Python Turtle - Lesson 4

Page Content

> **In this lesson you will learn:**
>
> - about coding modularisation
> - when and how to use functions in Python
> - how to accept user’s input into your code
> - about data types
> - how to convert between data types

Part 1: Functions

Video link

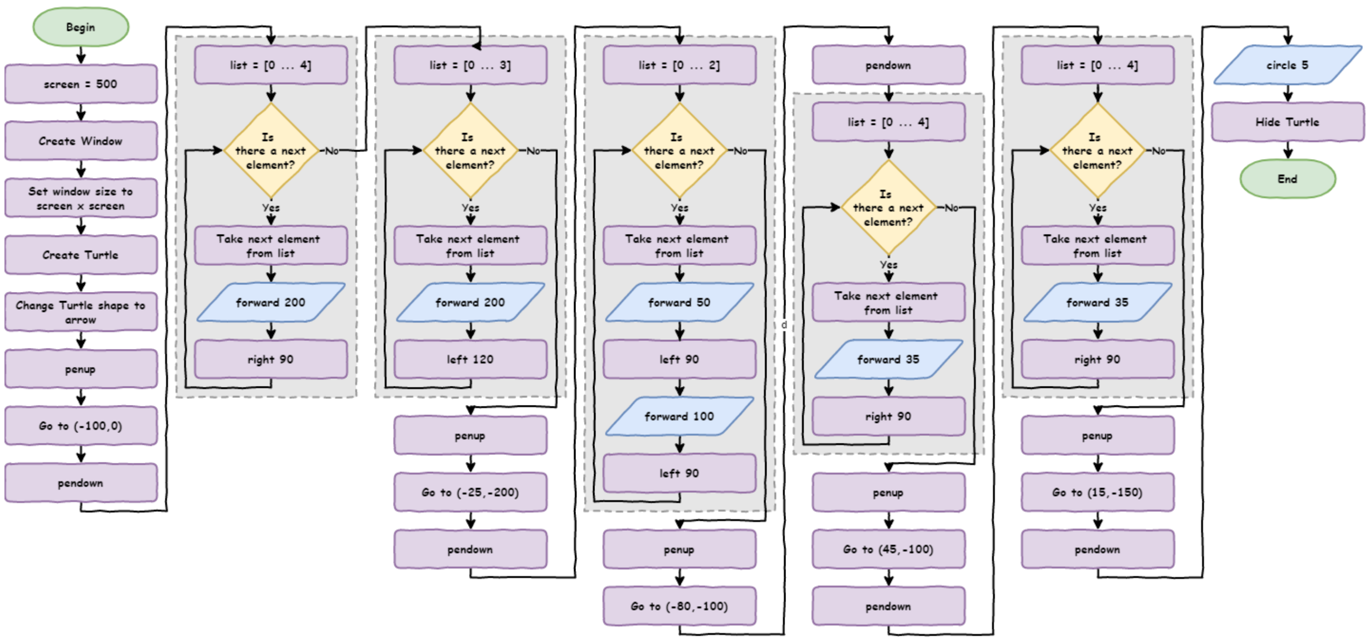
What are functions?

Functions are blocks of code that we can run several times in our program. So far in our programming, all our blocks of code are only run once. Even loop blocks are only run once. They repeat the code inside the block, but once the program has passed the loop, it won’t go back and run it gain.

With functions, we move a block of code outside of the main program sequence, then give it a name. The program can use that block as many times. To use it, the program **calls** the function name from within the main program sequence.

To understand this more clearly, we will start with my solution for **lesson_3_ex_4.py**.

Here is the flowchart for the solution:



Below is the code. You could type it into a new document, or you could just download the `lesson_4_pt_1.py` file.

```python
import turtle

# set up screen
screen = 500
window = turtle.Screen()
window.setup(screen, screen)

# create turtle instance
my_ttl = turtle.Turtle()
my_ttl.shape("arrow")

####################################
## Using the tutrle command you ##
## have learnt, draw a house.   ##
####################################

# move pen
my_ttl.penup()
my_ttl.goto(-100, 0)
my_ttl.pendown()

# draw square
for index in range(4):
    my_ttl.forward(200)
    my_ttl.right(90)

# draw triangle
for index in range(3):
    my_ttl.forward(200)
    my_ttl.left(120)

# move pen
my_ttl.penup()
my_ttl.goto(-25, -200)
my_ttl.pendown()

# draw rectangle
for index in range(2):
    my_ttl.forward(50)
    my_ttl.left(90)
    my_ttl.forward(100)
    my_ttl.left(90)

# move pen
my_ttl.penup()
my_ttl.goto(-80, -100)
my_ttl.pendown()

# draw square
for index in range(4):
    my_ttl.forward(35)
    my_ttl.right(90)

# move pen
my_ttl.penup()
my_ttl.goto(45, -100)
my_ttl.pendown()

# draw square
for index in range(4):
    my_ttl.forward(35)
    my_ttl.right(90)

# move pen
my_ttl.penup()
my_ttl.goto(15, -150)
my_ttl.pendown()

# draw circle
my_ttl.circle(5)
my_ttl.hideturtle()
```

PRIMM:

- **Predict** the type of house that the code will draw
- **Run** the code and see if it resembles your prediction.

