Using Text Files for Animation

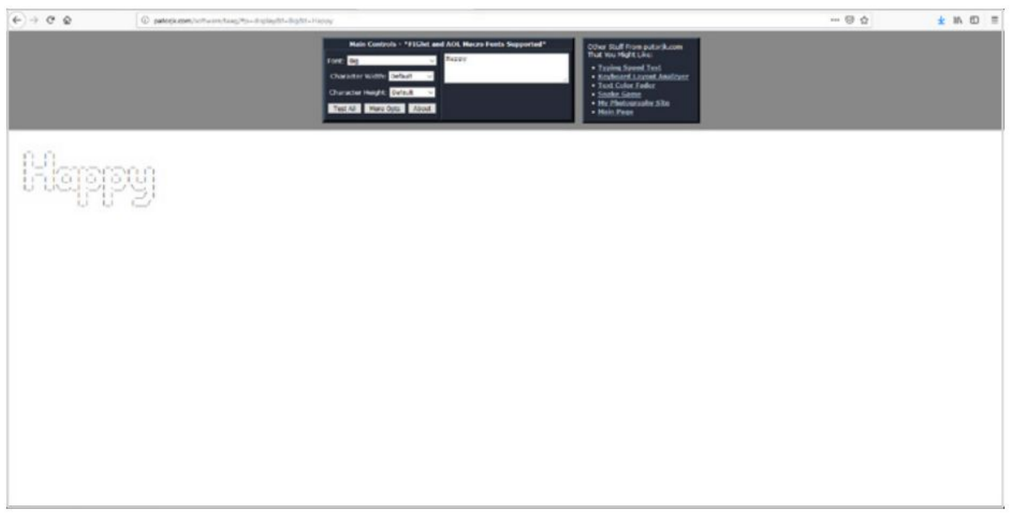
Animation in Python can be handled with the likes of the Tkinter and Pygame modules, however, there's more than one way to achieve a decent end result. Using some clever, text file reading code, we can create command-line animations.

ASCII ANIMATION

Let's assume you wanted to create an animated ASCII Happy Birthday Python script, with the words Happy and Birthday alternating in appearance. Here's how it's done.

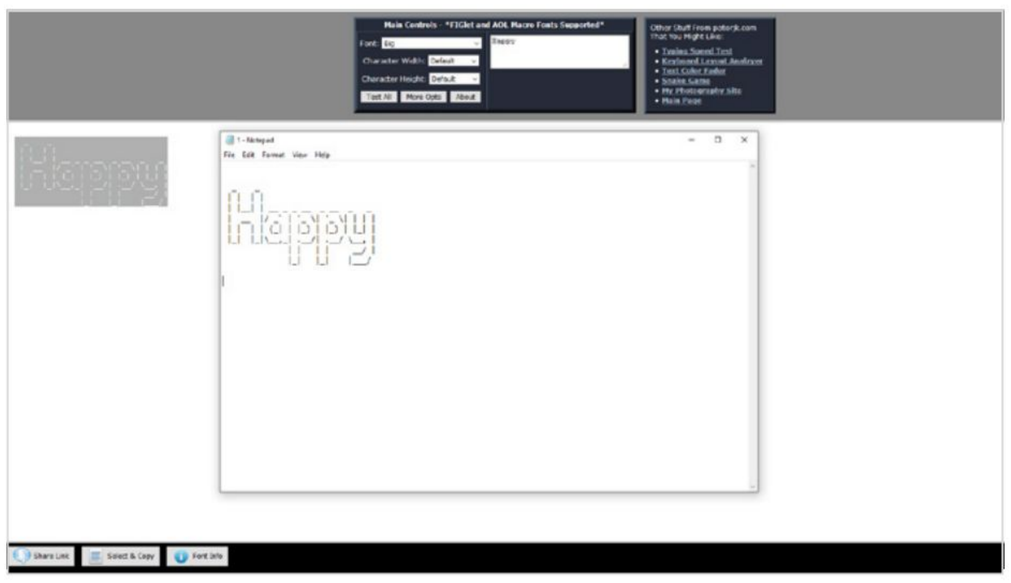
STEP 1

First we need to create some ASCII-like text, head over to **http://patorjk.com/software/taag**. This is an online Text to ASCII generator, created by Patrick Gillespie. Start by entering **Happy** into the text box, the result will be displayed in the main window. You can change the font with the drop-down menu, to the side of the text box; we've opted for **Big**.



