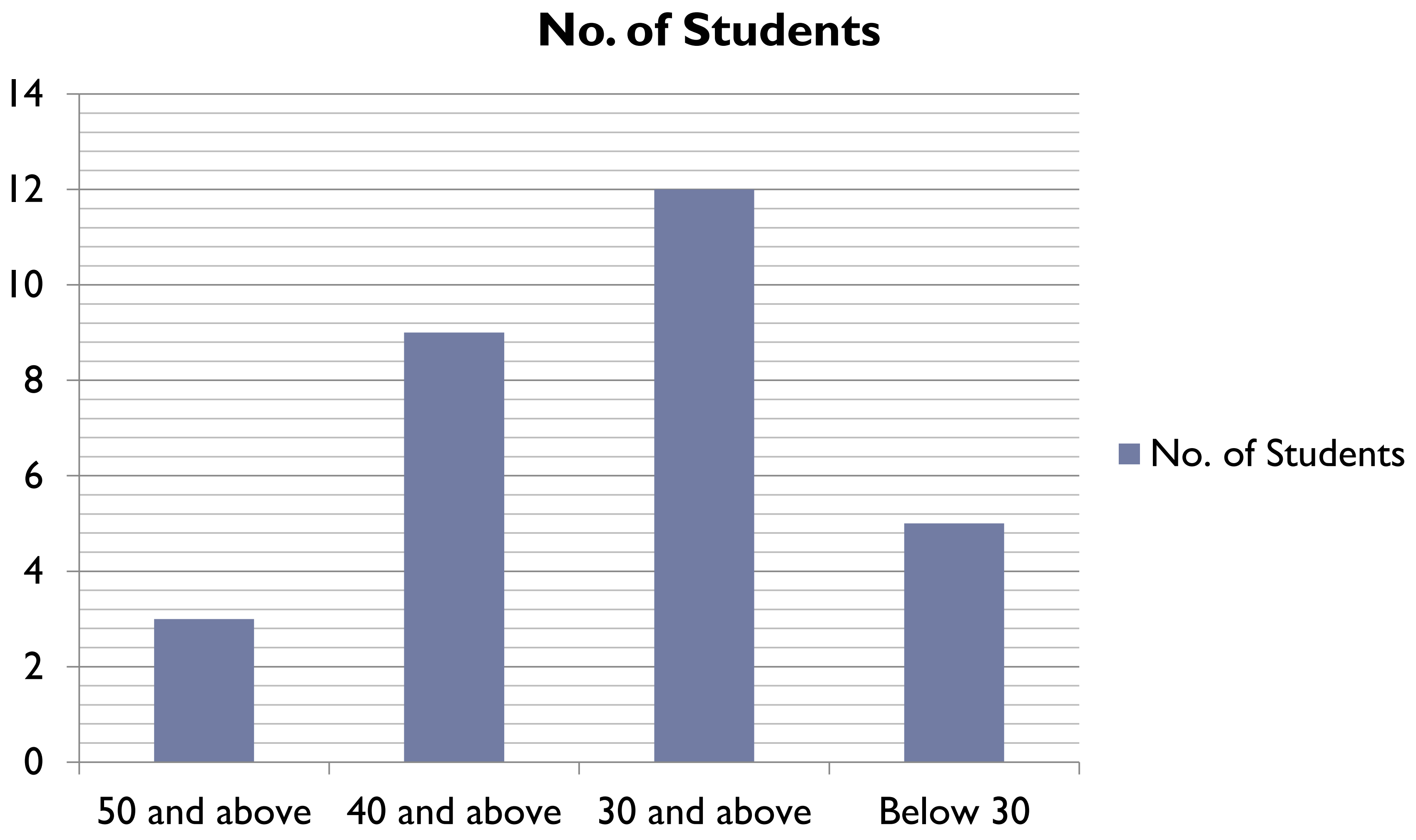
Week 2: Strings and String functions

Welcome Back!

Quiz 1 Statistics



Observations

- String formatting.
  - Solution: Don't worry this is what we will be covering this week.
- Use of return
  - Understanding that return is a key word to *return* values not store them
- Storing results
  - result = int(y) + x   …is correct
  - int(y) + x =result ….. Is incorrect
- Generality:
  - Why do we use variables?

A brief recap of Strings:

- Strings are sequences of letters or digits or special symbols.

- Strings are our first sequence. Specifically a string is a sequence of individual characters.

- More on the use of special symbols in Strings later.

Rules to follow for Strings

- X = ‘abc’

- Y = “Hey, how are you? I’m good.”

- Use only one type of quote, not both.

Notice that you can use either single or double quotes but they have to be matched

- Z = “this is a bad string’

FYI- a string of length 1 is usually called a character.

Strings and Operators

- Operators for strings are different than for numbers (i.e. int or float)
- X = ‘abc’
  Y = ‘ xyz ’
  print(X + Y)
  print(X * 3) this is called concatenation
  print(X + “.”)

  /, ^, **, // are not defined with strings

An important characteristic of a String: Length

- Since strings are sequences
  “abc” is really an “ a ” followed by a “b” followed by a “ c ”

  time for a new built in function
  len(“abc”)

  gives us the length of the string (also works on other sequences we’ll look at later)

- What use is the finding the length of a string?

