.
4. Now fix the variable names on line 9.
5. Add the missing variables into the `.format()` on line 11.
6. Fix the bugs on line 11.
7. Click to check it all works!

10.5. Review quiz time!

Time for the last review quiz in this section.

Review Quiz Questions:

1. Which is the better print statement?
2. Which calculation prints 12?
3. What is the value of x at the end of this code?
4. Which input correctly gets a user's favorite number for a program that will add 10 to it?

11. Introduction to selection

11.1. Introduction to selection

So far we have seen code that runs in **sequence**. However, often in a computer program **decisions** need to be made.

This is called **selection** in programming - making **decisions** about what should happen based on **conditions**.

In this course, the keywords `if` and `else` form our key **selection/conditional** structures.

This lesson will introduce you to what will be covered in **lessons 11-20**. Don't worry if anything seems confusing at this stage, as you'll learn about it all more in the next lessons. So, let's take a look at some code!

1. Click and type in your name.
2. Click and type in "Monty" with a capital "M".
3. Change line 3 to have your name after the `==` instead of "Monty".
4. Click again and type in your name.

Important Note: `if` statements are **case sensitive**, so that means typing "bob" is different from typing "Bob" - you'll learn how to deal with this in **lesson 12**.

11.2. Understand how a condition works

When an `if`/`else` statement is written, it always has a **condition**. That's the part that gets **checked**, and then a **decision** on what to do is made based on whether the condition **evaluates** to `True` or `False`.

If the condition is `True` the `if` branch gets run. The `else` branch gives a **default case** that gets run if the condition is `False`.

The simplest check is checking whether or not two things are **the same** with the **is equal to** `==` **operator**.

If we print out a **comparison** of some kind, Python will tell us if it's True or False:

`print(4 == 9)` prints out `False`, but:

`print("Bob" == "Bob")` prints out `True`.

1. Click to see these in action.
2. Write a print statement on line 5 that will print `True`.
3. Write a print statement on line 8 that will print `False`.
4. Click to check you're correct.

11.3. Use different comparative operators

There are a few different types of comparisons we can make inside **conditions**. The symbols we use, like `==`, are called **comparative operators**. You'll learn more about these in **lesson 13**, but here are a couple more: `>` is greater than `<` is less than

So a condition like `if 4 < 9:` would **evaluate** to `True` because 4 **is** less than 9.

Let's try checking some conditions!

1. Change the **operator** in the print statement on line 2 so that it prints `True`.
2. Change the print statement on line 4 so that it prints `False`.
3. Change the `if` statement so that the messages work properly.
4. Make sure you your code a couple of times, entering numbers **above and below 10** to check it works!

11.4. Use `elif` to make more options

So using **conditional `if`/`else`** structures is pretty much like putting a **fork in the road** in your program. There are 2 **branches** and it could go either way depending on whether the **condition** is **True or False**.

Sometimes we might want **more than two** branches on our path. We can do this with `elif` (short for else if), which gives us a chance to add another condition. Run the example code to see this in action.

In the code editor you will see another **comparative operator**: `<=` **less than or equal to**. There is also a `>=` **greater than or equal to.**

1. Click to see what the code does. Do this **3 times** with a number from **1-10**, one from **11-80**, and one **above 80**.
2. Change the `<=` on line 3 to `<`, then see what happens if you run it and type in 10.
3. Put the `<=` back, then change the condition on line 5 so that numbers **up to 100** are medium numbers *(instead of 80)*.
4. Test the program again with the **3 types of numbers** to make sure it works.

11.5. Review quiz on selection

Let's quickly review the basics so we can get into some more complex stuff!

Review Quiz Questions:

1. Which of these print statements prints **True**?
2. Which operator means "less than"?
3. Which operator means "greater than or equal to"?
4. Which `if` condition evaluates to **False** if the number `16` is stored inside the variable age?

12. More selection options

12.1. Get to know the `if` statement a bit better

The `if` keyword will make a piece of code run if the condition in it is `True`. It's a bit like if someone's parent says: "I'll buy you a new phone `if` you keep your room tidy for a month". As long as they meet that **condition** (tidy your room), they'll get the new phone.

Look at the example code. An `if` statement must look just like this with the word `if`, a **condition** and the colon `:`.

The code that we want to run is **indented** by pressing **tab** to show that it is **inside** the `if`. This is called a **block**.

1. In the code editor there is a `:` missing, put it where it should go.
2. **Indent** the block that should be inside the `if`.
3. Click and type in "happy" to see if it works.

Remember it's **case sensitive** so "Happy" is different from "happy".

12.2. Add a second if statement for more options

Maybe we want to have **2 different responses**. This time you will have a go at writing a **second** `if` statement.

1. On line 5 add another `if` statement that checks if the user types in "sad".
2. On line 6 add a block that prints "Sorry to hear that" if they do type in "sad".
3. Click and test your code by typing in "sad".
4. Click again and test it with "happy" to make sure you haven't broken the first `if` statement.

Handy Hint: Whenever you have `if` statements in your code, it is always important to test all the different **branches** or outcomes you are expecting, just like you did in this task.

12.3. Add a default response using `else`

If we wanted to check for **every possible mood** we could think of, we would need **a lot** of if statements! Luckily, Python has another handy keyword called `else`.

This basically says if it's **anything** other than the thing we checked for in the `if`, then do this e.g.
```
if fave_food == "pizza":
  print("Yum!")
else:
  print("Yuck")
```

In this code, if you type `"pizza"` it will say `"Yum!"` and if you type **anything else at all** it will say `"Yuck"`. This is why `else` doesn't need a **condition**.

You can test this out in the example code.

1. Change the second `if` (on line 7) to say `else` and delete the condition.
2. Click and try typing in anything other than "happy".
3. Click and check that "happy" still works as expected.

12.4. Adding more than one condition using `elif`

Hmm, our code works but it might be useful if we could have a couple more responses to **specific moods**, and then let our `else` run for anything else.

When we want more than one **branch** with a **condition** in our code, we use `elif` which stands for **else if**. The `elif` connects to the `if` before and the `else` after it, making sure that only one of the branches can run.

Click here for an explanation of this code:
```
if fave_food == "pizza":
  print("Me too!")
elif fave_food == "nachos":
  print("Those are pretty good")
else:
  print("Hmm, not one of my favorites")
```

1. Change the 2nd `if` to an `elif` to join the statements together.
2. On lines 9-10 add another `elif` that prints "You should get an early night" if the user enters "tired".
3. Below this, add an `else` that will print "Oh, really?" for all other cases.
4. Click 4 times and test **each branch** of your `if` statement to make sure they all print the **correct message.**

12.5. Review quiz

What have you learned about `if` statements?

Review Quiz Questions:

1. Which character goes at the end of a line of code that starts with `if`?
2. What is the keyword used for making a second branch in an if statement when both branches have a condition?
3. Which operator means **is equal to** in Python?
4. How do you know which bit of code is inside an if statement?

13. Boolean values and operators

13.1. Recognise and use > and < (greater than and less than)

In this lesson we will look more closely at **conditions**. **Boolean** (pronounced BOO-lee-in) is the name given to these special types of values that are either **True** or **False**.

The code `x = False` creates a variable called x and stores the value **False** in it. We then say that the **datatype** of x is Boolean, or x is a Boolean variable.

In Python: `<` means **less than**; `>` means **greater than**. `<=` means **less than or equal to**; `>=` means **greater than or equal** to. `==` means **equal to**; `!=` means **not equal to**.

The code `a = 25 > 12` sets a to **True** since `25 > 12` is **True** (25 **IS** greater than 12).

Any calculations are done **before** things are compared.

13.2. Using `and` and `or` keywords

In Python there are some other useful keywords for working with **Boolean** values, `and` and `or`:

a `and` b is True if **both** a **and** b are True.

a `or` b is True if **either** a **or** b or **both** are True.

Some examples:

`a = 8 < 9 and 2 >= 2`

This sets a to **True** since `8 < 9` is true **AND** `2 >= 2` is **True**.

`b = 5 > 2 or 3 == 2`

This sets b to **true** since `5 > 2` is True, even though `3 == 2` is not. If they were **both False**, b would be **False**.