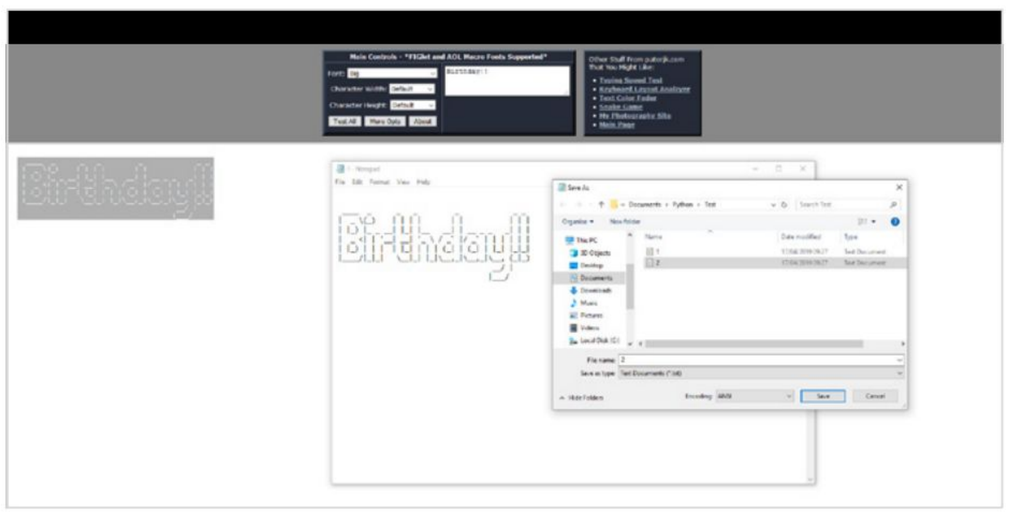
STEP 2

Now create a folder in your Python code directory on your computer (call it Test, for now), and open either Notepad for a Windows 10 computer, or, if you're using Linux, then the currently installed text editor. Click on the **Select & Copy** button at the bottom of the ASCII Generator webpage, and paste the contents into the text editor.



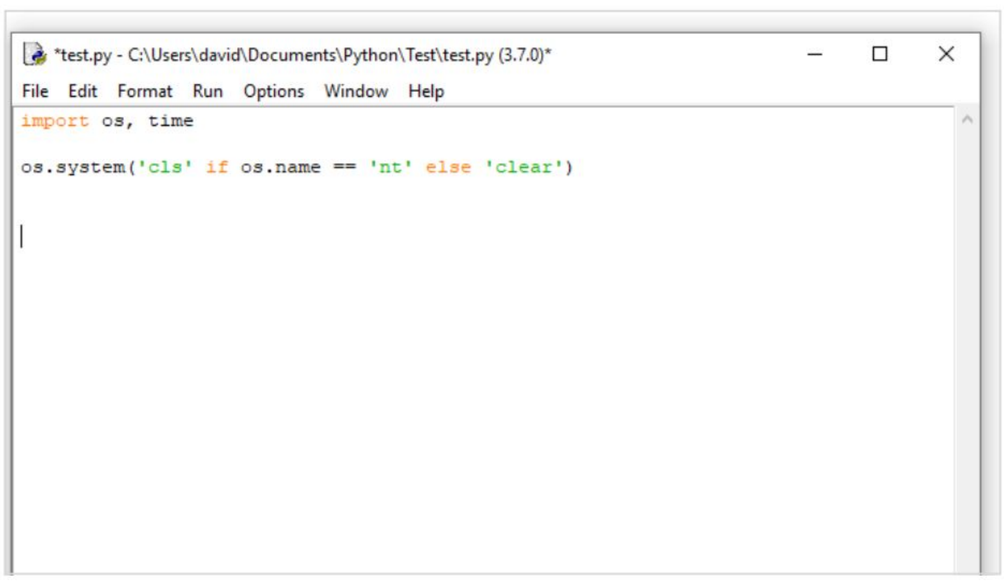
STEP 3

Save the text file as **1.txt** (you can call it what you like, but now for ease of use 1.txt will suffice). Save the file in the newly created Test folder. When it's saved, do exactly the same for the word Birthday. You can select a new font from the ASCII Generator, or add extra characters and when you're ready, save the file as **2.txt**.



