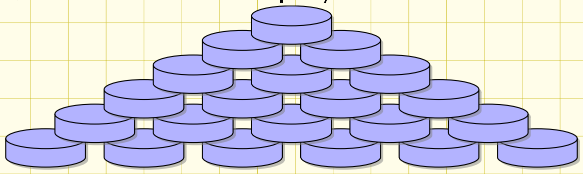
# 3 Graphical Take Away

## Introduction

In the previous worksheet, we created a text based version of the Take Away game:

**Take Away Game**

Game consists of a pile of coins with two players taking turns to play. At each turn the player decides to take 1, 2 or 3 coins. The player who take the last coin wins.



We would like to now create a graphical version of this game but there are a few concpets we need to cover first. In particular:
The competed text-based version of the game is included in the next page. In order to create a graphical version we will need to cover some additional concepts.

① **Turtle graphics**

Turtle graphics is a python module that allow us to create drawings It was designed to help teach programming concepts so is similar to Scratch programming.

② **Event programming**

We will cover a programming style that is more suited to graphical programs.

③ **Designing our Game**

OK, now we return to our game and design our graphic version.

CoderDojo, Tramore, Waterford. (`kmurphy@wit.ie`)

**Completed `TakeAway.py` from Chapter 2**

```python
# initialise game

import random
coins = random.randint(15, 21)
isComputerMove = False
skill = 0.5
print ("Instructions go here ...\n\n")

def getComputerMove(coins):
    if random.random()<skill:   # supersmart (perfect player mode)
        remainder = coins % 4
        if remainder==0:
            return random.randint(1,min(2,coins))
        else:
            return remainder
    else:                       # stupid (random mode)
        return random.randint(1,min(3,coins))

def getHumanMove(coins):
    # human move -- need to check input because humans cheat !
    while True:
        move = int(input("How many coins do you want to take? "))
        if move>coins:
            print ("Only %s coins remaining. Try again" % coins)
        elif move not in [1,2,3]:
            print ("Expected a move of '1', '2' or '3'.")
        else:
            return move

# play the game

while coins>0:

    print("The pile contains %s coins." % coins)

    if isComputerMove:
        move = getComputerMove(coins)
        print ("I'm taking %s coins" % move)
    else:
        move = getHumanMove(coins)
        print ("You're taking %s coins" % move)

    coins = coins - move
    isComputerMove = not isComputerMove

# output result of game

if isComputerMove:
    print ("\nYou moved last. you win")
else:
    print ("\nI moved last. you lose")
```

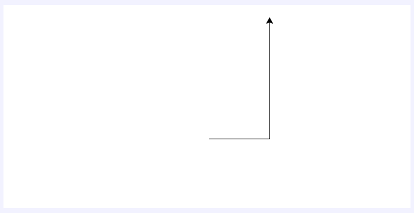
## 1 Turtle graphics

Turtle graphics is centred around the idea of little turtles that we can control and as they move they trace out a path. The idea is simple, but it is surprising simple to create complex pictures. It was used to develop programming languages such as LOGO which ultimately led to Scratch. To get us started let's write a few small programs demonstrating its capabilities …

Create a new file with the following code and save as `first.py`.

first.py

```python
import turtle

bob = turtle.Turtle()
bob.forward(100)
bob.left(90)
bob.forward(200)
```



- In line 1, we import the turtle module. This gives us access to all of the commands defined in this module. Remember we also used this command to import the `random` module previously.
- Next we create a new turtle (pen) and we call it `bob`. We could use any name here. I like bob because it is nice and short, so saves on the typing.
- In the remaining lines we tell `bob` what to do and `Bob` happily (i hope) does it. `bob` understands a large number of commands, including

### Movement commands

- `forward` — move forward a given distance.
- `backward` — move backward a given distance.
- `left` — turn left a given angle (measured in degrees)
- `right` — turn right a given angle (measured in degrees)
- `setheading` — turn to face in a given direction.
- `setposition` — move to a given position (horizontal and vertical.
  * `setx` — move to a given horizontal position.
  * `sety` — move to a given vertical position.
- `home` — move back to starting position and direction

### Drawing commands

- `dot` — draw a dot of given radius in current position
- `circle` — draw a circle of given radius in current position
- `penup` — "lifts pen" so subsequent movement will not be drawn.
- `pendown` — "lowers pen" so subsequent movement will be drawn.
- `color` — set the drawing colour. Use a known colour such as `"red"` or by building your own by mixing the primary colours (red, green and blue).