Note that `>` and `<` are evaluated **before** `and` and `or`.

13.3. Use `not` - not ! but the other NOT...

We also have the word `not`. This sort of "reverses" whether something is `True` or `False`. So `not True` is **False**, and `not False` is **True**.

Here's another example:

`not 3 > 4` is `True` because `3 > 4` is `False` and the `not` reverses it.

We can store **Boolean** values inside variables too, just using the words `True` and `False` (which must start with a capital in Python):
```
x = True
y = False
```

Now `not x` is **False**, and `not y` is **True**.

Note that `not` is evaluated **before** `and` and `or`.

13.4. Using operators with strings

You might be wondering what happens if you put **text** rather than **numbers** into these comparisons.

If we use our `<,<=,>=,>` operators, then Python works out which string comes first **alphabetically** based on the first letter, like the letters are numbered 1-26:

`"a" < "b"` is **True** because **a is before b** in the alphabet *(1 is less than 2).*

`"Turtle" < "Aardvark"` is **False** because **T** comes after **A** in the alphabet *(20 is greater than 1).*

We also have a fun operator called **`in`**. When used with strings, this will tell you if a string is in another string:

`"a" in "apple"` is **True**, but `"B" in "banana"` is **False** - **B** is not the same as **b**.

13.5. Putting it all together

Let's review **Boolean** values by trying some examples with **all of the operators**.

Boolean is the word used to describe values that are either **True** or **False**. Just like **strings**, **integers** and **floats**, if a **variable** contains a Boolean value, we call it a **Boolean variable**.

Remember:
```
< less than
> greater than
<= less than or equal to
>= greater than or equal to
== equal to
!= not equal to
and - both must be True
or - one must be True
not - makes it the opposite
```

We have looked at how these operators store Boolean values in variables, but they work the same way in **`if`** statements as you'll see in upcoming lessons.

14. Matching strings

14.1. Take a closer look at the == operator

Let's take a closer look at `==` and how it works with **strings**.

With **numbers**, it's pretty easy:

`4 == 4` is clearly **True**, while `3 == 27` is clearly **False**.

With strings it's a bit more complicated. As a human, you might look at the comparisons below and think they would be **True**: `"Hello" == "Hello "` `"Hello" == "hello"`

Computers, however, think of capital letters and lower case as different (**case sensitive**), and they don't ignore extra spaces like we might.

1. Click and type in "bob" with no capitals to see what happens.
2. Which message is displayed? ▼
3. Click and type in "Bob" to see what happens.
4. Which message is displayed? ▼
5. Click to carry on.

In this lesson we will look at how to **work with strings and** `==` so that our **`if`** statements work properly.

14.2. Match strings using the == operator

What does **case sensitive** mean? It's important to understand what that means for comparing two strings because it can cause unexpected results if you get it wrong.

The bottom line is this:

`print``("Hello" == "Hello")` will print **`True`**, but:

`print``("Hello" == "hello")` will print **`False`**.

Even if all of the letters are the same, if one is a **different case** the strings are **not** the same.

1. Make the print statement on line 2 print **`True`**.
2. Replace the `??` to make the print statement on line 3 print **`True`** *(Use the same sentence).*
3. Make the print statement on line 6 print **`False`**
4. Replace the `??` to make the print statement on line 7 print **`False`** *(Use the same sentence).*

14.3. Match strings with spaces

Believe it or not, **spaces** also make a difference when matching strings! So these 4 strings are **all different**: `"Hello, how are you?"`
`" Hello, how are you?"`
`"Hello,  how are you?"`
`"Hello, how are you? "`

The next task will show you how to deal with extra spaces at the beginning and end, but for now let's check that you've got the idea.

1. Add some **spaces** to the start or end of the strings on line 2 so that the print statement prints **`False`**.
2. There's a string stored inside the `phrase` variable but it doesn't quite match the **condition** on line 7. Fix up the **`if`** statement so it will print!
3. Make the print statement on line 11 print **`True`**.

As you can see on line 7, comparing strings, and variables that contain strings works exactly the same way:

`if` `"Hello" == "Hello"` and **`if`** `greeting == "Hello"` and **`if`** `greeting == other_greeting` are all valid conditions.

14.4. Deal with capitals and spaces

Luckily, Python has some **built-in functions** that help us to deal with **capital letters** and **spaces** so that our code is more **robust** (doesn't break).

Here are a few, although there are lots more:
```
.strip() #removes all spaces from the start and end of the string.
.lower() # converts a whole string to lower case
.upper() # CONVERTS A WHOLE STRING TO UPPER CASE
.title() # Gives Each Word A Capital
.capitalize() # Makes the first letter of the string a capital
```

1. Click and type in a word with some **spaces at the start** and some **capital letters** in it to see the result of `.lower()`.
2. Under the relevant comment, use `.strip()` to remove any spaces.
3. Do the same for `.title()`.
4. Finally, use `.upper()` to convert the word to upper case.
5. Test your code a few times by clicking and entering a **range of words** with and without capitals and spaces.

These functions are important when users are typing text into an `input` and you can't control whether they type capitals or spaces.

14.5. Review quiz on comparing strings

Let's review what we've learned about comparing **strings**!

Review Quiz Questions:

1. Which string is identical to the string in the print statement inside the **`if`** branch?
2. If the user types in `"  SHRUBBERY"`, what will be the value stored in `password` after line 3?
3. Which of these would not result in "Traitor" being printed?
4. What is printed if the user types `"  ShRuBbErY  "`?

15. Numeric input in if statements

15.1. Use numeric input in `if` statements

We learned in lessons 1-10 that if we want to **USE** a number that the user has typed in **AS** a number, we have to use `int()` to convert it.

This is also true if we want to use numbers entered by a user in a **comparison**.

1. Click and enter your height to see what happens. Type **only** the number e.g. **156** not **156cm**.
2. Fix up the `input` statement on line 2 so that the code works.
3. Click again and enter your height to see if you are tall or short.

`TypeError: unorderable types: 'str' > int()` This error means that you can't compare 2 **different types** of data, in this case the **string** you typed in (`str`) and the **integer** (`int`) 165 in the condition.

15.2. Write an `if`/`else` statement using numbers

Let's try another example so we can practice asking for numeric input and comparing it.

1. Ask the user how much time they spend on the internet each day and store it in a **well-named** variable.
2. Write an if statement that checks if the user spends **less than 3 hours** online.
3. If the user spends **less than 3 hours**, print "That's a healthy amount of time".
4. Otherwise, print "You need to get more fresh air!"
5. Click and test **both branches** of your `if` statement.

15.3. Adding more branches with `elif`

Let's use `elif` to add some **more options** based on the user's input. The code below shows how you could check if a person's age is **between 15 and 30**:

```
if age > 15 and age < 30:
  print("You are older than 15 but younger than 30")
```

Remember an `and` condition is `True` if **both** parts are `True`.

What happens if you type 15 or 30? Because we have used `<` and `>`, 15 and 30 (our **boundary** values) are **not included**. If we wanted to include them we would have to write: `if age >= 15 and age <= 30:`

1. Add an `elif` on line 7 that checks if the user spends between 3 and 5 hours online, **including 3 and 5**.
2. On line 8 make it print out "Make sure you are also being active." if the `elif` is run.
3. Read the `elif` and `else` statements on lines 9-12 and replace the `??` with a **boundary** value that makes sense.
4. Click and test **all branches**.
5. **(Optional)** Edit the first `if` so that when the user enters **less than `0`** hours, it will run the `else` branch.

15.4. Make comparisons between 2 stored values

So far we have made **comparisons** using strings, numbers, and variables containing a string or number. We can also compare **2 variables**:

```
retirement_age = 65
if age == retirement_age:
  print("You can retire now!")
```

And this will work whether the variables are **strings**, **integers** or **Boolean**, although they must be the same type if you're using the `<` or `>` operators.

1. Store the number 9 in the **constant** on line 2
2. Complete the condition in the if statement on line 8 using the variable and the constant.

Click here for an explanation of what is meant by a "constant".