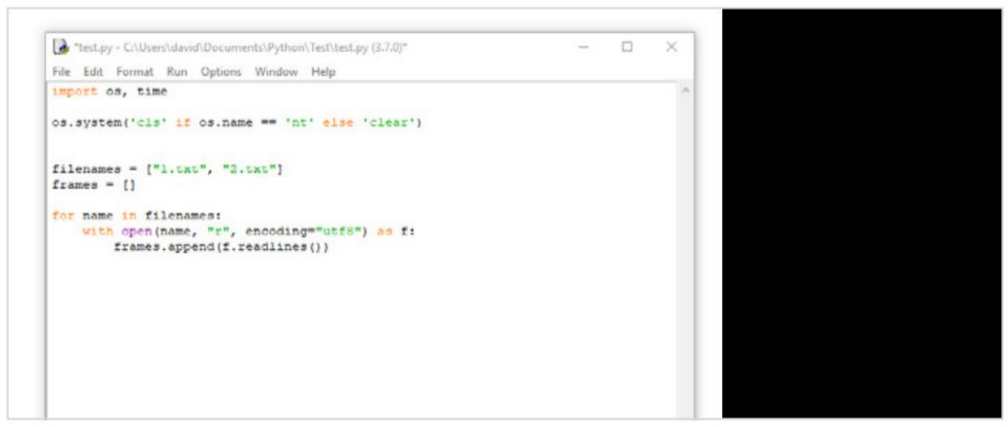
STEP 4

Open up Python and create a **New File**. We're going to need to import the OS and Time modules for this example, followed by a line to clear the screen of any content. If you're using Windows, then you'll use the CLS command, whereas it's Clear for Linux. We can create a simple if/else statement to handle the command.



STEP 5 Next we need to create a list of the names of the text files we want to open, and then we need to open them for display in the Terminal.

```
filenames = [“1.txt”, “2.txt”]
frames = []

for name in filenames:
    with open(name, “r”, encoding=”utf8”) as f:
        frames.append(f.readlines())
```



STEP 6 We’ve used the UTF8 standard when opening the text files, as ASCII art as text, within a text file, often requires you to save the file as UTF compliant – due to the characters used. Now we can add a loop to display the files as 1.txt, then 2.txt, creating the illusion of animation while clearing the screen after each file is displayed.

test1.py - C:/Users/david/Documents/Python/Test/test1.py (3.7.0)

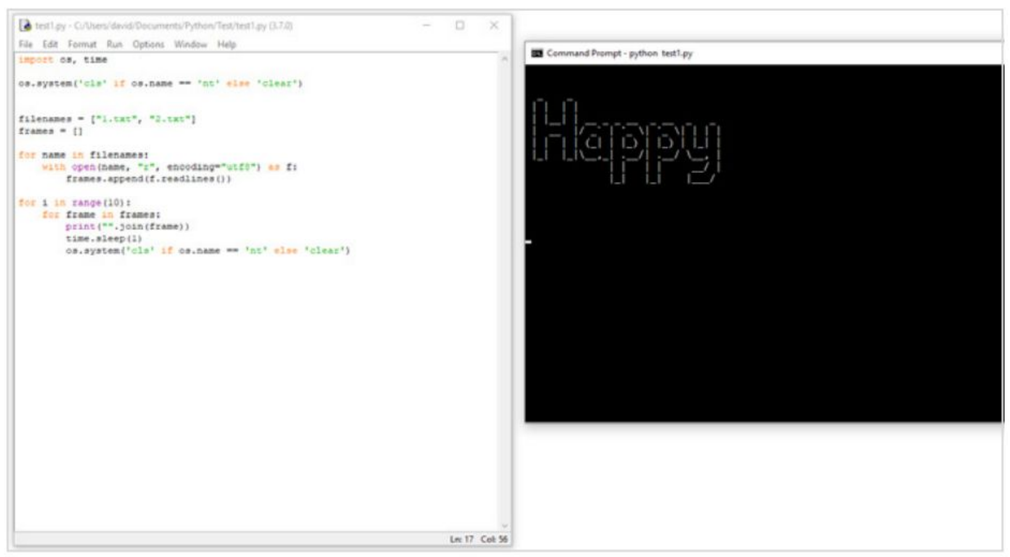
File Edit Format Run Options Window Help

```
import os, time

os.system('cls' if os.name == 'nt' else 'clear')


filenames = ["1.txt", "2.txt"]
frames = []

for name in filenames:
    with open(name, "r", encoding="utf8") as f:
        frames.append(f.readlines())

for i in range(10):
    for frame in frames:
        print("".join(frame))
        time.sleep(1)
        os.system('cls' if os.name == 'nt' else 'clear')
```

STEP 7 Save the Python code in the same folder as the text files and drop into a Terminal or Command Prompt. Navigate to the folder in question, and enter the command:

```
python NAME.py
```

Where NAME is whatever you called your saved Python code.