Special Characters

- x = ‘ab\tc’
  ‘ \t ’ is rendered as a tab
  ‘ \n ’ is rendered as a new line (i.e. a return)
  ‘ \\ ’ is rendered as a backslash
  ‘ \’ ‘ is rendered as a single quote (why would you need this?)
  ‘ \” ‘ is rendered as a double quote (same reason)

- Try printing something like ‘This isn’t mine, you’re mistaken!’
- Is there a simpler way to avoid using special characters?

Multi Line Strings

- x = """

  “hey how's it going?“

  """

  Triple quotes (either single or double but not mixed) allow for multiline strings. These strings capture new lines and also deal well with quotes.

The Doc- String

```
def times_two(number):
"""

(num) -> num
times_two() takes a number and returns twice that number.
>>> times_two(5)
10
>>> times_two(3.4)
 6.8
 """

          return 2 * num
```

Doc-String

```
def times_two(number):
"""

(num) -> None
times_two() takes a number and returns twice that number.
>>> times_two(5)
10
>>> times_two(3.4)
 6.8
 """

          print( 2 * num)
```

Welcome
Back!

A Quick Recap

- Strings are our first sequence. Specifically a string is a sequence of individual chararacters.
- “**abc**”
- is really an “**a**” followed by a “**b**” followed by a “**c**”
- Python makes it easy for us to grab individual characters out of a string or runs of characters
- This is called slicing and it uses the square brackets which we have not used yet

But before that..

- A small introduction to Lists.
- Lists are a special type of ‘variable’ which can store multiple values. They are our second type of sequence.
- Denoted by [ ]
- More on this in Week 6.
- Strings can be represented as lists too.
- This way “abc” can actually be represented as [‘a’,’b’,’c’]

Slicing

- Remember that python uses *0-indexing*
- **X = 'abc'**
- **X[0]**
- gives us the first character of string X

  we could also have written "**abc**"**[0]**
- we can also use
- **X[1]**
  to get the second letter
- **X[2]**
  to get the last letter
- Note that the highest index for a letter is *1 less than the length*, again due to 0-indexing.

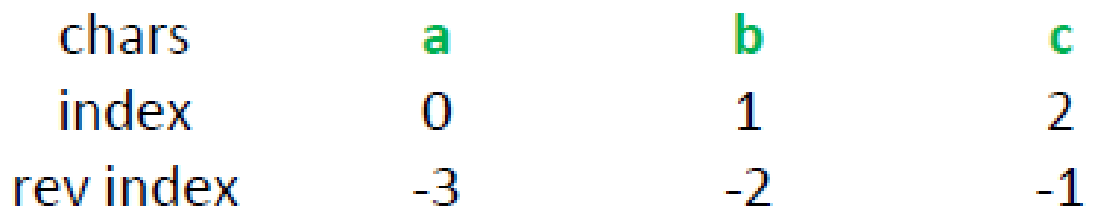
Reverse Indexing

- What if I want the last letter?
- **X = 'abc'**
  X[len(X) - 1]

  will work just fine but python also gives us an easier way
- **X[-1]**

- using a negative number starts at the end and counts backwards through the string
- Practice Question: How would I get the middle letter of " Computer Science"? Hint: Use len()

A Visual Representation



chars	a	b	c
index	0	1	2
rev index	-3	-2	-1

Sequence Runs

- What if I want a run of letters? Again not hard but we need new

syntax- [start:end]

X = 'abc'

X[0:2]

will give us "**ab**"

- Notice that the run includes the start index but stops ***before*** the end index

so

- **X[0:len(X)]**

  gives us the entire string. But again python has given us a short cut if we leave the start blank it will start at the beginning and if we leave the end blank it will end at the end of the string.

X[:2]

X[1:]

Questions?

- Slicing always returns a new string

  so “abc”[:2] + “xyz”[:1]

  works fine.
- Slicing takes a little time to get comfortable with but is amazingly handy.

- It has a new notation but you use exactly the same notation with other sequences (lists, tuples, dictionaries- all data structures in python)

String Methods

- Methods are functions associated with a specific object or type. They are normally *called* using the dot operator(?)

- Strings have lots of useful methods:

upper(), lower()

isupper(), islower(), isnumeric(), isalpha()

capitalize()

strip(), lstrip(), rstrip()

split()

lots more…

String Methods

- To use a string method you can call it like this:

‘abc’.upper()

or equivalently

X = ‘abc’

X.upper()

- in the second case did we change X?

String Constants

- In addition to the methods, strings have a number of constants.

However these constants are not built in, we need to import them.

```
import string                    #orange? New keyword, score!
print(string.ascii_letters)
```

other constants:
ascii_lowercase
ascii_uppercase
punctuation
digits
hexdigits
octdigits

- we'll talk more about import tomorrow

Some Cheesy Practice

- What do these do?
- **X = ‘Python’**
- **str1 = X[0]**
- **str2 = X[1:]**
- **print(str2.capitalize() + str1.lower())**
- **print(str1)**
- **print(str2[-1])**

Welcome Back!