15.5. Review Quiz

Time to review `if` statements with numbers!

Review Quiz Questions:

1. What needs to be added to the `input` statement for this code to work?
2. Assuming that bug is fixed, what will be printed if 156 is entered?
3. What will be printed if 175 is entered?
4. What will be printed if -4 is entered?
5. Which of these is a constant?

16. Formatting your code

16.1. Write more good Python code!

There are **conventions**, or guidelines, for writing **conditional** code as well in Python, and this lesson will go over the key points.

In `if`/`elif`/`else` statements, **blocks** must be **indented** for code to run correctly.

You can have as many lines of code as you want inside each branch of your `if` statement, as long as they are **all indented**.

1. In the code editor, lines 8-10 are indented, but they don't line up. **Indent** them all the **same** distance so the code will work.
2. Lines 13-14 should also be a part of the `else` branch. **Indent** them correctly so that **all 3** print statements are printed when the user guesses wrong.
3. Click to check it's all working.

16.2. Use correct spacing and punctuation around conditions

Just like when we do `print` statements, `if` statements have guidelines about where to put spaces.

- **DO** put spaces between each part of your condition:

Use `if age > 16:` rather than `if age>16`

Even if it is long do this:

`if age * 3 > 21 and "b" in "birthday":` not: `if age*3>21 and "b"in"birthday":`

- **Don't** put spaces around the colon `:`

Use `if age > 16:` rather than `if age > 16 :`

- **Don't** use unnecessary brackets in `if` statements (commonly done in other programming languages:

Use: `if "a" in "age" and "b" in "birthday":` Rather than these: `if ("a" in "age") and ("b" in "birthday"):` `if ("a" in "age" and "b" in "birthday"):`

1. Fix up the condition in the `if` statement.
2. Fix up the `elif` statement.
3. Fix up the `else` statement.
4. Test **all 3 branches** of the statement.

16.3. Use constants for values that are set or don't change

We have touched briefly on **constants**. These are variables that **don't change**, and they are an important part of writing good code.

Constants are written in `UPPER_WITH_UNDER` or caps, with underscores separating words. They should be used when a value is set up and **doesn't or shouldn't change**.

1. Read through the code in the editor carefully. Click to see what it does.
2. Decide which variables are numbers that are set and don't change in the program, and **rewrite them as constants.**
3. Click and check it still works.

NOTE: The comments in this block of code are examples of **bad comments** - you will fix these in the next task!

16.4. Write useful comments

Remember how we talked about writing comments that are **short, simple and necessary?** `if` statements are a common place where people write **unnecessary** comments:

```
#If guess equals answer
if guess == ANSWER:
  score += 10
```

This is pointless because if you read the actual line of code below it, you would say exactly the same thing.

A better approach is to write a comment that explains what **the purpose** of that `if` block is, such as:

```
#Check whether to add points for a correct answer
```

1. Read the code in the editor and work out what each part does.
2. Select a better comment for line 4:
3. Select a better comment for line 7:
4. Select a better comment for line 10:
5. Select a better comment for line 17:

16.5. Review quiz

What else have you learned about conventions?

Review Quiz Questions:

1. Which `if` statement is written most correctly?
2. Which is the best comment for these lines of code:
3. How far should code be indented if it's inside a block (e.g. `if`) in Python?
4. We've got another program that calculates pay for workers: everyone works 40 hours/week, some people get paid different hourly rates, some work overtime (extra) hours. Which value is most likely to be a constant?

17. Formatting using an index

17.1. Use **indices** to format strings

We're going to take a short break from `if`/`else` structures in order to look at some more things we can do with `.format()`:

```
print("Hello, {}! You must be {}!".format(name, mood))
```

When we use `.format()` like this, the "missing values" in the brackets (called **arguments** or **parameters**) are inserted **in order**. But each argument is numbered **from `0`**, and we can write that number, or **index**, inside a placeholder to say where it should go:

```
print("Hello {1}, it's {0} today".format("sunny", "Teresa"))
```

Here the arguments are in the **wrong order** inside the `.format()` brackets, but we can insert them into the **correct** placeholders by using the **index**: `"sunny"` is **0**, and `"Teresa"` is **1**.

1. Fill each `{}` placeholder in the print statement on line 1 with the **index** for the correct word so that the quote makes sense.
2. Do the same for the print statement on line 6.
3. Click to check it works OK!

If you're a bit rusty on `.format()` you might like to revisit **lesson 4**.

17.2. Using an index more than once

You might be wondering why we wouldn't just write the **arguments** in order. A better use for this is when you need to insert the **same argument more than once** in a string: `print("{0}, {0}, wherefore art thou {0}?".format("Romeo", "Bob"))`

We can use each argument **as many times** and **anywhere** we want, as long as we put numbers in **all** of the placeholders.

1. Fill in the `{}` placeholders with the correct **indices** on line 1.
2. See if you can reconstruct the famous Monty Python quote on line 3 using the index for each word (Google might help if you get stuck).
3. Click to check it's working.

Remember to **count from 0**!

17.3. Store a string to be formatted in a variable

If you store a **string** containing the `{}` placeholder inside a variable, you can then use `.format()` straight on the variable:

```
greeting = "Hello {}, welcome to The Quiz Game!"
print(greeting.format(name))
```

This could be useful if you need to use a sentence **more than once** in a program, but with **different values**. It can also be used for **constants** - remember they function exactly the same as variables.

1. A template sentence has been stored in the constant on line 2. Write a print statement on line 10 using `.format()` that puts the variables in the right part of the sentence.
2. Click and enter some random words to see if your sentence makes (grammatical) sense!

17.4. Format floats so they have the right number of decimal places

The last **handy tip** we're going to learn about `.format()` for now is how to make **floats** look better when we plug them into a string. Run the example code to remind yourself of what it usually looks like.

In many cases, like money, we might only want 1 or 2 decimal places (dp) like: **$36.75**. This is how we control the format:

```
print("${:.2f}".format(12 / 7))
```

This prints out **$1.71**. The `.2` tells us how many decimal places. **Click here for a little more detail.**

1. Look at the code in the editor and click to see what it does.
2. Hmm that looks a bit weird. Change the print statement on line 6 so that it prints out the hours to **1dp.**
3. Fill in the {} on line 8 so that the wages are printed to **2dp.**
4. Write a print statement on line 14 that prints the wages after tax with **2dp**, in the sentence "After tax: ${}".

17.5. Review quiz!

Hopefully you've got the hang of working with `.format()` now!

Review Quiz Questions:

1. What is the index of the variable age in this list of arguments (the bits inside the brackets) below?
2. What does the following statement print?
3. Which statement correctly prints "I like to eat, eat, eat, apples and bananas" after this line of code?
4. Which of these is the correct placeholder to use if we are formatting money to 2dp?

18. Debugging if statements

18.1. Bug hunt #2

There can be a lot of pesky little **bugs** in `if`/`elif`/`else` structures, so you'd better get some practice at finding them!

1. Find the **1st bug** in the `if`/`else` statement!
2. Find the **2nd bug** in the `if`/`else` statement!
3. You could the code to see what the error message says if you get stuck. And run it again at the end and test it with the **right and wrong answer**.

18.2. Indenting bugs

Python code **must** be **indented** correctly in order to run, otherwise you'll get the dreaded: `SyntaxError: unindent does not match any outer indentation level`

1. Fix up the indents in the `if`/`else` statement!
2. The final `print` statement printing the score should happen **after** the `if`/`else`, for any answer the user types in. Fix this bug.
3. Click and test all the branches of the `if` statement.

18.3. Greater... or less than? No... definitely greater. I think...

Another common bug is getting `<` and `>` mixed up! If we put a **greater than** instead of a **less than**, our code will do **exactly the opposite** of what we want it to do!

Did you know that 10 bees weigh approximately the same as 1 M&M candy?

1. Click to run the quiz question code in the editor and test it with a number **higher than 10**.
2. Test it again with a number **lower than 10**. Are these the expected responses?
3. Fix the 2 bugs in the `if`/`elif`/`else` statement so that the code works properly.
4. Click again and make sure it works!