STEP 8 Here’s the code in full:

```
import os, time

os.system(‘cls’ if os.name == ‘nt’ else ‘clear’)

filenames = [“1.txt”, “2.txt”]
frames = []

for name in filenames:
    with open(name, “r”, encoding=”utf8”) as f:
        frames.append(f.readlines())

for i in range(10):
    for frame in frames:
        print(“”.join(frame))
        time.sleep(1)
        os.system(‘cls’ if os.name == ‘nt’ else
        ‘clear’)
```

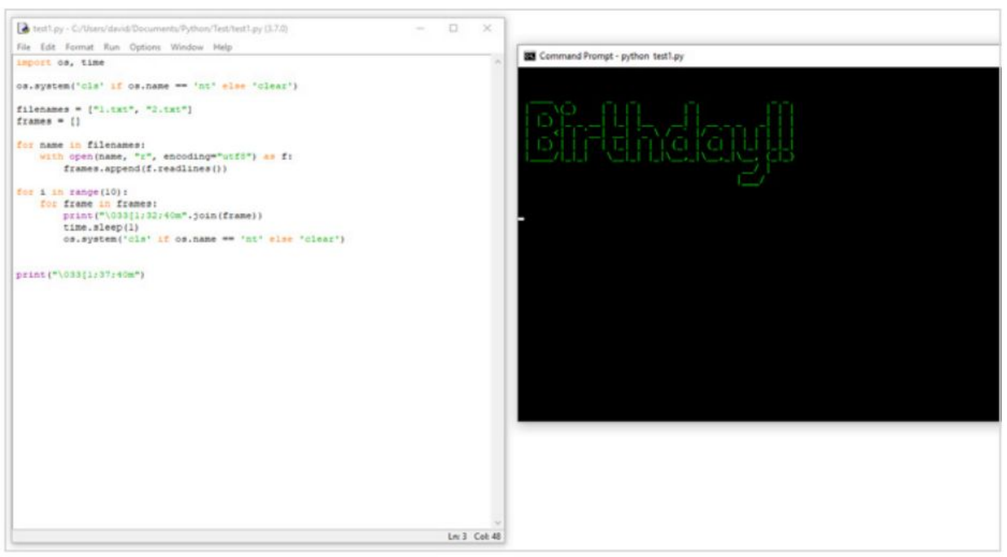
STEP 9 Note from the loop within the code, we’ve used the same CLS and Clear if/else statement as before. Again, if you’re running on Windows then the OS module will use the CLS command, ‘ELSE’ if you’re using Linux or a Mac, the Clear command will work correctly. If you want, you could use a Try/Except statement instead.

test1.py - C:/Users/david/Documents/Python/Test/test1.py (3.7.0)

File Edit Format Run Options Window Help

```
import os, time

'''os.system('cls' if os.name == 'nt' else 'clear')'''

try:
    os.system('cls')
except OSError:
    os.system('clear')


filenames = ["1.txt", "2.txt"]
frames = []

for name in filenames:
    with open(name, "r", encoding="utf8") as f:
        frames.append(f.readlines())

for i in range(10):
    for frame in frames:
        print("".join(frame))
        time.sleep(1)
        try:
            os.system('cls')
        except OSError:
            os.system('clear')
```

STEP 10 You can spice things up a little by adding system/Terminal colours. You’ll need to Google the system codes for the colours you want. The code in our example turns the Windows Command Line to green text on a black background, then changes it back to white on black at the end of the code. Either way, it’s a fun addition to your Python code.



Common Coding Mistakes

When you start something new you're inevitably going to make mistakes, this is purely down to inexperience and those mistakes are great teachers in themselves. However, even experts make the occasional mishap. Thing is, to learn from them as best you can.

X=MISTAKE, PRINT Y

There are many pitfalls for the programmer to be aware of, far too many to be listed here. Being able to recognise a mistake and fix it is when you start to move into more advanced territory, and become a better coder. Everyone makes mistakes, even coders with over thirty years' experience. Learning from these basic, common mistakes help build a better coding foundation.

SMALL CHUNKS

It would be wonderful to be able to work like Neo from The Matrix movies. Simply ask, your operator loads it into your memory and you instantly know everything about the subject. Sadly though, we can't do that. The first major pitfall is someone trying to learn too much, too quickly. So take coding in small pieces and take your time.