Importing

- Libraries are nothing more than a bunch of code written for you that you can import and use in your own programs without having to credit the original authors.

- Hey, who doesn’t like free stuff?

Ways to Import

- two methods, they work slightly different in terms of how you then use the imported code. Assume we have a library called foo which has a function called bar().
- Method 1:

import foo

foo.bar()

- With this method any function you call from foo has to have "foo." appended to the front so python knows where to find the function. The import lets python know that foo.py exists

The other Method

- Method 2:

```
from foo import *
bar()
```

- With this method we can directly call any function in foo as if it were a built in function. The import pulls all the code in foo into the current project.

Why do we need two methods?

- Method 2 seems more convenient so why have 2 methods?

Imagine we have a library foo that has a function called bar() but we also have a library foo2 that has a different function called bar().

Method 1 allows us to use both bar functions calling them through

foo.bar()

and

foo2.bar()

Method 2 will overwrite the first bar you import with the second, you can only use one of them.

Hence for larger projects where you may have dozens of libraries and name collisions are not unlikely the first method is much safer.

Turtle

- Jokes apart, turtle graphics is one of the fun parts of Python.
- Based upon legacy programming language called Logo.
- Uses a relative cursor aka the ‘turtle’ upon a cartesian plane.
- A simple code to demo the use of a turtle

```
import turtle

turtle.forward(20)
```

What does this do?

What is the orientation of the ‘turtle’?

That’s cool, but what else can it do?

fd(), bk()	move forward, back
lt(), rt()	turn left, right
color()	change drawing color
begin_fill(), end_fill()	fill in closed polygons
reset()	reset the canvas
bye()	close the canvas
shape()	change appearance of the turtle
stamp()	stamp the turtle shape on the canvas
setpos()	move to certain coordinates

So how do we use it?

- Remember what we did with String methods?
- The use is similar, only in this case the *object* is the turtle.
- Hence, turtle.*method_name()* will invoke the method for that turtle object.

Question time!

- Using the turtle how would you draw a square?
- Well ok that was simple, can we try and break that down in to certain substeps? Maybe that would repeat?
- Ok seems like you guys came prepared today….go squirtle!!
- Try making a triangle?

Important point

- Something you shouldn’t do while saving your Turtle program is save it as “turtle.py”.

- Why?

Welcome Back!

More on Turtle Functions

- **turtle.pencolor(**‹*color*›**):**
  Sets pen color. Example: turtle.pencolor('green')
- **turtle.speed(**‹*speed*›**)**
  Set turtle drawing speed; 1 is slow, good for debugging through 9, fast; 0 is fastest, but does not show the turtle.
- **turtle.pendown()**
  Turtle leaves a visible trail - pen is down touching screen.
- **turtle.penup()**
  pen is lifted and does not leave a trail
- **turtle.shape(*name*=*None*)**
  there are the following polygon shapes: “arrow”, “turtle”, “circle”, “square”, “triangle”, “classic”

- **turtle.fillcolor(**‹*color*›**)**
  Sets fill color.
- **turtle.color(**‹*pencolor*›**,**‹*fillcolor*›**)**
  Sets both pen color and fill color.

- **turtle.begin_fill()**
  Turtle moves now begin to define an area for eventual fill.

  turtle.end_fill()
  Turtle has now finished defining a fill area; area fills with color.

Typical sequence to create a color fill:

```
turtle.fillcolor('yellow') # then
turtle.begin_fill()
# do some drawing such as a square or some other figure
turtle.end_fill() # Now you see the fill color
```

Turtle Position methods

- **turtle.goto(*‹x›*, *‹y›*)**

  Move turtle to the position x over, y down.

- **turtle.pos()**

  Return the x,y position of the turtle.

Example:

tp = turtle.pos()

print(tp) # (20.0, 40.0) for example

- Both of these methods have something to do with positional coordinates, what's the difference?

- What would be the output of turtle.goto(turtle.pos())?

- Would the reverse work as well? Why?

Helper Functions

- Remember when I asked you to break down the drawing of a square into repeatable steps?
- Helper functions allow us to do that.
- This is essentially an abstraction on a smaller scale of the practice of breaking the problem into sub-problems.
- So, can we break down the drawing of a square into repeated calls to a helper function?

An example

```
import turtle
def line_and_turn(length, angle):
        turtle.fd(length)
        turtle.rt(angle)

def square(size):
        line_and_turn(size, 90)          #do we need a left
        line_and_turn(size, 90)          turn version of line
        line_and_turn(size, 90)           and turn?
        line_and_turn(size, 90)

def triangle(size):
        line_and_turn(size, 60)          #is triangle()
        line_and_turn(size, 60)          correct?
        line_and_turn(size, 60)
```

Group exercise:

- Can you think of a helper function using the other turtle methods?

- Write a helper function for drawing a square and coloring it with a given color.

  Hint: color_square(square_size,square_color)

- How much easier would it be to use the square function?

A Quick Reminder

All Projects are due at 5:00 pm today.

Test tomorrow will cover String Methods and turtle graphics.

You will be allowed to use IDLE (like, d-uh!) as well as the Turtle reference that is present online.