Remember the DRY principle (**Don’t Repeat Yourself**)? Look at the code. How well does it go in relation to DRY?

Can you identify any repetition?

Look at the comments:

```python
import turtle

# set up screen
screen = 500
window = turtle.Screen()
window.setup(screen, screen)

# create turtle instance
my_ttl = turtle.Turtle()
my_ttl.shape("arrow")

##################################
## Using the tutrle command you ##
## have learnt, draw a house.   ##
##################################

# move pen
my_ttl.penup()
my_ttl.goto(-100, 0)
my_ttl.pendown()

# draw square
for index in range(4):
    my_ttl.forward(200)
    my_ttl.right(90)

# draw triangle
for index in range(3):
    my_ttl.forward(200)
    my_ttl.left(120)

# move pen
my_ttl.penup()
my_ttl.goto(-25, -200)
my_ttl.pendown()

# draw rectangle
for index in range(2):
    my_ttl.forward(50)
    my_ttl.left(90)
    my_ttl.forward(100)
    my_ttl.left(90)

# move pen
my_ttl.penup()
my_ttl.goto(-80, -100)
my_ttl.pendown()

# draw square
for index in range(4):
    my_ttl.forward(35)
    my_ttl.right(90)

# move pen
my_ttl.penup()
my_ttl.goto(45, -100)
my_ttl.pendown()

# draw square
for index in range(4):
    my_ttl.forward(35)
    my_ttl.right(90)

# move pen
my_ttl.penup()
my_ttl.goto(15, -150)
my_ttl.pendown()

# draw circle
my_ttl.circle(5)
my_ttl.hideturtle()
```

In summary we have two main types of repetition:

- moving the pen
- drawing the shape

When I wrote this code, I didn't type it straight out, I copied and pasted a lot of the code and changed the magic numbers' values. Copying and pasting is a clear indicator that you need to use a function. Why? Because functions are one of the main tools we can use to enforce the DRY Principle.

Creating functions

Let's look at how this works.

1. Take all the **move pen code** and consolidate that in one spot.
    - Below I have copied the first move pen action (`lines 17` to `20` in the previous code)
    - I have pasted them up to the top (`lines 4` to `7`)
    - I then turned them into a function
2. Replace the original code with a **call** to the function (`line 24`).

Adjust your code so that it looks the same as below:

```python
import turtle


def move_pen():
    my_ttl.penup()
    my_ttl.goto(-100, 0)
    my_ttl.pendown()


# set up screen
screen = 500
window = turtle.Screen()
window.setup(screen, screen)

# create turtle instance
my_ttl = turtle.Turtle()
my_ttl.shape("arrow")

##################################
## Using the tutrle command you ##
## have learnt, draw a house.   ##
##################################

move_pen()

# draw square
for index in range(4):
    my_ttl.forward(200)
    my_ttl.right(90)

# draw triangle
for index in range(3):
    my_ttl.forward(200)
    my_ttl.left(120)

# move pen
my_ttl.penup()
my_ttl.goto(-25, -200)
my_ttl.pendown()

# draw rectangle
for index in range(2):
    my_ttl.forward(50)
    my_ttl.left(90)
    my_ttl.forward(100)
    my_ttl.left(90)

# move pen
my_ttl.penup()
my_ttl.goto(-80, -100)
my_ttl.pendown()

# draw square
for index in range(4):
    my_ttl.forward(35)
    my_ttl.right(90)

# move pen
my_ttl.penup()
my_ttl.goto(45, -100)
my_ttl.pendown()

# draw square
for index in range(4):
    my_ttl.forward(35)
    my_ttl.right(90)

# move pen
my_ttl.penup()
my_ttl.goto(15, -150)
my_ttl.pendown()

# draw circle
my_ttl.circle(5)
my_ttl.hideturtle()
```

PRIMM:

- **Predict** what you think will happen
- **Run** the code and check you prediction

Now lets **investigate** the code by unpacking it:

- `Line 4`: `def move_pen():` create the function:
    - In programming we call this **defining** a function.
    - The program reads and bookmarks the code, but does not execute it.
    - `def` is the key word for defining a function.
    - `move_pen` is the name we are giving the function.
        - This name is how the program calls the function. It follows the same rules as variable names.
        - By using a descriptive name, we also remove the need for comments, as the code explains itself.
    - `()` is where we can put values. We'll deal with this soon.
    - `:` tells Python that an indented code block follows (the same as a `for` loop).
- `Lines 5` to `7` are indented:
    - This is the code that Python executes with a function call
    - The indentation rules are the same as the `for` loop
        - indentations can be many lines
        - multi-line indented code is called a **block**
        - indents should be four spaces
- `Line 24`: `move_pen()` calls the function:
    - At this point the program will go to `line 4` run the code in the function.
    - When Python finishes the function code, it returns to `line 24` and continues with the rest of the code.