//COMMENTS

Use comments. It's a simple concept but commenting on your code saves so many problems when you next come to look over it. Inserting comment lines helps you quickly sift through the sections of code that are causing problems; also useful if you need to review an older piece of code.

```
        orig += 2;
        target += 2;
        --n;
    }
#endif
  if (n == 0)
    return;

  //
  // Loop unrolling. Here be dragons.
  //

  // (n & (~3)) is the greatest multiple of 4 r
  // In the while loop ahead, orig will move ov
  // increments (4 elements of 2 bytes).
  // end marks our barrier for not falling outs
  char const * const end = orig + 2 * (n & (~3)

  // See if we're aligned for writting in 64 or
#if ACE_SIZEOF_LONG == 8 && \
    !(defined(  amd64  ) || defined (  x86_64
```

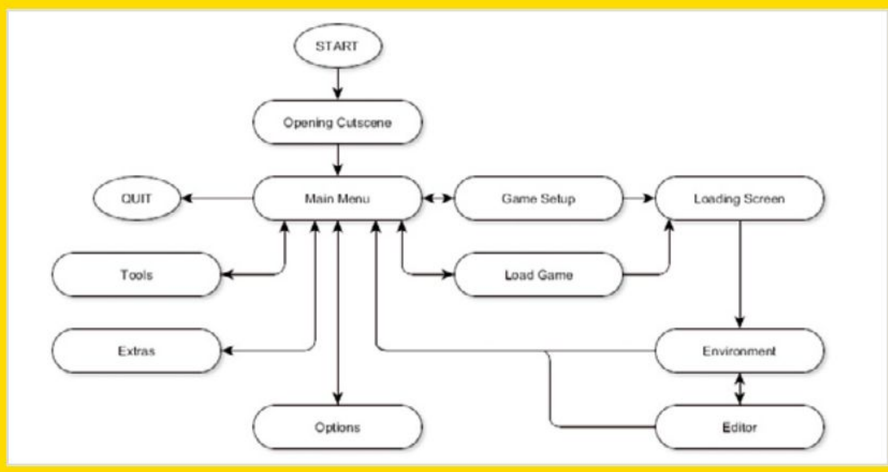
EASY VARIABLES

Meaningful naming for variables is a must to eliminate common coding mistakes. Having letters of the alphabet is fine but what happens when the code states there's a problem with x variable. It's not too difficult to name variables lives, money, player1 and so on.

```
var points = 1023;
var lives = 3;
var totalTime = 45;
write("Points: "+points);
write("Lives: "+lives);
write("Total Time: "+totalTime+" secs");
write("--------------------------");
var totalScore = 0;
write("Your total Score is: "+totalScore);
```

PLAN AHEAD

While it's great to wake up one morning and decide to code a classic text adventure, it's not always practical without a good plan. Small snippets of code can be written without too much thought and planning but longer and more in-depth code requires a good working plan to stick to and help iron out the bugs.



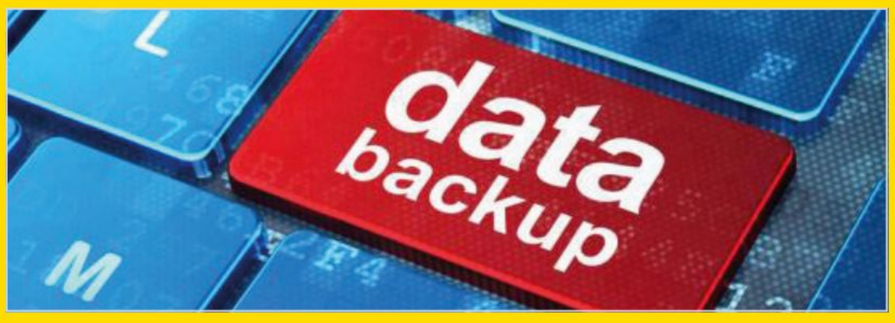
USER ERROR

User input is often a paralysing mistake in code. For example, when the user is supposed to enter a number for their age and instead they enter it in letters. Often a user can enter so much into an input that it overflows some internal buffer, thus sending the code crashing. Watch those user inputs and clearly state what's needed from them.

```
Enter an integer number
aswdfdsf
You have entered wrong input
s
You have entered wrong input
!"£"!£!"
You have entered wrong input
sdfsdf213213123
You have entered wrong input
123234234234234234
You have entered wrong input
12
the number is: 12

Process returned 0 (0x0)   execution time : 21.495 s
Press any key to continue.
```

BACKUPS

Always make a backup of your work, with a secondary backup for any changes you've made. Mistakes can be rectified if there's a good backup in place to revert to for those times when something goes wrong. It's much easier to start where you left off, rather than starting from the beginning again.