- `stamp` — stamp a copy of the turtle shape at the current turtle position.
- `write` — write a message to the screen.

**Control commands**

- `speed` — set how fast the turtle moves: `"slowest"`, `"slow"`, `"normal"`, `"fast"`, and `"fastest"`.
- `undo` —undo (repeatedly) the last turtle action(s).
- `clearstamp` — delete a perviously placed stamp.

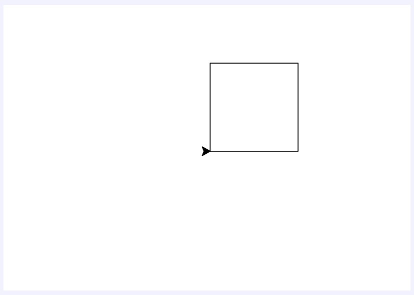
## 1.1 Drawing Shapes

Just listing commands for the turtle to do is great and all that but the magic really happens when we use repetition (**for** and **while** loops) and decisions (**if**–**elif**–**else** statement) to build more complicated diagrams with only little effort from us, the programmer.

Create a new file with the following code and save as `Square_Basic.py`.

Square_Basic.py

```python
import turtle

bob = turtle.Turtle()
bob.forward(100)
bob.left(90)
bob.forward(100)
bob.left(90)
bob.forward(100)
bob.left(90)
bob.forward(100)
```

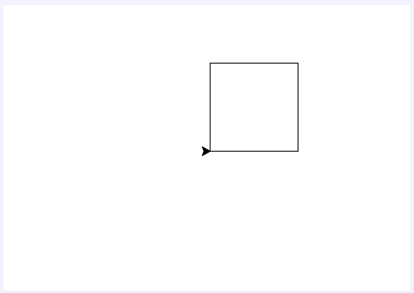


OK, the generated diagram was (hopefully) not a surprise — a square is just four lines of equal length joined by 90° angles. But our code is getting long and, worse, it is getting repetitive. Let's fix that with a **for** loop.

Create a new file with the following code and save as `Square_Loop.py`.

Square_Loop.py

```python
import turtle

bob = turtle.Turtle()

for k in range(4):
    bob.forward(100)
    bob.left(90)
```



OK, the code is not that much shorter. But it no longer repetitive! it is a big deal when we want to write good code.

Create a new file, save it as `Triangle.py` which draws a triangle.

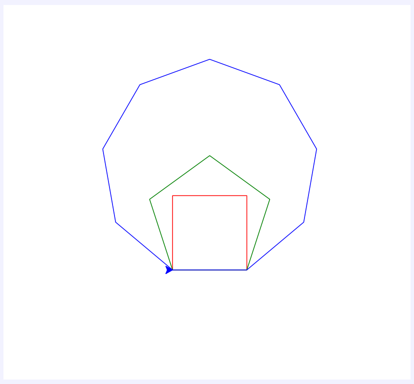
Create a new file, save it as `Pentagon.py` which draws a pentagon.

The steps for drawing the triangle, square and pentagon are similar — so again to avoid repetitive code — we could identify the pattern and write a function for this …

Create a new file with the following code and save as `Polygon.py`.

polygon.py

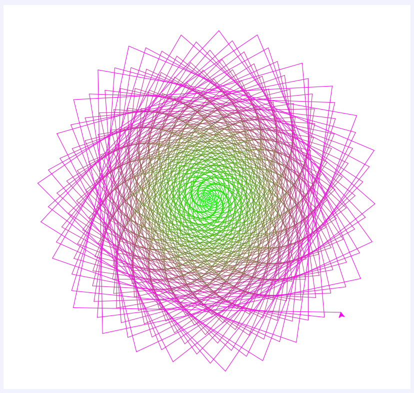
```
import turtle

bob = turtle.Turtle()

def drawPolygon(n):
    angle = 360/n
    for k in range(n):
        bob.forward(100)
        bob.left(angle)


bob.color("red")
drawPolygon(4)
bob.color("green")
drawPolygon(5)
bob.color("blue")
drawPolygon(9)
```



We will talk in class and if you like we play with turtle graphics a while to see what we can generate. for example, I really like the following

spiral.py

```
import turtle

bob = turtle.Turtle()

size = 500
bob.speed("fastest")

for k in range(size):
    bob.forward(k)
    bob.left(101)

    # mix a new color
    f = k/size    # number between 0 and 1
    myColor = (f, 1-f, f*f)

    bob.color(myColor)
```