Passing arguments

This works for our first pen movement. Since the coordinates are magic numbers, it won't work for the rest. I would have to create a function for each movement of the pen. This defeats the purpose of functions. What we need is a way to send the coordinates to the function when we call it. We can. Python uses **arguments** to pass values to a function.

Looking back at our `move_pen` function in the code, we need to get rid of those magic numbers.

```
4 def move_pen():
5     my_ttl.penup()
6     my_ttl.goto(-100, 0)
7     my_ttl.pendown()
```

What do the two magic numbers in `my_ttl.goto(-100,0)` represent? The `x` and the `y` of the coordinates. So let's replace them with variables.

```
4 def move_pen():
5     my_ttl.penup()
6     my_ttl.goto(x, y)
7     my_ttl.pendown()
```

But how do we assign values to `x` and `y`? We use **arguments.**

1. Change the function definition to `def move_pen(x, y):` so it will **accept** two values.
2. Change the function call in `line 24` to `move_pen(-100,0)` passing two values to the function.

Let's unpack that:

- `def move_pen(x, y):` says:
    - When you call the `move_pen` function, you need to provide two values.
    - First value is assigned to the variable `x`.
    - Second value is assigned to the variable `y`.
- `move_pen(-100,0)` says:
    - Call the `move_pen` function.
    - Use `-100` as the first value (the `x` value).
    - Use `0` as the second value (the `y` value).

Your code should now look like the code below:

```python
import turtle


def move_pen(x, y):
    my_ttl.penup()
    my_ttl.goto(x, y)
    my_ttl.pendown()


# set up screen
screen = 500
window = turtle.Screen()
window.setup(screen, screen)

# create turtle instance
my_ttl = turtle.Turtle()
my_ttl.shape("arrow")

##################################
## Using the tutrle command you ##
## have learnt, draw a house.   ##
##################################

move_pen(-100, 0)

# draw square
for index in range(4):
    my_ttl.forward(200)
    my_ttl.right(90)

# draw triangle
for index in range(3):
    my_ttl.forward(200)
    my_ttl.left(120)

# move pen
my_ttl.penup()
my_ttl.goto(-25, -200)
my_ttl.pendown()

# draw rectangle
for index in range(2):
    my_ttl.forward(50)
    my_ttl.left(90)
    my_ttl.forward(100)
    my_ttl.left(90)

# move pen
my_ttl.penup()
my_ttl.goto(-80, -100)
my_ttl.pendown()

# draw square
for index in range(4):
    my_ttl.forward(35)
    my_ttl.right(90)

# move pen
my_ttl.penup()
my_ttl.goto(45, -100)
my_ttl.pendown()

# draw square
for index in range(4):
    my_ttl.forward(35)
    my_ttl.right(90)

# move pen
my_ttl.penup()
my_ttl.goto(15, -150)
my_ttl.pendown()

# draw circle
my_ttl.circle(5)
my_ttl.hideturtle()
```

PRIMM

- **Predict** what this code will now do.
- **Run** the code to check if your prediction was correct.

- **Investigate** the code by using the debugger and stepping your way through the program.

> **Arguments vs Parameters**
>
> In programming discussions the terms **arguments** and **parameters** are often swapped around. It is safe to use either term, but they do have distinct meanings:
>
> - arguments are the values the main program passes to a function
> - parameters are the variables named in the function definition

---

Go through the code and replace the remaining `# move pen` blocks with a `move_pen()` call.

Your code should now look like this:

```python
import turtle


def move_pen(x, y):
    my_ttl.penup()
    my_ttl.goto(x, y)
    my_ttl.pendown()


# set up screen
screen = 500
window = turtle.Screen()
window.setup(screen, screen)

# create turtle instance
my_ttl = turtle.Turtle()
my_ttl.shape("arrow")

##################################
## Using the tutrle command you ##
## have learnt, draw a house.   ##
##################################

move_pen(-100, 0)

# draw square
for index in range(4):
    my_ttl.forward(200)
    my_ttl.right(90)

# draw triangle
for index in range(3):
    my_ttl.forward(200)
    my_ttl.left(120)

move_pen(-25, -200)

# draw rectangle
for index in range(2):
    my_ttl.forward(50)
    my_ttl.left(90)
    my_ttl.forward(100)
    my_ttl.left(90)

move_pen(-80, -100)

# draw square
for index in range(4):
    my_ttl.forward(35)
    my_ttl.right(90)

move_pen(45, -100)

# draw square
for index in range(4):
    my_ttl.forward(35)
    my_ttl.right(90)

move_pen(15, -150)

# draw circle
my_ttl.circle(5)
my_ttl.hideturtle()
```

Run the code to make sure the house is still drawn.

Notice that our line count is down from the original `71` to `63`.

> **Testing tips**
>
> - It is good to frequently test your code.
> - Each time you change your code, test it.
> - Try not to make too many changes between testing, it makes it harder to identify your errors.
> - If function passes its test, you don't have to testing it again, unless your change the function.
> - If your functions passed their tests, then you know the error is elsewhere in the code.