18.4. Spell those keywords right

Sometimes it's hard to remember exactly what all the **keywords** are in a programming language. Especially if you haven't used the language for a while, or you learn more than one language!

Whoever wrote this code must have gotten mixed up between Python and some other languages!

1. Find and fix the incorrect **keyword** in the **conditional** part of the code.
2. There are 2 more lines of code that have not been written in a very **Pythonic** way. Find them and fix them up!
3. Click and test the code works

Important note: If you use `.strip()` and/or `.lower()` on a variable before using it in a comparison, make sure you put the **lower case** form of the word in the condition! E.g.: `if variable == "right answer":` rather than `if variable == "Right Answer")`

Same idea if you use: `.upper(), .title(), .capitalize()` Make sure you're checking for the right thing!

18.5. Linking `if`s with `else`s

Any time you put in an `if` keyword it starts a **new** selection structure. In other words, you can't have more than 1 `if` in the **same selection code**. If you add an `else`, it will only be linked only with the `if` **directly above it**.

If you need more than 2 options it is **really important** to use the `elif` keyword so that all of the branches are **linked together.**

1. Click and type in "titan" to see **what happens.**
2. Fix the keywords so that it will only give us the "correct" message when "titan" is typed in.
3. Click and check it works!

Click here for an explanation of why this happens.

19. Testing if statement conditions

19.1. Test if statements using `==`

In this lesson, you'll test **if statement** code from the **Knights who say "Ni!"** online training program. Anyone who wants to join the order must complete this induction before actual knighthood training begins.

The application for a knighthood begins by asking the applicant's **location**. Training courses start on **different dates** depending on the location.

1. Click and enter input to run the **1st branch** of the **if statement**.
2. Click and run the **2nd branch** (`elif`).
3. Click and run the **3rd branch**.
4. Click and run the **else branch**.

19.2. Testing `if` conditions with numbers

The **Knights of Ni** training program has a **weight restriction**, so the next question asks the user's weight.

Note that if you type **75kg** you will get an **error** because **75kg** is not a valid number. Type numbers, e.g. **75** (**without kg**) to test this code.

1. Click and run the **1st branch** of the **if statement**.
2. Click and run the **2nd branch.**
3. Click and run the **3rd branch.**
4. Click and run the **else branch.**

Important note: If you just click OK with this code, your will get an **error** because we have not yet learned how to deal with a user typing **non-numerical data** into an input with `int()` around it. Don't panic! We'll get there.

19.3. Test `if` statement conditions

The **Knights of Ni** training course has a **height restriction** too. They are a fussy bunch.

1. Click and test the **1st branch** of the **if statement**.
2. Click and test the **2nd branch.**
3. Click and test the **3rd branch.**
4. Click and test the **4th branch.**
5. Click and test the **else branch.**

As before, if you type **1.5m** you will get an error because **1.5m** is not a valid number. You must type a **1.5** without the **m** on the end.

19.4. Test more complex `if` statement conditions

A potential knight's endurance is tested by their ability to completely **weed and prune** a **shrubbery**.

Shrubs in the **southern hemisphere** are much **tougher** and the weeds grow **twice as big**, so the time limit is different depending on the applicant's location.

1. Click and test the **1st branch** of the **if statement**.
2. Click and test the **2nd branch**.
3. Click and test the **3rd branch.**
4. Click and test the **4th branch.**
5. Click and test the **5th branch.**
6. Click and test the **else branch.**

If you type **10m45s** or **10:45** you will get an error because they are not valid numbers. Type **10.75** instead.

19.5. Test **boundary** values in `if` statements

It is important to test **boundary** values to make sure you have got your **conditions** correct.

The easy way to figure out what the **boundary values** are, is to look at any **ranges** or **limits** you have in your conditions.

In the code in the editor, we have **3 ranges** to test: **1-30, 31-129 and 130+.** The boundary values are the numbers on either side of each end of the ranges - the numbers on the **boundaries**.

This gives us: **0 and 1, 30 and 31, 129 and 130** as boundary values.

1. Click and test the boundary values for the **1st branch.**
2. Click and test the boundary values for the **2nd branch.**
3. Click and test the boundary value for the **3rd branch.**

Notes: The numbers in your conditions are easy to identify as boundary values, but the other boundary values will depend on whether you use `<, >, <=, >=, == or !=` In an `if`/`elif` structure, some of the boundary values will double up. For example, in this code 31 was a boundary value for the 1-30 range as well as the 31-129 range, but you only need to test it once.

20. Review lessons 11-19

20.1. Review the `if` statement

This lesson will review the content you've learned in **lessons 11-19**. For the first task, let's get you to **write some code!**

These lessons taught you about **selection** or **conditional** structures. They are called this because they are the **decision-makers** in a program. They test out a **condition** and then run one of several possible **branches** depending on whether the condition is met or not.

First let's practice a simple `if` statement.

1. Ask the user for their favorite ice cream flavor and store it in a well-named variable.
2. Convert their answer to **lower case** and **strip** any spaces.
3. Write an `if` statement that checks if their favorite flavor is "cookies and cream" and print "Mine too!" if it is.
4. Click and test that the message prints.
5. **(Optional)** Add **concise** comments to your code.

20.2. Review quiz!

To mix it up a little bit, let's do a quiz to review the basics.

Review Quiz Questions:

1. Which operator means **greater than or equal to**?
2. What does the `or` operator mean? For example: `x = 3 > 2 or 9 < 6`
3. What symbol goes at the end of an `if` statement to start a block?
4. Which of these `if` statements is written most correctly?

20.3. Review `if`/`else` statements

The `else` branch, when added to an `if` statement, gives us a **default case**. It gets run if none of the conditions in the `if` or `elif` branches are met.

An `if` statement works fine without an `else`, but an `else` must be attached to an `if` or `if`/`elif`.

Let's write one.

1. Ask the user **how many servings** of fruit and vegetables they have had today and store it in a well-named variable.
2. Write an `if` statement that checks if they have had **5 or more.**
3. If they have, print "Well done, you've had your 5+ today!"
4. Write an `else` branch that says "You should eat 5+ a day, every day."
5. Test both branches using .
6. **(Optional)** Test the boundary values, and add **concise** comments to your code.

20.4. Conditions review quiz

Let's have another go at working out whether conditions are `True` or `False`.

```
Remember: < Less than
> Greater than
<= Less than or equal to
>= Greater than or equal to
== Equal to
!= Not equal to
or - is True if either or both sides are True
and - is True only if both sides are True
not - reverses it e.g. True becomes False
```

Review Quiz Questions:

1. Is this condition `True` or `False`?
   `345 < 274`
2. Is this condition `True` or `False` if **x = True**?
   `3 * 3 >= 4 + 5 and x`
3. Is this condition `True` or `False` if **x = False**?
   `9 - 7 < 1 or not x`
4. Is this condition `True` or `False` if `phrase` = **The noblest of all dogs is the hotdog; it feeds the hand that bites it**?
   `"c" in phrase`

20.5. Review `if`/`elif`/`else` statements

Lastly, let's practice writing an `if`/`elif`/`else` statement one more time.

Remember that if you want **all** of your **branches** to be **linked together** so that only one is run, you need to use `if`/`elif`/`else` not multiple `if` statements. Otherwise the `else` will only be attached to the `if` right before it, and **more than one** branch will run.

1. Ask the user to rate the last movie they watched out of 5 and store it in a well-named variable.
2. Write an `if` statement that prints "Pretty average huh?" if the rating is **3.**
3. Write an `elif` branch that prints "Wow, that must have been bad!" if the rating is **less than 3.**
4. Write another `elif` that prints "Sounds like a great movie!" if the rating is **more than 3** and **less than or equal to 5.**
5. Write an `else` branch that prints "I said out of 5!" for anything else.

Don't forget: It's important to test the **boundary** values once you start writing conditional structures, especially ones with multiple branches.

21. Introduction to for loops

21.1. Introduction to loops

In **lessons 11-20** you learned about **selection** (also known as **conditional**) structures that let your code make decisions.