RE-INVENTING WHEELS

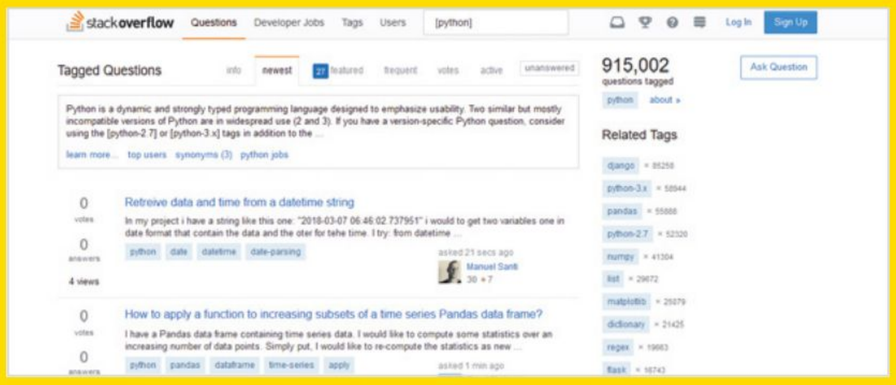
You can easily spend days trying to fathom out a section of code to achieve a given result and it's frustrating and often time-wasting. While it's equally rewarding to solve the problem yourself, often the same code is out there on the Internet somewhere. Don't try and re-invent the wheel, look to see if some else has done it first.

SECURE DATA

If you're writing code to deal with usernames and passwords, or other such sensitive data, then ensure that the data isn't in cleartext. Learn how to create a function to encrypt sensitive data, prior to feeding into a routine that can transmit or store it where someone may be able to get to view it.

HELP!

Asking for help is something most of us has struggled with in the past. Will the people we're asking laugh at us? Am I wasting everyone's time? It's a common mistake for someone to suffer in silence. However, as long as you ask the query in the correct manner, obey any forum rules and be polite, then your question isn't silly.



MATHS

If your code makes multiple calculations then you need to ensure that the maths behind it is sound. There are thousands of instances where programs have offered incorrect data based on poor mathematical coding, which can have disastrous effects depending on what the code is set to do. In short, double check your code equations.

```
set terminal x11
set output

rmax = 5
nmax = 100

complex (x, y) = x * {1, 0} + y * {0, 1}
mandel (x, y, z, n) = (abs (z)> rmax || n>= 100)? n: mandel (x, y, z * z + complex (x, y), n + 1)

set xrange [-0.5:0.5]
set yrange [-0.5:0.5]
set logscale z
set samples 200
set isosample 200
set pm3d map
set size square
a= #A#
b= #B#
splot mandel(-a/100,-b/100,complex(x,y),0) notitle
```

Python Beginner's Mistakes

Python is a relatively easy language to get started in where there's plenty of room for the beginner to find their programming feet. However, as with any other programming language, it can be easy to make common mistakes that'll stop your code from running.

DEF BEGINNER(MISTAKES=10)

Here are ten common Python programming mistakes most beginners find themselves making. Being able to identify these mistakes will save you headaches in the future.

VERSIONS

To add to the confusion that most beginners already face when coming into programming, Python has two live versions of its language available to download and use. There is Python version 2.7.x and Python 3.6.x. The 3.6.x version is the most recent, and the one we'd recommend starting. But, version 2.7.x code doesn't always work with 3.6.x code and vice versa.