## 2 Event programming

Today we are going to write our programmes using a different style or **programming paradigm** called **event programming**

**Programming Paradigms**

**Programming paradigms** is not about using different programming languages but about different ways of structuring our code. Picking the right programming paradigms can have a huge impact on the effort needed to implement a task. So far we have used **imperative programming** but there are many others …

- **Imperative programming** is a programming style where we write a sequence of commands in a particular order that the computer must execute in that order. Essentially, we are telling the computer "do this … do this … do this … do this … do this …"

  *Imperative programming is a very common style and because we decide on the order commands are executed it can be easier to debug.*

- **Event programming** is a programming style where we write separate blocks of code, usually functions, and we tell the computer to listen/watch for events to happen. An event could be a key pressed, or a message received across the internet, etc. With each event we link it to one of our blocks of code.

  Whenever a listed event occurs the computer tries to interrupt what it is doing and run the linked code immediately, and then resume its previous task.

  *Because the programmer does not control what event occur, the programmer does not know the order their commands will be executed. This can make writing/debugging event programming code more difficult. However, event programming is ideal for programs that have a graphical interface with the user..*

- **Object oriented programming** is a programming style where we group data and code into an **object**. This is useful because often only certain tasks make sense to be applied to certain data.

  *This paradigm is popular for developing large programs or for programs that will be used a part of developing other programs.*

- and many, many more …

  Some programming languages are designed around one programming paradigm, but python support many. You, the programmer, can decide which paradigm to use, and often mix style to suit the task at hand.

Create a new file with the following code and save as `Square_Event.py`.

Square_Event.py

```
 1  import turtle
 2
 3  window = turtle.Screen()
 4
 5  bob = turtle.Turtle()
 6
 7  def drawSquare():
 8      for k in range(4):
 9          bob.forward(100)
10          bob.left(90)
11
12  window.onkey(drawSquare, "s")
13
14  window.listen()
15  window.mainloop()
```

What happened? Well, unless you pressed "S" nothing happened. Let's look at the code …

- Line 3 is new. Here we store the window that our little turtle runs around in. We will use this later in the code to 'listen' out for events — things that happen, such a key press or mouse clicks.
- In lines 7 to 10 we just define a function to draw a square. We have seen this before, but make sure you are happy with it. If not then please ask!
- In line 12 we tell our window to listen out for the key "S" being pressed. And we tell it to run the function `drawSquare` when the windows 'hears' the key "S".
- Lines 14 tells our window to "start listening now" and line 15 tells the window to "keep listening for ever and ever".

## 2.1 I'm Busy! Stop interrupting Me!

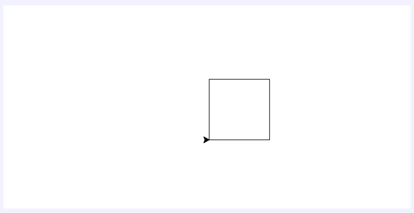
Rerun your `Square_Event.py` program and press "S" once. Everything should work as expected.

Rerun `Square_Event.py` a second time and now press "S" quickly a few times. What happens?

Have a look at the previous page on programming styles to get an idea of what has gone wrong.

Square_Busy.py

```
 1  import turtle
 2
 3  window = turtle.Screen()
 4  bob = turtle.Turtle()
 5  bob.busy = False
 6
 7  def drawSquare():
 8      if bob.busy: return
 9      bob.busy = True
10
11      for k in range(4):
12          bob.forward(100)
13          bob.left(90)
14      bob.busy = False
15
16  window.onkey(drawSquare, "s")
17
18  window.listen()
19  window.mainloop()
```



So what are we doing here ?

- In line 5, we are adding a property/flag to `bob` called `busy`. This will store `True` when `bob` should not be interrupted, and `False` when `bob` can be interrupted to do another task.
- Then every task that we plan for `bob` should have the same structure and `drawSquare`. In that we
  - Check if `bob` is `busy`, and if yes the return without doing anything.
  - Set property/flag `busy` to `True` so `bob` can't be interrupted.
  - Do the task at hand — here the task is drawing a square.
  - Once task is finished we need to set property/flag `busy` to `False` so `bob` can be interrupted again.

Modify the above code so that it prints out a message "I'm busy" whenever `bob` is interrupted during drawing a square.

## 3 TakeAway game —- remember me ?

OK, we are 8 pages in and we have not started on our TakeAway game yet. I'm too lazy to change the chapter title so we will leave the game until next week.