In **lessons 21-30** you will learn about **iteration** or **code that repeats**. These structures are also known as **loops**.

We will look at 2 types of loops: **for loops** and **while loops**.

Loops are useful if we want to do something **multiple times** in a program.

1. Look at the code in the editor and try to guess what it might do.
2. Click and see what it does.

We had to write out that part that asks for the number of hours worked **5 times**! If only there was a way to make that easier...

21.2. Meet the `for` loop

Let's take a look at our first loop... the `for` loop:

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

A **for loop** repeats the **block** of code inside it a **set number** of times.

`i` is the built-in **counter** that keeps track of how many times the loop has repeated.

1. Run the code in the editor to see how `i` changes as the loop is run.
2. Change the loop statement so that the numbers **0-9** are printed.

Note that because it **starts from 0**, the last number printed is always **one less** than the number we put in `range()`.

21.3. Customise a `for` loop

We can customise a `for` loop in several ways including making it start and stop at **different numbers.** You will learn more about this in the next lesson.

1. Click to see what the code in the editor does.
2. Compare the output to the code and see if you can figure out how it works. Make sure you don't change these 3 loops.
3. Add a for loop to line 18 where the **first number** printed is **5** and the **last number** printed is **17**.
4. Click to see if you got it right!

21.4. Meet the `while` loop

The `while` loop can be used in a similar way to a `for` loop for printing out numbers, but there are some differences.

In a `while` loop, instead of saying how many times we want the loop to repeat, we use a **condition** -- just like in an `if` statement. The loop will run until the **condition is met**: `while i <= 5:`
  `print(i)`

This is the basic structure of a `while` loop, but we have to write a bit more code for it to work because it doesn't have a **built-in counter** like the `for` loop.

1. Look at the code in the editor and see what we have to do with `i` to make it work.
2. Change the code so that the numbers **0-10** are printed.
3. See if you can change the code so that it **starts from 1**.
4. **(Optional)** Change the conditional **operator** so that **10** is **NOT** printed.

So we have to set up the counter first, with the start point, and then **increment** it inside the loop.

21.5. Use other conditions in a `while` loop

Because a **while loop** runs on a **condition**, we don't have to use just numbers. We can write any condition we want in there, but be careful because something like: `while 3 < 4` will make an **infinite loop**, because it will never be **False**, and so won't get switched off!

Have a look at the code editor to see a condition based on the user's input.

1. Click to see what the code does. Type "no" a few times. Then type "yes".
2. Change the condition in the `while` statement so that the loop repeats as long as **response is NOT equal to "yes"**.
3. Click again and test it the same way as in step 1 - it should work the same way.

Code explanation: Although both versions of this loop have the same results for **yes** and **no**, in some cases there's a good reason to write the condition one way as it can make a difference to other values.

Click here for an example.

22. Customising for loops

22.1. Use for loops for printing numbers

Let's take a closer look at the **for loop**:
```
for i in range(5):
  print(i)
```

You can put any code inside the loop, not just `print(i)`. To do this you just **indent** any code that should be repeated - like `if` statements.

Let's try a few more examples to help show how it works.

1. Change the `for` loop on lines 2-3 so that it prints out the numbers **0 - 8.**
2. Write a `for` loop on lines 6-8 that prints out "I will practice coding every day!" 5 times.
3. Give yourself 3 cheers for learning `for` loops by putting **2 print statements** in the loop on lines 12-13, one saying "Hip hip..." and the second saying "Hooray!"

Click here for a more detailed look at the parts of a for loop.

22.2. Customising the start and end points

The **syntax** for a `for` loop statement is this:

```
for i in range(start, stop, step):
```

Where:

`start` is the number you want the loop (or `i`) to **start** at,

`stop` is the number you want it to **stop** at, and

`step` is how much `i` should **change by** each time through the loop.

We will customize `step` in the next task!

1. Write a for loop on lines 2-3 that prints the numbers from **1-10.**
2. Write a for loop on lines 7-8 that prints the numbers from **20-35.**
3. Write a for loop on lines 12-13 that prints the numbers from **9-18** in the format "The next number is 9"- **tip.**

Note: The empty `print()` statements just put a gap between the output from each loop so you can see where each starts and ends. Make sure you don't remove them!

22.3. Customise the step

Now let's play with the `step` **argument**.

The `step` value lets us say how much `i` should **change by** on each **iteration**, or **pass** through the loop.

We can use any **positive OR negative** number in `step`. For example:

`for i in range(0, 20, 2):` counts up in **2s**

`for i in range(5, 35, 5):` counts up in **5s**

`for i in range(21, 0, -3):` counts **DOWN** in **3s**:

Pretty simple but very useful (and fun)!

1. Click to see the initial code for **Loop #1** in action.
2. Under the **Loop #2** print statement, write a `for` loop that counts **up in 3s** from **6 - 21.**
3. Under **Loop #3**, write a `for` loop that counts **down** from **10 to 1** (just in 1s).
4. Under **Loop #4**, write a loop that counts **down** from **100 to 0 in 10s.**

Remember that it still won't reach the `stop` value, even if counting in a different step.

22.4. Using the `i` variable

The `i` variable is just like any other **variable**. It stores an **integer** which can be used the same way as any other integer variable - in calculations, conditions, `.format()` and so on.

You can, in fact, call it **any valid variable name** too. You could write:

```
for spaghetti in range(3):
  print(spaghetti)
```

But it doesn't make as much sense!

Click here for some examples of when this IS useful.

1. Write a for loop statement on line 2 that will count from **1 to 12 inclusive.**
2. Inside the loop, on line 3, write a print statement that uses `i` to print out the **3 times table**. You should print each line as "3 x 1 = 3" and so on using `.format()`.
3. Click to check that it works!

22.5. Review quiz!

Let's review `for` loops with a quiz.

Review Quiz Questions:

1. How many times will "Ni!" be printed out with the following loop?
2. What is the last number that the following loop will print out?
3. What is the 3rd number that the following loop will print out?
4. What is the output of the 4th time through this loop?

23. Refactoring turtle graphic loops

23.1. **Refactor** turtle code using loops

Let's play with some more **turtle graphics**. **For loops** work exactly the same way with turtle graphics as they do with any other Python code.

In the editor you will see the code from lesson 6 for **drawing a square**. It's very **repetitive** so let's see if we can improve it.

1. Click to remind yourself of what it does.
2. **Refactor** the code using a for loop so that it is shorter.
3. Click to check that `tiny` still draws a square!

23.2. Use a **list** to run a `for` loop

We saw that we can **iterate** (loop) through each letter in a word by using: `for letter in "Xylophone"`

We can also make a **list**, and loop through each **item** in the list automatically. A list looks like this: `foods = ["cereal", "pizza", "chocolate", "cake"]`

And a loop that runs **using the list** looks like this:
```
for food in foods:
  print("{} is yum!".format(food))
```

1. In the editor a list called `colors` has been added. Add the colors "blue" and "green" to the end of the list (in that order).
2. **Refactor** the `for` statement in the square code again so that it loops through **once for each** `color` in `colors`.
3. On line 11 (before `tiny.forward(150)`) set `pencolor()` to the current `color`.

23.3. Use loops to draw other shapes

Let's practice drawing some **other shapes** using `for` loops.

The *`#Go to position`* code has been used between exercises to make sure the shapes don't overlap.

For all shapes, use `.left()` to turn, and `100` as the length of the sides.

1. Under the correct comment, set the `pencolor()` and draw a **hollow red hexagon** using a for loop.
2. Under the correct comment, set the `pencolor()` and `fillcolor()` and use a for loop to draw a **triangle with a blue line and yellow fill** (**tip**).
3. Under the correct comment, draw a **pink pentagon with pink fill**.

Hint: If you're not sure how far to turn for the **hexagon** or **pentagon** this is the rule:

"The external angles of polygons add up to 360".

So divide 360 by how many sides the shape has!

23.4. Spirographs!

Now let's have some fun!

Because we are using **loops**, we can do some cool things that would take us a really long time to do if we had to write it all out line by line.

Let's have a go at drawing some **spirograph**-like designs.