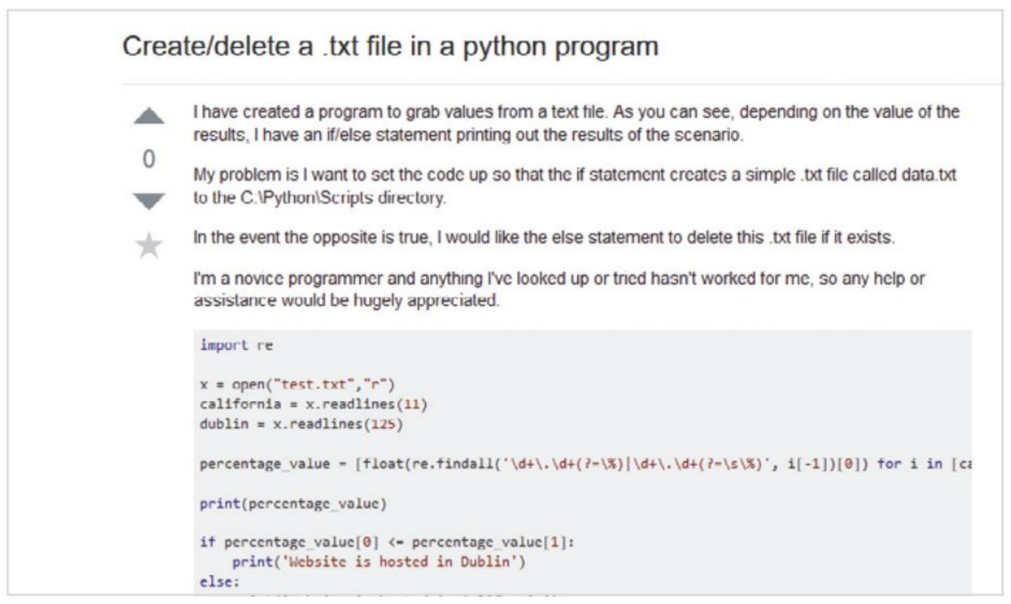
INDENTS, TABS AND SPACES

Python uses precise indentations when displaying its code. The indents mean that the code in that section is a part of the previous statement, and not something linked with another part of the code. Use four spaces to create an indent, not the Tab key.

```
MOVESPEED = 11
MOVE = 1
SHOOT = 15

# set up counting
score = 0

# set up font
font = pygame.font.SysFont('calibri', 50)

def makeplayer():
    player = pygame.Rect(370, 635, 60, 25)
    return player

def makeinvaders(invaders):
    y = 0
    for i in invaders:
        x = 0
        for j in range(11):
            invader = pygame.Rect(75+x, 75+y, 50, 20)
            i.append(invader)
            x += 60
        y += 45
    return invaders

def makewalls(walls):
    wall1 = pygame.Rect(60, 520, 120, 30)
    wall2 = pygame.Rect(246, 520, 120, 30)
    wall3 = pygame.Rect(432, 520, 120, 30)
    wall4 = pygame.Rect(618, 520, 120, 30)
    walls = [wall1, wall2, wall3, wall4]
```

THE INTERNET

Every programmer has and does at some point go on the Internet and copy some code to insert into their own routines. There's nothing wrong with using others' code, but you need to know how the code works and what it does before you go blindly running it on your own computer.



COMMENTING

Again we mention commenting. It's a hugely important factor in programming, even if you're the only one who is ever going to view the code, you need to add comments as to what's going on. Is this function where you lose a life? Write a comment and help you, or anyone else, see what's going on.

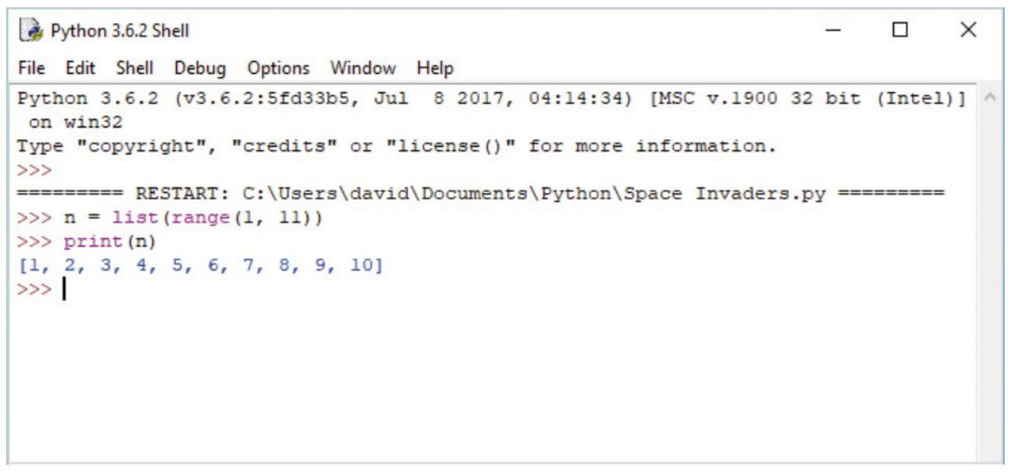
```
# set up pygame
pygame.init()
mainClock = pygame.time.Clock()

# set up the window
width = 800
height = 700
screen = pygame.display.set_mode((width, height), 0, 32)
pygame.display.set_caption('caption')

# set up movement variables
moveLeft = False
moveRight = False
moveUp = False
moveDown = False

# set up direction variables
DOWNLEFT = 1
DOWNRIGHT = 3
```

COUNTING LOOPS

Remember that in Python a loop doesn't count the last number you specify in a range. So if you wanted the loop to count from 1 to 10, then you will need to use:

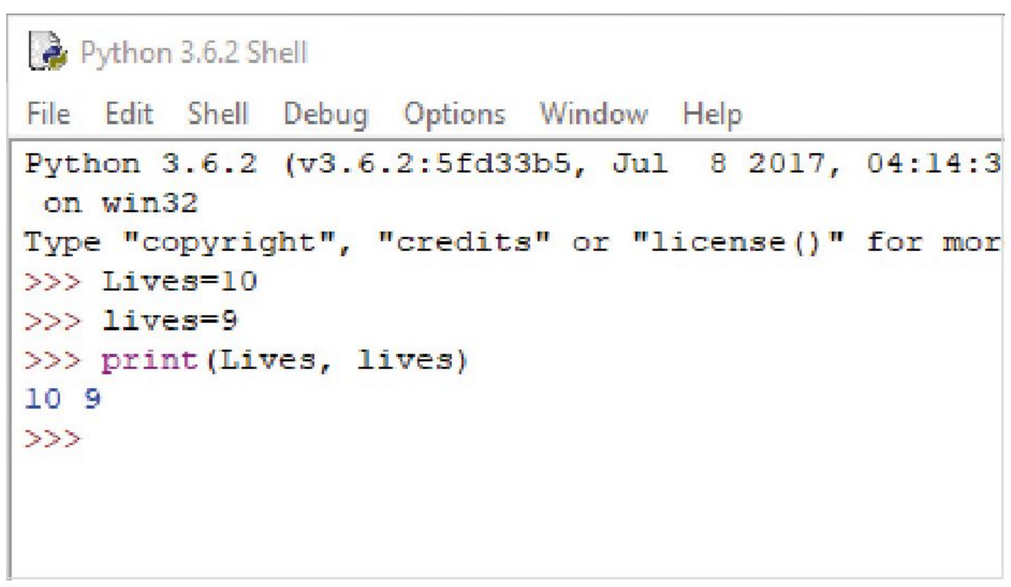
```
n = list(range(1, 11))
```

Which will return 1 to 10.



CASE SENSITIVE

Python is a case sensitive programming language, so you will need to check any variables you assign. For example, `Lives=10` is a different variable to lives=10, calling the wrong variable in your code can have unexpected results.



BRACKETS

Everyone forgets to include that extra bracket they should have added to the end of the statement. Python relies on the routine having an equal amount of closed brackets to open brackets, so any errors in your code could be due to you forgetting to count your brackets; including square brackets.

```
def print_game_status(self):
        print (board[len(self.missed_letters)])
        print ('Word: ' + self.hide_word())
        print ('Letters Missed: ',)
        for letter in self.missed_letters:
                print (letter,)
        print ()
        print ('Letters Guessed: ',)
        for letter in self.guessed_letters:
                print (letter,)
        print ()
```

COLONS

It's common for beginners to forget to add a colon to the end of a structural statement, such as:

```
class Hangman:
def guess(self, letter):
```

And so on. The colon is what separates the code, and creates the indents to which the following code belongs to.

```
class Hangman:
        def __init__(self,word):
                self.word = word
                self.missed_letters = []
                self.guessed_letters = []

        def guess(self,letter):
                if letter in self.word and letter not in self.guessed_letters:
                        self.guessed_letters.append(letter)
                elif letter not in self.word and letter not in self.missed_letters:
                        self.missed_letters.append(letter)
                else:
                        return False
                return True

        def hangman_over(self):
                return self.hangman_won() or (len(self.missed_letters) == 6)

        def hangman_won(self):
                if '_' not in self.hide_word():
                        return True
                return False

        def hide_word(self):
                rtn = ''
                for letter in self.word:
                        if letter not in self.guessed_letters:
                                rtn += '_'
                        else:
                                rtn += letter
                return rtn
```

OPERATORS

Using the wrong operator is also a common mistake to make. When you're performing a comparison between two values, for example, you need to use the equality operator (a double equals, ==). Using a single equal (=) is an assignment operator that places a value to a variable (such as, lives=10).

```
1 b = 5
2 c = 10
3 d = 10
4 b == c #false because 5 is not equal to 10
5 c == d #true because 10 is equal to 10
```

OPERATING SYSTEMS

Writing code for multiple platforms is difficult, especially when you start to utilise the external commands of the operating system. For example, if your code calls for the screen to be cleared, then for Windows you would use `cls`. Whereas, for Linux you need to use `clear`. You need to solve this by capturing the error and issuing it with an alternative command.

```
# Code to detect error for using a different OS
run=1
while(run==1):
    try:
        os.system('clear')
    except OSError:
        os.system('cls')
    print('\n>>>>>>>>>>Python 3 File Manager<<<<<<<<<<\n')
```