Functions in Flowcharts

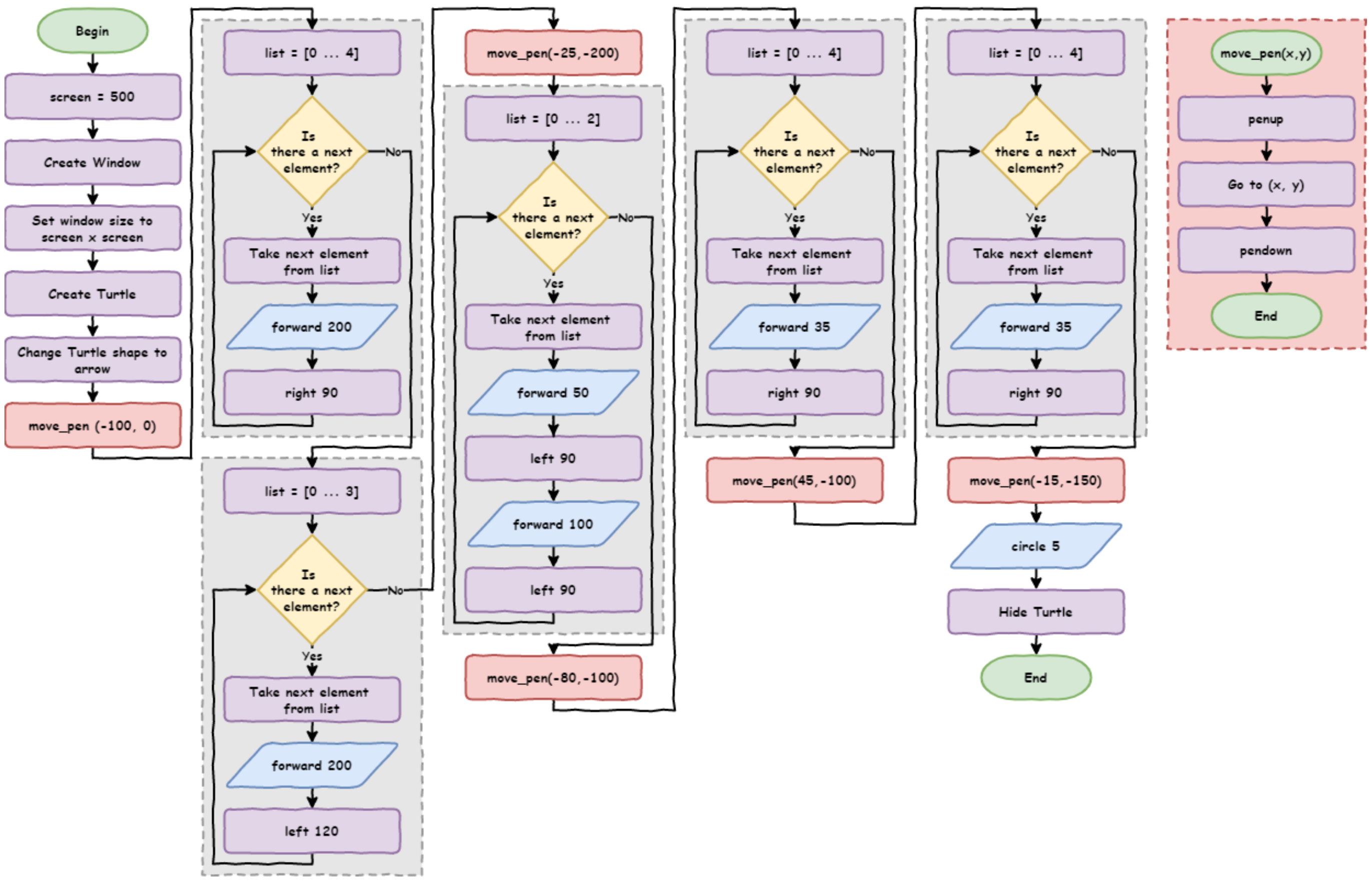
Flowcharts don't represent whole programs, they represent algorithms.

> **What are algorithms?**
>
> Algorithms are a set of rules to follow to solve a problem. A cake recipe is an algorithm to bake a cake. You follow an algorithm to perform long division in maths. In computers, you code instructions are the algorithms.

When a program consists of smaller algorithms (eg. functions), create a flowchart for each algorithm. Then show where algorithms call other algorithms.

We show the name of the function in the terminator symbol. **Main** is the name of the starting algorithm.

Here is the flowchart of the code with the `move_pen` function. The function calls use the procedure symbol (I have coloured them red to make them stand out).



Shape functions

When we first looked for repetition, we also identified the drawing shapes repetition. Lets make a function to draw squares.