1. Click to see what the code in the editor does.
2. Change the number `19` in the for loop to `100`.
3. Change the angle inside `.left()` to a number close to `180`.
4. Try adding in a second `.forward()` and a `.left()` or `.right()` with a different distance and angle and after `.left()`.
5. Experiment however you want!
6. Under *`#Change colors`* add an `if` statement that checks if `i == 20` and sets `.color()` to **red** if so.
7. Add `elif` branches that do the same for these: **40: orange, 60: yellow, 80: green** .
8. Finally, set the *`#Draw the spirograph`* code to the code in the note below. Make sure it is **indented** so it is inside the loop and click to finish the task.

Final movement code:
```
spiro.forward(200)
spiro.left(184)
spiro.forward(40)
spiro.right(30)
```

23.5. Random numbers and **RGB** colors

Sometimes we want to pick a random number, and Python has a built in `random` module for this purpose. `random` has tons of functions, but we're just going to look at random numbers for now:
```
import random #Have to import first
my_num = random.randrange(1, 11)
#randrange() boundaries work the same as range()
#This chooses a number from 1 to 10
```

In Python's `turtle` module, we can also set the color using numbers, by giving it a number from 0-255 for **red**, **green** and **blue**:
```
#RGB color format: t.color((red, blue, green))
timmy.color((135, 14, 212))
```

Let's try out both of these with our spirograph!

1. In the editor is the code for a spirograph, without the code for changing the color.
2. On line 10, create a variable called `red` and set it equal to a random number between **0 and 255 inclusive**.
3. On lines 11-12, create two more variables called `green` and `blue` and set them to a random number between 0 and 255 too.
4. On line 13, set `spiro`'s `color()` to the random color using the `red, green, and blue` variables.
5. Click several times to see the result.

Note on brackets - ().

24. Complex for loop example

24.1. Working through an example

Let's work through a problem that needs a `for` loop, in order to see how one might be used in practice.

In this lesson, you will build a program that calculates the **total** and **lowest** number of hours a user has spent online in the past week.

1. Firstly, let's ask the user for their name and store it in a variable called `name`, on line 2.
2. Now on line 3, greet them with "Hello, ... !" using a print statement, where **...** is their name.

24.2. Create the for loop

Now we want to ask the user how many hours they have spent online **each day**, for the past **7 days**. We could write out 7 different input statements and use 7 different variables, but that would take a long time!

The **rule of 3** says we should use a loop if we're going to have to repeat something **3 or more** times.

1. On line 6, write a `for` loop statement that will repeat our code **7 times** (we need `i` to go from 1 - 7).
2. Inside the loop, print `i` and click to check it works.
3. Delete this print statement and replace it with an input that asks the user "How many hours did you spend online on day 1? " and stores it in a variable called `hours`. The next time through the loop it should say "day 2" and so on.
4. Underneath this, add `print(hours)` just to check it is storing the numbers.
5. Click to check that the numbers you enter are printed correctly in the console.

24.3. Adding up the total hours

The next step is to calculate the **total number of hours** they have entered.

1. On line 5, create a variable called `total_hours` and set it equal to 0.
2. Inside the loop, delete `print(hours)` and replace it with a calculation that adds the **number of hours** that have been entered to `total_hours`.
3. We need to use the input as a number, and they might say 3.5 or similar, so we had better put `float()` around our input.
4. Print out `total hours` **AFTER** the loop in the sentence "You spent {} hours online in total".
5. Test your code and type in at least 1 **float** value.
6. **(Optional)** Format the float output in the final print statement to **1dp**.

24.4. Finding the lowest number

Finally, we are going to be really clever, and combine a **selection** structure with our **iterative** or **loop** structure! We're going to use an `if` statement to find the **lowest number of hours.**

This task uses a variable that stores a special value called **None**. This basically creates an **empty** variable - we have given it a name, but there is nothing stored inside it when we create it.

1. On line 6, add a variable called `lowest_hours` and set it equal to **None**.
2. Inside your loop, write an `if` statement that will check if `lowest_hours` equals `None` **OR** whether the number of hours entered is **less than** `lowest_hours`. If you do not check these conditions in this order your program will not work.
3. If this is true (inside the `if`), set `lowest_hours` equal to that number of hours.
4. Add a print statement at the end of your program that says "The least time you spent online in one day was {} hours"
5. Test your code!
6. **(Optional)** Format the float output to **1dp**.

24.5. Review quiz!

Let's just double check that you understand for loops!

Review Quiz Questions:

1. What symbol goes at the end of a `for` loop statement to start a block?
2. How many times does this loop repeat?
3. What is the first number this loop prints out?
4. What is the last number this loop prints out?

25. Introduction to while loops

25.1. Get to know the `while` loop

It's time to take a closer look at the **while loop**.

A `while` loop can be used to do the same things as a `for` loop, but it can take a couple of extra lines of code: `i = 0`

```
while i < 10:
  print(i)
  i += 1
```

It is important to understand how to run a while loop with a **counter** though, so this lesson will let you practice that before we look at other uses for a while loop!

Click here for an explanation of the code above.

1. Change the `while` loop on lines 2-5 so that it prints out the numbers **0 - 8.**
2. Write a `while` loop starting on line 8 that prints out "I will practice coding every day!" 5 times.
3. Give yourself 3 cheers for learning `while` loops by putting **2 print statements** in the loop on lines 17-18, one saying "Hip hip..." and the second saying "Hooray!"
4. Add **1** to `i` after the print statements so that you don't get an **infinite loop**!

25.2. Customise the start and stop points for a `while` loop

Just like a `for` loop, we can also customise the **start and stop** points for a `while` loop. For example, a loop that starts at **5** and stops at **15**: `i = 5`

```
#This is our start point
while i <= 15: #This is our end point
  print(i)
  i += 1 #Don't forget this!
```

Because we can write **any kind of condition** we want in a `while` loop, we can use `<=` in this code and write **15**, instead of in a `for` loop where we have to write **one number higher** than the last one we want printed.

1. Write a `while` loop under the relevant comment that prints the numbers from **1-5.**
2. Write a `while` loop under the relevant comment that prints the numbers from **10-25.**
3. Combine the variable `i` with a print statement using `.format()` by writing a `while` loop under the relevant comment that prints the numbers from **9 to 18** in the format: "The next number is 9".

Note: It also works perfectly fine if you write the example loop this way, using `< 16`: `i = 5`

```
#This is our start point
while i < 16: #This is our end point
  print(i)
  i += 1 #Don't forget this!
```

25.3. Customise the step in a `while` loop

Now let's play with the **step** in a `while` loop!

Remember that the step is how much `i` should **change by** on each **iteration**, or pass through the loop.

To change the step in a while loop, we simply change the line: `i += 1`

We can change it to pretty much any kind of calculation that changes `i`:

```
i -= 3
i *= 2
i = i * 3 - 2
```

These will all give quite different results. We can also use any of these in our conditions: `<, >, <=, >=, ==, !=, and, or, not`

1. Click to see the initial code for **Loop #1** in action.
2. Under the **Loop #2** print statement, write a `while` loop that counts **up in 3s** from **6 - 21.**
3. Write a `while` loop that counts **down** from **10 to 1** (just in 1s).
4. Write a `while` loop that counts **down** from **100 to 0 in 10s.**

Note: If you use `<` rather than `<=` it will not print the `stop` value, even if counting in a different step.

25.4. Using the `i` variable

Just like in a `for` loop, the `i` variable can be used in calculations, substituted into a string using `.format()` and so on inside a `while` loop.

You can also call it any valid variable name, but again it's best to use `i` if it's just a simple **counter**, or choose something that **makes sense** rather than:

```
peanut_butter = 0
while peanut_butter <= 10:
  print(peanut_butter)
```

Click here for some examples of when this IS useful.

1. Create a variable `i` and write a while loop statement on lines 2-3 that will count from **1 to 12 inclusive.**
2. Inside the loop, write a print statement that uses `i` to print out the **3 times table**. You should print each line as "3 x 1 = 3" and so on using `.format()`.
3. Don't forget to **increment** `i`!
4. Click to check that it works!

