>>> in code means you should put the code in the console

Strings start and end with the same characters ('he said "Hello World" today' would print as *he said "Hello World" today*.

Three double-quotes denote multi-line quotes (or use \n)

Escape Characters

\n = new line

\t = puts 8 spaces between characters **(if there are more characters before it, it will use them to fill the space**

'\' ' = '

Numbers and Operators

+ adds strings together

* multiplies the string by a number of times (ha * 3 = hahaha)

len(string) returns the length of a string

str(), int(), float() convert into different types (int only works with whole numbers)

round() rounds……..come on man

// means whole number division ALWAYS ROUNDS DOWN

% can be used to find the remainder

% does not change (except the sign, and only if it was on the second number (# % -#))

Order of operation (left to right)

( ), **, *, /, //, %, +, - (sub not neg)

Print

flush = true used for debugging (prints in real-time)

Sep = sets the separator *("hello", "world", sep = "**"* will print hello**world)

End = same as *sep* but for line endings

input() = user input, returns string

.*lower()* changes it to lower case

.*title()* changes the first letter of each word to a capital

denotes a comment

Functions:

Use *def function():* to denote a function, then indent four spaces (tab?)

Tuples:

Tuples are a container-type that allows you to group values together

Ex: x = (1, 2, 3)

Ex2: y = 1, 2, 3

You can not change individual values in a tuple, only the whole thing

You can assign values to things in a tuple

Ex: *y = 2, 3, 4*

A, b = y

A = 2, b = 3 will be the result

You can use arrays to do things like *s = "Hello World" print(s[2])* will print "e"

Modules:

append()	Adds an element at the end of the list

clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

```
q = input('read or write?')

if q.lower() == 'read':

    #FOR READING
    f = open('example.txt')
    #reads first line then advances f to the next line
    #line = f.readline()
    #reads the stuff from f to the end of the file
    r = f.read()
    print(r)
    f.close()
elif q.lower() == 'write':
    #FOR WRITING
    #a is for 'append'
    f = open('example.txt', 'a')
    f.write('This  is text\nWhere is this anyways?')
    f.close()
```

```
'''
@author: [REDACTED]
'''
import urllib.request
```

```
thingread = 'http://www.greenteapress.com/thinkpython/code/words.txt'
file = urllib.request.urlopen(thingread)

read = file.read()
read = str(read).replace('\r', ', ')

f = open('newfile.txt', 'a')
f.write(str(read))
f.close()

'''
for i in read:
    print(i)
    #i.replace('\r', ', ')

print(read)
'''
```

Sets:

- s.add(x) — add an element if it is not already there
- *s.clear()* — clear out the set, making it empty
- *s1.difference(s2)* — create a new set with the values from s1 that are not in s2.
    - s1 - s2
- *s1.intersection(s2)* — create a new set that contains only the values that are in **both** sets
    - s1 & s2
- *s1.union(s2)* — create a new set that contains values that are in either set
    - s1 | s2
- *s1.issubset(s2)* —- are all elements of s1 also in s2?
    - s1 <= s2
- *s1.issuperset(s2)* — are all elements of s2 also in s1?
    - s1 >= s2

- *s1.symmetric_difference(s2*) — create a new set that contains values that are in s1 or s2 but **not in both**.
    - s1 ^ s2

Sorting:

Use the function *sorted()*

Dictionaries:

Declaring:

name = dict()

name = {}

Adding Values:

name['string'] = thing

You can also use the appropriate adding command to add to the dict (like *.add* for *sets*)

Calling:

name.keys() will give you the list of keys (*'string'*)

name.values() will give you the list of values (*'thing'*)

name.clear() empties the function

name.del() deletes a value

name.update(string:thing) update the value of string to thing

Classes:

Stores a type of data

Declared as *class className(object):*

thing = thing

OR 'pass' if there's nothing there

YOU CAN USE ATTRIBUTES!!!!!

p = Classname()

p.x = 10

p.y = 20

You can use *self* or *other* to refer to things

To initialize things inside a class at a certain point in code, use

```
class ClassName(object):
    def.__init__(self, x0, y0):
        self.x = x0
        self.y = y0
```

You can set *x0* and *y0* to a default by writing

If there are two underscores, then it is a special function

```
class Point2d(object):
    def __init__(self, x0=0, y0=0):
        self.x = x0
        self.y = y0
```

You can also define functions inside the class then use *class.function* to call them

Algorithms:

- Precise description of the steps necessary to solve a computing problem
- Description is intended for people to read and understand

Gradual refinement:

- Starts with English sentences
- Gradually, the sentences are made more detailed and more like programming statements
- Allows us to lay out the basic steps of the program before getting to the details.

A program is an implementation of one or more algorithms.

Tkinter:

- Is a module (use from tkinter import *)

To initiate use:

*From tkinter import **

root = tkinter.Tk()

Anything in here will repeat in the loop

root.mainloop()

#anthing that's written here will be executed after the mainloop is done

NOTE: This is an infinite loop

Types of widgets

Widgets are objects that have a function and are visible. These are the main elements of an interface

- Buttons: can do something when clicked
- Canvas: you can draw shapes or put shapes in them
- Radio boxes:
- Checkboxes:
- Menu of items:

Types of containers:

Containers are invisible, but they allow us to group widgets and arrange them in visual groups

- Frames: can contain any of the widgets of the above types

NOTE: A Container's size will grow and shrink to fit the objects contained within. Containers can contain other containers as well

Initiate a frame by using:

main_frame = Frame(root)

frame_name = main_frame(side = *insert side here**)*

button_name = Button(frame_name, *insert text here**)*

button_name.pack(side = *insert side here**)*

main_frame.pack(side = *insert side here**)*

Sides:

- TOP
- BOTTOM
- RIGHT
- LEFT

NOTE: All the elements (frame and button in this case) have parent elements that are declared BEFORE them. For example, *main_frame* has *root* as its parent and *button_name* has *frame_name* as its parent

The parent sets the position of the child and the size of the parent is determined by the number of children it has

Pack will attach the element to the GUI, so remember to use *.pack(side =* *insert side here**)*

NOTE: you can and should use classes as a way to simplify the process:

```
from tkinter import *

Class MyApp(object):

        def __init__(self, parent):

                self.main_frame = Frame(parent)

                You get the idea

        def terminate_program(self):

                self.parent.destroy()

root = Tk()

myapp = MyApp(root)

root.mainloop()
```

Clickable Buttons:

Use *self.button = Button(self.frame_name, text = "Quit", command = self.terminate_program)*

Where *terminate_program* is a def inside the class

Button Appearance:

Use *self.button.configure(width=#, padx='#', pady='#')*

Where pad controls how much space is between the edge of the frame and the button

Canvas:

A canvas is a blank area. You can continuously draw on it or put text on it.

Creating a canvas:

canvas = Canvas(parent, height=#, width=#)

canvas.pack()

Drawing in a canvas:

canvas.create_oval((#,#,#,#)) (remember that this is an image (so #1 and #2 will be the top left)

NOTE: you can force all drawing to happen at once with *canvas.update()* or have them draw after a bit using *canvas.after(waittime)*