From the current code:

- copy one of the `# draw square` blocks to the top of the code
- change it into a function that draws a square called `draw_square`
- the function will need to accept a value for the `length` of the square's side
- then replace all the `# draw square` blocks with an appropriate `draw_square` call

> **Where should I place functions?**
>
> Function definitions are place at the top of the code, right after the `import` statements.
>
> This has two reasons:
>
> - If the function is not defined before you call it, your code will generate a `NameError`.
> - Placing all your functions at the start improves makes them easier to find them. This improves your code's maintainability

Once you have made `draw_square` function changes, you code should look like:

```python
import turtle


def move_pen(x, y):
    my_ttl.penup()
    my_ttl.goto(x, y)
    my_ttl.pendown()


def draw_square(length):
    for index in range(4):
        my_ttl.forward(length)
        my_ttl.right(90)


# set up screen
screen = 500
window = turtle.Screen()
window.setup(screen, screen)

# create turtle instance
my_ttl = turtle.Turtle()
my_ttl.shape("arrow")

####################################
## Using the tutrle command you ##
## have learnt, draw a house.   ##
####################################

move_pen(-100, 0)
draw_square(200)

# draw triangle
for index in range(3):
    my_ttl.forward(200)
    my_ttl.left(120)

move_pen(-25, -200)

# draw rectangle
for index in range(2):
    my_ttl.forward(50)
    my_ttl.left(90)
    my_ttl.forward(100)
    my_ttl.left(90)

move_pen(-80, -100)
draw_square(35)
move_pen(45, -100)
draw_square(35)
move_pen(15, -150)

# draw circle
my_ttl.circle(5)
my_ttl.hideturtle()
```

We are now down to 55 lines of code.

There is no more repetition in the main code, but there is still three code blocks remaining. Notice how the rest of the code is easier to read? Therefore, we will transform the `# draw triangle`, `# draw rectangle` and `# draw circle` code blocks into functions.

This will provide two benefits:

- It will improve maintainability by making the code more readable.
- If we want to extend the drawing we can easily add more rectangles, triangle and circles.

See if you can change all three blocks into functions. Remember to test each function when you create it.

When you finish your code should look like this:

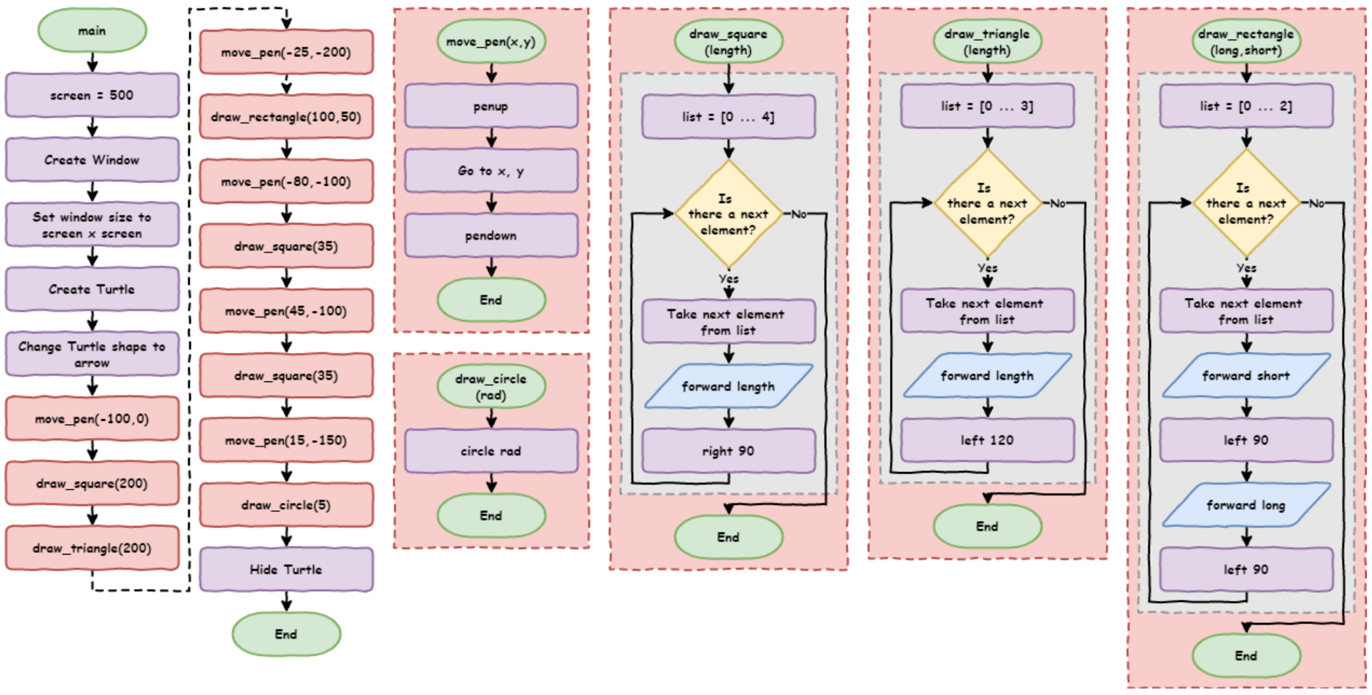
```python
import turtle


def move_pen(x, y):
    my_ttl.penup()
    my_ttl.goto(x, y)
    my_ttl.pendown()


def draw_square(length):
    for index in range(4):
        my_ttl.forward(length)
        my_ttl.right(90)


def draw_triangle(length):
    for index in range(3):
        my_ttl.forward(length)
        my_ttl.left(120)


def draw_rectangle(long, short):
    for index in range(2):
        my_ttl.forward(short)
        my_ttl.left(90)
        my_ttl.forward(long)
        my_ttl.left(90)


def draw_circle(rad):
    my_ttl.circle(rad)


# set up screen
screen = 500
window = turtle.Screen()
window.setup(screen, screen)

# create turtle instance
my_ttl = turtle.Turtle()
my_ttl.shape("arrow")

##################################
## Using the tutrle command you ##
## have learnt, draw a house.   ##
##################################

move_pen(-100, 0)
draw_square(200)
draw_triangle(200)
move_pen(-25, -200)
draw_rectangle(100, 50)
move_pen(-80, -100)
draw_square(35)
move_pen(45, -100)
draw_square(35)
move_pen(15, -150)
draw_circle(5)
my_ttl.hideturtle()
```