25.5. Review quiz

Now let's review `while` loops with a quiz!

Review Quiz Questions:

1. How many times will "Hello!" be printed out with the following loop?
2. What is the last number that the following loop will print out?
3. What is the 3rd number that the following loop will print out?
4. What is the highest number that the following loop will print out?

26. Alternate loop counters

26.1. Run a `while` loop with a variable other than `i`

Sometimes we might want to run a loop that depends on a variable that has a purpose in our code, rather than just a generic counter like `i`. For example, if we're giving a user 3 tries to guess a password:
```
num_tries = 0
while num_tries < 3:
  password = input("Password: ")
  if password == "ekki-ekki-ekki":
    print("Correct!")
    break
  else:
    print("Incorrect.")
    num_tries += 1
```

In this case, rather than `i` it makes sense to have a variable name that shows what the loop is doing. This is part of writing **well-documented** code.

Let's try an example.

1. Write a `while` loop statement on line 8 that will make the loop run as long as the user has lives left.
2. Inside the loop, ask the user to guess the number you're thinking of and store it as `guess`.
3. Write an `if` statement that checks if the user's guess is correct and says "Correct!" if so.
4. Add an `else` that prints "Wrong!" if the user is incorrect and removes one life then your code.
5. **(Optional)** Set `lives` to 0 if they get it correct, so that the loop will stop running.

Note that the `break` keyword is used to exit the loop early. We'll cover this in the next lesson.

26.2. Run a `while` loop with user input

We might want our loop to depend on what a user types in. The example code shows one way we could do this.

In this example, the **user's input** of a number is used to **change the variable** `total_servings` which is **running the loop.** Once the user's input adds up to **5 servings**, the loop will **stop.**

A while loop is useful here; we don't want it to **run 5 times** because the user might type **5** as their first input!

1. Read the code in the code editor.
2. On line 5, write a `while` loop statement that will repeat until the user has **no pocket money** left.
3. Inside the loop, ask the user for the price of the item they want to buy and store it as `price`.
4. Subtract the price from their pocket money and then tell them "You have ${} left". Don't forget to format the **float** output as money!

We will improve on the **logic** of this in the next couple of tasks, because right now they can spend as much as they want the last time and end up **in debt!**

26.3. Use a Boolean variable to run a `while` loop

We don't just have to use greater thans and less thans to run our loops; we can also use **Boolean** variables.

Instead of storing the result of a comparison like this:

```
my_var = 3 <= 5 #my_var is True
```

We can store either `True` or `False` straight into a variable:

```
repeat = True
```

We can then run a loop using : `while repeat == True: #As long as repeat is True`
```
  #Do something
```

We can then "switch off" the variable somewhere inside the loop by setting it to **False.**

1. Create a variable on line 3 called `still_shopping` and set it to `True`.
2. Write a condition in the `while` statement on line 6 that will keep our loop going as long as the user is `still_shopping`.
3. On line 12, ask the user if they would like to keep shopping and store the response in a variable called `confirm`.
4. Below this, write an `if` statement that sets `still_shopping` to `False` if the user says "no".
5. Click and test your program with both "yes" and "no" to keep shopping.

26.4. Use more complex conditions to run a `while` loop

Uh oh, now our shopper can spend **as much as they want** and get into as much **debt** as they like! We need to fix this.

In task 2 we stopped the loop when the user ran out of money, and in task 3 we stopped it when they said they were done. Now we'll **combine the two** so that the program makes more sense.

Are you ready?

1. Using the `and` operator, make the loop run as long as the user is still_shopping **AND** they still have pocket money left.
2. Now let's stop them getting into debt! On line 10, write an `if` statement that checks whether their **pocket money minus the price** they entered is **$0 or more.**
3. **Indent** the calculation on line 11 in so that it only happens if they **can afford** the item.
4. Add an `else` branch that prints "You can't afford that".
5. Click and test your code with all the tests listed in the note below.
6. **(Optional)** You might like to put the **confirm** segment of code inside its own if statement so that it will only be run if the user **still has money**.

Tests:

1) Using up exactly all of your money.

2) Spending some and then saying "no" to keep shopping.

3) Trying to spend more than you have.

26.5. Use conditions to make sure input is within the right boundaries

Another good use for this kind of loop is to force the user to enter **valid input.** Let's look first at how to get the user to enter a number within a certain range.

Say we want the user's **age**. A sensible range for this might be **0 - 100**. We don't want them to enter a **negative** number, or tell us they're **225.**

We can use our loop conditions to do this, by making the loop repeat if they enter either of these values:
```
age = int(input("How old are you?"))
while age < 0 or age > 100:
  print("Please enter a valid age.")
  age = int(input("How old are you?"))
```

This loop will repeat until they enter a valid number **between 0 and 100.** Here's an example for you to try:

1. Ask the user how many minutes of exercise they did today and store the response in a variable called `time`. Don't forget `int()`!
2. The number of minutes in a day is **1440**. Write a `while` loop that will continue until they enter a value from **0 to 1440.**
3. Inside the loop print "Please enter a valid number. The total minutes in one day is 1440".
4. Prompt them again for their exercise time.

27. Using break and continue

27.1. Use the `break` and `continue` statements with for loops

In the last lesson we saw the word `break`. **Break** is used to exit a loop early, usually if a condition is met:
```
for i in range(10):
  print(i)
  if i == 8:
    break # Exit the loop after 8 has been printed
```

This `break` statement should be **used sparingly** because if it's not used well it can make your code really messy and hard to debug.

We also have `continue` which will **skip any lines of code after it** in the loop for that round or **pass**, and start again at the first line of code in the loop for the next **iteration**.

1. Click to see what the code in the code editor does - **look closely** at the output.
2. Change the loop on lines 2-5 so that it stops running after the number **13** is printed.
3. Change the loop on lines 10-13 so that it skips printing the number 9.

27.2. Using an `else` with a for loop

Python has a cool feature where you can add an `else` clause to a for loop! That might sound weird, but you'll see what it does if you the example code.

The `else` branch runs only if the loop is **NOT broken**, in this example, if the thing we're looking for is **not found**.

This gives us a way to make our **Knights of Ni** online training password program much more logical.

1. Complete the `for` statement on line 4 so that the loop will run **3 times.**
2. If the user guesses the password correctly, print "Correct, authorisation complete."
3. After this line add a `break` statement to **exit the loop** since they were right (still inside the `if` branch).
4. The first `else` branch goes with the `if` and runs if the guess is incorrect, so print "Incorrect" in here.
5. The second `else` on line 12 runs if the loop exits normally without being broken, in this case if the user fails 3 times. Print ""Sorry, 3 incorrect guesses, your account has been locked." in this branch.
6. Click and test your code!

27.3. Use `break` with a `while` loop

Because a `while` loop checks a condition, we can usually just **change a variable** inside the loop to switch it off.

Sometimes we might want to do something small, however, that we don't want to make a whole new variable for. In this case we might use `break` with an **infinite loop**.

An infinite loop can be done in many ways (often accidentally!) but a deliberate one might look like this:
```
while True:
  #do something
```

We can't ever switch it off because the only part to our condition is the word `True`, which is always **True!**

Let's look at forcing a user to enter valid input (**validation**).

1. Complete the `while` statement so it is an **infinite loop** (line 2).
2. If they type in "true" or "false", break the loop (line 6).
3. Otherwise print "Please answer with true or false" (line 8).
4. Click and test your code several times to check all of the branches.

Note that this code is just a snippet; it doesn't actually check if they are right or wrong, just that they have entered a **valid** answer. This is important for writing **really good code**!

27.4. Use `try`/`except` to force numerical input

At the moment, when we use: `int(input("Enter a number: "))` our program crashes if we type in letters, or leave it blank - you can test this by running the example code.

To get around this, we use `try` and `except`. It looks like this:
```
while True:
  try:
    number = int(input("Enter a number: "))
    break
  except ValueError:
    print("Please enter a number!")
```

Click here for an explanation.

1. Create an infinite loop using the `while` statement on line 2.
2. This time we're expecting a height such as **1.8**, so add `float()` to the input on line 4.
3. If the user enters invalid data (in the `except` branch), print "Please enter a valid height in metres e.g. 1.65".
4. Click and test the program with a word, an **integer** and a **float**.