That's our final code:

- Down from `71` lines to `59` lines.
- Easier to read.

- Easier to test and troubleshoot errors.

Maybe the easiest way to see the improvement in our code is to look at the flowchart.



Part 1 Exercises

In this course, the exercises are the **make** component of the PRIMM model. So work through the following exercises and make your own code.

Exercise 1

Download `lesson_4_ex_1.py` file and save it to your lesson folder. Below is its code.

```python
import turtle

# set up screen
screen = 500
window = turtle.Screen()
window.setup(screen, screen)

# create turtle instance
my_ttl = turtle.Turtle()
my_ttl.shape("arrow")

##############################################
## Convert the code below using functions ##
##############################################

# move pen
my_ttl.penup()
my_ttl.goto(0, -200)
my_ttl.pendown()

# draw head
my_ttl.color("black", "yellow")
my_ttl.begin_fill()
my_ttl.circle(200)
my_ttl.end_fill()

# move pen
my_ttl.penup()
my_ttl.goto(-75, 0)
my_ttl.pendown()

# draw eye
my_ttl.color("black", "black")
my_ttl.begin_fill()
my_ttl.circle(50)
my_ttl.end_fill()

# move pen
my_ttl.penup()
my_ttl.goto(75, 0)
my_ttl.pendown()

# draw eye
my_ttl.color("black", "black")
my_ttl.begin_fill()
my_ttl.circle(50)
my_ttl.end_fill()

# move pen
my_ttl.penup()
my_ttl.goto(-100, -75)
my_ttl.pendown()

# draw mouth
my_ttl.color("black", "black")
my_ttl.begin_fill()
for index in range(2):
    my_ttl.forward(200)
    my_ttl.right(90)
    my_ttl.forward(25)
    my_ttl.right(90)
my_ttl.end_fill()

my_ttl.hideturtle()
```

Follow the instructions in the comments and adapt the code so it uses functions.

Exercise 2

Download `lesson_4_ex_2.py` file and save it to your lesson folder. Below is its code.

```
import turtle

# set up screen
screen = 500
window = turtle.Screen()
window.setup(screen, screen)

# create turtle instance
my_ttl = turtle.Turtle()
my_ttl.shape("arrow")

############################################
## Use you knowledge of Python and Turtle ##
## to draw a car. Use functions to ensure ##
## that you Do not Repeat Yourself.       ##
############################################
```

Follow the instructions in the comments and write a program that draws a car.

Part 2: User Input

Video link

Introduction

Download the `lesson_4_pt_2.py` file and save it to your lesson folder.

```
 1 import turtle
 2
 3
 4 def draw_poly(length, sides):
 5     for index in range(sides):
 6         my_ttl.forward(length)
 7         my_ttl.right(360 / sides)
 8
 9
10 # setup window
11 screen = 500
12 window = turtle.Screen()
13 window.setup(screen, screen)
14
15 # create instance of turtle
16 my_ttl = turtle.Turtle()
17 my_ttl.shape("turtle")
18
19 sides = 9
20 length = 100
21
22 draw_poly(length, sides)
```

PRIMM

- **Predict** what you think will happen.
- **Run** the code and see how close your prediction is.
- **Modify** the code so the shape fits within the window.

When we run the code, part of the shape is off the screen. This is not a big problem. Change the length from `100` to `80`. This is something quite simple for you because you have learnt how to code. What about people who haven't?

How do we make our programs interactive by getting input from users who cannot code?

Making your program interactive

The simplest way to make your program interactive is to use the `input` command. It will use the **Shell** to ask the user for their input.

To do this change:

- `line 19` to `sides = input("How many sides?> ")`

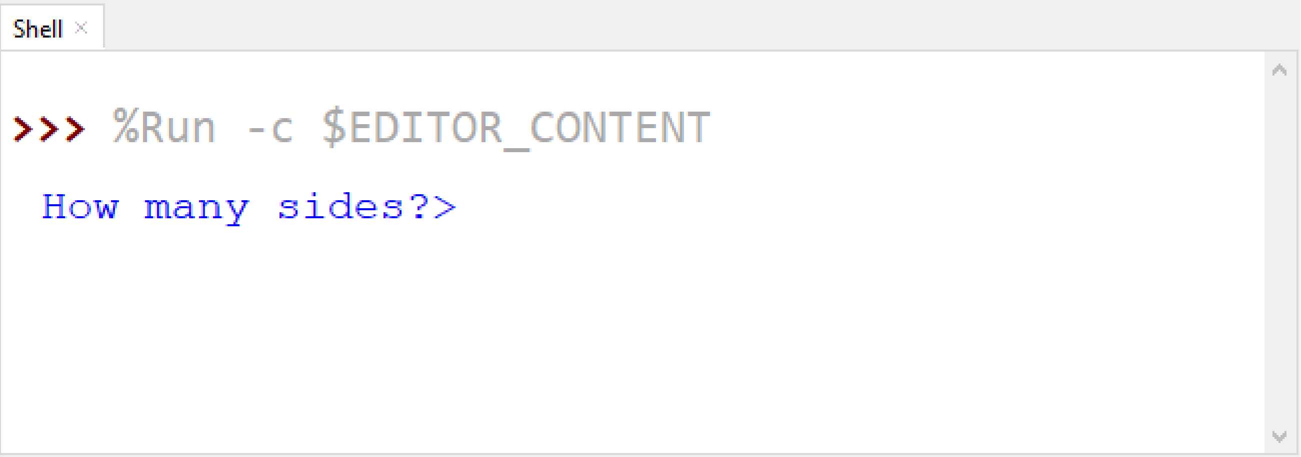
- `line 20` to `length = input("How long are the sides?> ")`

Your code should look like the following:

```
 1 import turtle
 2 
 3 
 4 def draw_poly(length, sides):
 5     for index in range(sides):
 6         my_ttl.forward(length)
 7         my_ttl.right(360 / sides)
 8 
 9 
10 # setup window
11 screen = 500
12 window = turtle.Screen()
13 window.setup(screen, screen)
14 
15 # create instance of turtle
16 my_ttl = turtle.Turtle()
17 my_ttl.shape("turtle")
18 
19 sides = input("How many sides?> ")
20 length = input("How long are the sides?> ")
21 
22 draw_poly(length, sides)
```

PRIMM

- **Predict** what you think will happen.
- **Run** the code. Did it do what you thought?
  - Did you predict:
    - a **prompt** appearing in the **Shell** like the image below?
    - the program raising an error.



```
>>> %Run -c $EDITOR_CONTENT
 How many sides?>
```

```
Traceback (most recent call last):
  File "<string>", line 22, in <module>
  File "<string>", line 5, in draw_poly
TypeError: 'str' object cannot be interpreted as an integer
```

- Let's **investigate** by:
  - unpacking the code we changed
  - explaining the error

Unpacking `line 19` (note `line 20` is virtually the same):

- `input`: is the keyword that tells Python to wait for an input from the user from the **Shell**.
- `("How many sides?> ")` tells Python what **prompt** to write to the **Shell** before it waits for a response.
- `sides =` takes whatever the user enters and assigns it to the variable `sides`

Now for the error. This is a `TypeError` and to understand it we need to learn about **data types**.

Data types

Variables in Python can hold different types of data. The four types of data we will use are:

- **integer numbers** (`int`)

    - stores whole numbers
    - identified by a whole number
- **floating point numbers** (`float`)
    - stores numbers that have a decimal points
    - identified by having a decimal point with at least one number after it. For example, `1` is and integer, `1.0` is a float
- **strings** (`str`)
    - stores characters like letters, numbers and special characters
    - start and end with `"` or `'` (just make sure they are the same the at beginning or end)
    - numbers can be a string. For example, a phone number like `0432 789 367` is a string not and integer or float. It contains spaces and you would never do a calculation with it.
- **Booleans** (`bool`)
    - store either `True` or `False`

Using data types helps Python work out what kind of operations it can do with the variable. For example, it wouldn't make much sense to divide a string. Python also has special operations called **methods**. Each data type has its own collection of methods. You will learn more about data types throughout your programming journey.

Now, lets look at the error again:

```
1 Traceback (most recent call last):
2   File "<string>", line 22, in <module>
3   File "<string>", line 5, in draw_poly
4 TypeError: 'str' object cannot be interpreted as an integer
```

Breaking the error down:

- Error `line 4`: `TypeError: 'str' object cannot be interpreted as an integer`:
    - This tells us that this involves two data types (string and integer).
    - It says we are trying to use a string when Python is expecting an integer.
- `Traceback`:
    - When looking at a `Traceback` always check the last line first
    - Error `line 3` tells us that the error occurred at `line 5` in the code: `for index in range(sides):`
        - Here we are trying to use the values in `sides` in a `range` function, but Python thinks it is a string.
        - let's look at where we got the value for `sides`
    - `Line 19`: `sides = input("How many sides?> ")`
        - We took the value the user entered and assigned it to `sides`.
        - I entered `3` which is an integer.
        - Why does Python think it's a string?

When Python accepts a value using the `input` function, it is always accepted it as a string. This is because strings can contain all characters.

How do we fix this? Luckily, we can convert a variable's data type.

Converting data types

There is a function to convert any data type into each other data type (other than Boolean).