Note: If you think the input might be an integer or a float, then use `float()`.

If you don't want to accept decimals, then use `int()`.

27.5. Review quiz time!

Let's review `break` and `continue`.

Review Quiz Questions:

1. What does the `break` statement do?
2. What does the `continue` statement do?
3. What does the following code print out?
4. What are the keywords that can be used to write code that will force a user to enter a number?

28. Testing for infinite loops

28.1. Testing expected values

In this lesson you will learn about testing your code. It's **really important** to know how to test your code thoroughly if you want to write **good code** that works.

There are generally **3 types** of input values that we test. For example, if we're asking the user for the **number of hours** they exercised for today:

- **Expected** values - the values you expect your user to enter. In this case **0-24 inclusive** (24 hours in a day).
- **Boundary** values - data that is on either side of the acceptable limits. In this case **-1 and 0** and **24 and 25**.
- **Invalid** or **exceptional** values - values your program should **NOT** accept. In this case for example: **three**, **-3**, **lots**.

1. Click twice, and test this code with an **expected** value.
2. Click again and try typing in **25**.
3. Click once more and try typing in **three.**

This code works fine for **expected values**, but it **crashes** on **invalid values** containing letters, and should make us **try again** if we type in an outer **boundary** like 25 or an **invalid number** like -3.

Extra Note.

28.2. Test boundary values

Here is a new version of our code that has been set up to handle the **boundary** input cases properly.

This code snippet now **only** accepts numbers between **0 and 24**. Let's test it out!

1. Click and type in the first boundary value **-1**, then the value **0**.
2. Click and type in the first boundary value **25**, then the value **24**.
3. Click one more time and enter an **invalid** value that contains letters.

Usually you would pick a sample of expected and invalid values to test your program on, but it's best to test **ALL** boundary values because it's the only way to be sure that you've got all your **conditions** right.

28.3. Test invalid or exceptional values

Now we just need to make it so that our program doesn't crash when an **invalid** value that contains **text** is entered!

Remember `try`/`except`? We're going to use that here.

This version of our code uses the `while True:` loop to force valid input. If the user enters a number, we will `try` to **convert** it using `float()`, and if that works **then** we will check if their number is in the right range. If not, we ask them to enter a number in the right range and **stay in the loop**. Otherwise (`else`), if they have entered a number in the right range, we will **break.**

The `except` block runs if the number can't be converted using `float()`, and gives them a specific error message. This code handles all **expected**, **boundary** and **invalid** values.

1. Replace the ??? to complete the `if` statement on line 5 with a condition that will make sure the user's input is within the **boundary** values.
2. Replace the ??? on line 9 with the correct keyword.
3. Click and test the program with a **boundary** value that is **outside** the boundary then with an expected value.
4. Click and test the program with an **invalid** value that contains letters then with an expected value.

Don't worry if the code is a little bit confusing, this is getting quite complex! If you are keeping up, well done.

28.4. Other ways of testing input data

Python has some other built-in **functions** that we can use to find out information about **strings**:
```
.isnumeric() #True if a string contains only numbers
.isalpha()  #True if a string has only letters
.isalnum()  #True if a string has only letters and/or numbers (alphanumeric)
```

An example of where this might be useful is if you wanted to force a user to enter **text only** e.g. a **name**, or **numbers and letters only** e.g. a **serial number**.

1. The code in the editor forces the user to enter a name with no numbers in it. Click and test it with "9000" to see what happens.
2. Click again and test it with "R2D2".
3. Click once more and enter a valid name.

28.5. Review quiz

Let's review what we've covered about testing.

Review Quiz Questions:

1. Which of the following is (only) an expected value for this code?
2. Which of the following is an acceptable boundary value for this code?
3. Which of the following is an unacceptable boundary value for this code?
4. Which of the following is an invalid value for this code?

29. Debugging loops

29.1. Bug hunt #3

It's time to practice finding bugs in loops!

Many of the bugs you'll come across are similar to those that are common in **sequential** and **conditional** code - **spelling, punctuation, indenting, getting the conditions right** and so on.

Most bugs don't cause a program to crash but they make it do the wrong thing. Read the comments in the code carefully to help you find all of the bugs.

1. Find the bug in the for loop on lines 2-3.
2. Find the 2 bugs in the while loop on lines 6-9.

29.2. Laying out code correctly

It's important to lay out your code well with proper punctuation, brackets, **operators** and **white space**.

1. Fix the bug on line 5.
2. Fix the bug on line 9.
3. Some of the indenting is wrong. This code should be a `for`/`else` loop with an `if`/`else` inside it. Fix up the indenting so that it works properly.
4. Click and test it to make sure it works correctly.

29.3. Debug `while` loop conditions

It's very easy to make a mistake with the **condition** in a `while` loop, especially with the boundary values.

1. Fix the bug on line 2.
2. Click and test the **boundary** values for the loop.
3. Fix the condition on line 4 so that the loop only accepts valid values.
4. Which value was being accepted when it shouldn't have been?

29.4. Debugging unintentional infinite loops or loops that don't run

Sometimes we use an **infinite loop** for a reason, but often they can happen accidentally because we mess up our code.

This **interpreter** will only print out so many lines of an infinite loop, but in a normal program an infinite loop would cause the program to **crash or freeze!**

1. Fix the bug that is making the first loop infinite!
2. Fix the bug that makes the second loop not even run at all.

29.5. Debugging intentionally infinite loops!

Now, if we actually want a loop to be infinite and repeat forever until we get the data or result we are looking for, then there's something really important we have to remember to do when we find it!

1. Click and test this with some **invalid** numbers or words.
2. Click and type in an **expected** value. Hmm...
3. Fix the bug that is making this loop repeat even with **valid** input.

30. Review lessons 21-29

30.1. Review loops

Time to review lessons 21-29 and make sure you understand **iteration**, or loops!

Loops are also called **iterative** programming structures because they make code **iterate** or repeat. Each time that you go through a loop it is called an **iteration** or sometimes a **pass**.

Click here for a `for` loop summary.

Click here for a `while` loop summary.

1. Write a for loop that prints the numbers from **2 - 18**.
2. Write a while loop that does the same.

30.2. Loops review quiz

Let's mix it up a bit with a quiz!

Review Quiz Questions:

1. Which `for` loop statement would accurately print out the numbers 3 - 7?
2. Which `for` loop **statement** would accurately print out "Hello" 7 times?
3. What is used inside a `while` loop to make it count up in 2s?
4. When is a `for` loop usually used?
5. When is a `while` loop usually used?

30.3. Review using the `i` variable

We can use `i` (or whatever we have called a variable that runs our loop) inside the loop just like a **regular variable.**

It's important to know what you want to use `i` for, though, so you can write your loop appropriately.

1. Click and look at the code and the output.
2. Fix the `range()` in the first loop so that it prints out "day 1, day 2" etc.
3. The `for` loops on lines 7-9 should print out the **1-3 times tables** starting with "1 x 1 = 1" and ending with "3 x 3 = 9". Fix up the `range()` in each so it works properly.

Click here for explanation of the times tables loops.

30.4. Another review quiz!

How much do you remember?

Review Quiz Questions:

1. Which is the best description of this `for` loop?
2. What is a **Boolean** variable?
3. What does this code draw?
4. What does this code draw?

30.5. Review writing `for` and `while` loops that depend on user input

In this final task you have one more chance to write some loops that are affected by **user input**.

1. Under the `#For loop` comment, ask the user to enter a number and store it as `maximum`.
2. Write a for loop that will print out all the numbers from 0 up to **but not including** the user's chosen number.
3. Under the `#While loop` comment, ask the user "What is the coolest programming language?" and store their input as `answer`.
4. Write a while loop that will force them to keep answering until their answer is "python".
5. **(Optional)** Change your while loop so that it will check if their answer **contains** the word python, so that answers like "Python for sure!" and "it's python!" are accepted (hint: you might find keywords like `and`, `or`, `not`, `in` etc. useful here).