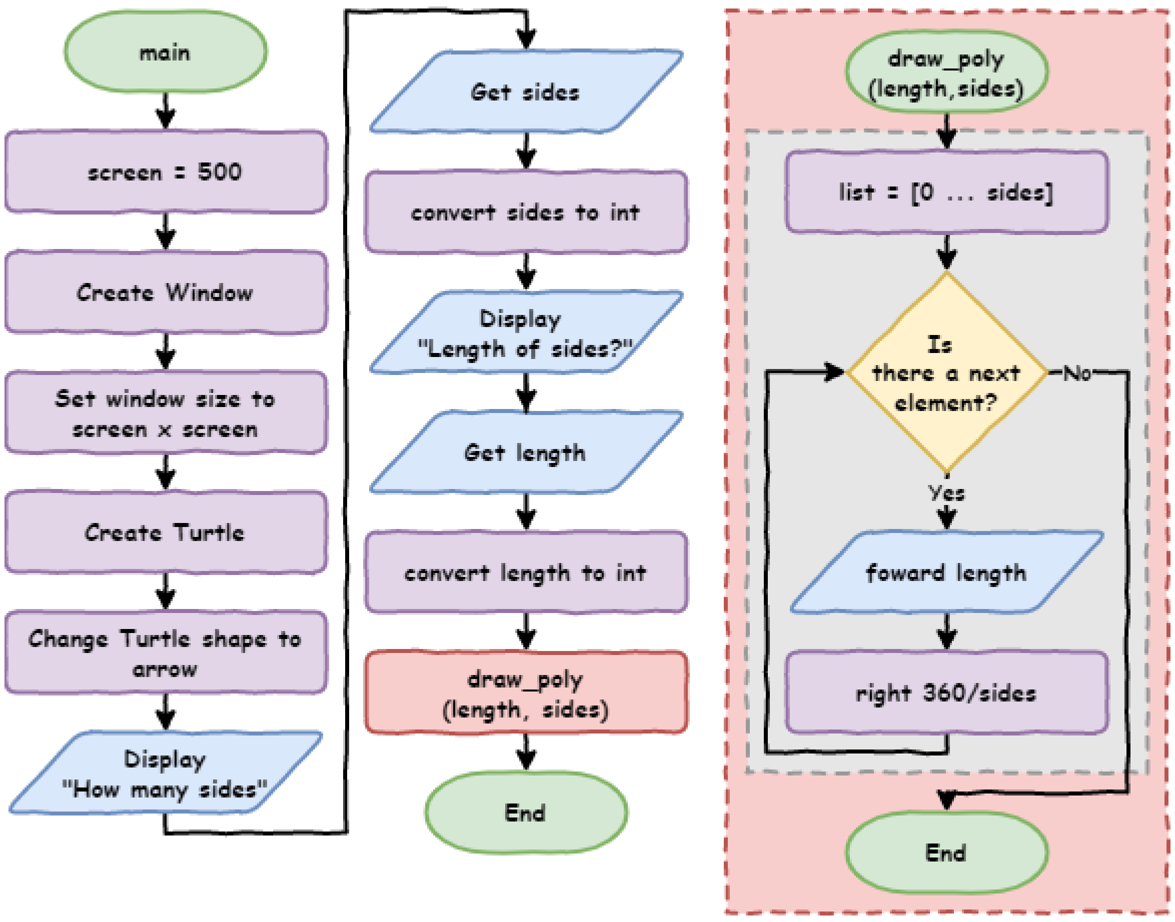
If we had a variable called `var`:

- convert `var` → string, use `str(var)`
- convert `var` → integer, use `int(var)`
- convert `var` → a float, use `float(var)`

There is a great deal more to this, but at the moment this is all you need to know.

Let's change our code. Take the strings returned by the `input` function and convert them into integers.

Here is the finished code as a flowchart. Note that we use the same symbol for input as we do for output, with different wording.



Below is the finished code, with the changes on `lines 19` and `20`.

```
 1 import turtle
 2 
 3 
 4 def draw_poly(length, sides):
 5     for index in range(sides):
 6         my_ttl.forward(length)
 7         my_ttl.right(360 / sides)
 8 
 9 
10 # setup window
11 screen = 500
12 window = turtle.Screen()
13 window.setup(screen, screen)
14 
15 # create instance of turtle
16 my_ttl = turtle.Turtle()
17 my_ttl.shape("turtle")
18 
19 sides = int(input("How many sides?> "))
20 length = int(input("Length of sides?> "))
21 
22 draw_poly(length, sides)
```

PRIMM

- **Predict** what you think will happen
- **Run** you code and see if your predictions were correct
- **Investigate** by trying to enter different values for sides and length:
  - draw different shapes
  - what are the correct values to make your turtle draw a circle?
  - what happens when you enter a float or a string?
- **Modify** your code to use different prompts

Part 2 Exercise

In this course, the exercises are the **make** component of the PRIMM model. Work through the following exercises and make your own code.

Exercise 3

Download `lesson_4_ex_3.py` file and save it to your lesson folder. Below is its code.

```
################################################
## write a program that askes the user for a ##
## number and then counts up to that number. ##
################################################
```

Follow the instructions in the comments and use your Python knowledge to create a count up app. Remember to apply the DRY principle