CSSE 120 Cheat Sheet – Python at a Glance, Part 1 (v. 1, 3-18-2010)

Here are the concepts that you should become comfortable with by the middle of Week 3.

1. The ***input/compute/output*** pattern for programs
   a. The *input* function to get input from the console
   b. Using variables for numeric computation
   c. The *print* function to display results on the console

Example:

```
x = input("Enter a number: ")
y = input("Enter another number: ")
z = x ** y
print x, "raised to the", y, "power is", z
```

2. Getting ***input from the console***
   a. The *input* function to get input from the console
      ✓ The inputted value is evaluated before being returned by *input*
   b. The *raw_input* function to get input from the console
      ✓ The inputted value is returned "raw" as a string (i.e., as a sequence of characters)
   c. The *eval* function that relates *input* to *raw_input*
      ✓ *eval* takes a string and evaluates it
      ✓ **`input(…)`** is the same as **`eval(raw_input(…))`**

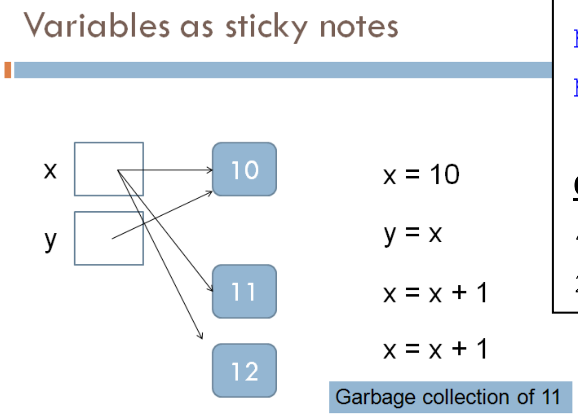
Example:

```
x = input("Enter a number: ")
y = raw_input("Enter a string: ")
z = y * x
print z
v = raw_input("Something to evaluate: ")
print eval(v)
```

Sample run of above (***red italic*** for what user typed:

```
Enter a number: 3
Enter a string: ok, now what?
ok, now what?ok, now what?ok, now what?
Something to evaluate: 7 * x
21
```

3. ***Variables*** and ***assignment***
   a. **`foo = blah`** read this as "foo gets blah" or "foo becomes blah"
      ✓ **`foo = foo + 1`** A common pattern that means "increment foo"
      ✓ **`foo += 1`** *is same as* **`foo = foo + 1`**
   b. Case matters. Style: use *namesLikeThis* for variables and *NamesLikeThis* for classes (more on classes later, but *Point* and *Rectangle* are examples)
   c. Variables are references. See diagram.
   d. **`x, y = blah, foo`** does assignment in parallel.


Variables as sticky notes

Example:

```
x = 47
x = x + 1
a, b = 10 * x, x ** 2
print x, a
print b
```

Output from the above:

```
48 480
2304
```

4. ***Arithmetic operators***

   a. `+ - * /` are as you would expect
   `**` for exponentiation (raising to a power)
   `%` for remainder
   `//` for integer division (discard the fractional part)

5. ***Printing*** on the console, ***strings***

   a. **`print blah, blah, …, blah`**

   ✓ Comma at end means don’t do a newline

   b. Expressions in quotes (single, double or triple quotes) are *strings*; printing them prints the string literally

Example: **Output from example**

```
x, y = 9, 2.5
a = x / y
b = x // y
print a, b                        3.6 3.0


c = x ** y
d = 19 % 4
print “c, d are”, c, d
                            c, d are 243.0 3
```

6. ***Calling (invoking) functions***

   a. Function name, open parenthesis, arguments to the function (separated by commas), close parenthesis.

   b. Calls (executes) the function, then returns control to the statement following the function call. The called function can return a value if it wishes.

   c. Don’t try to memorize all the functions! Instead:

   ✓ Use *autocomplete*: pause after typing a dot and see what functions you can apply (sometimes you have to backspace over the dot and retype it)

   ✓ Keep a Cheat Sheet of common functions

   ✓ Hovering over parts of a program gives you help

   ✓ In an interactive shell (e.g. in IDLE), use **`help(…)`**

Here are some functions that we have seen:

```
max   min   sum   abs   factorial
math.sqrt  math.cos   math.sin
int        float      str   round
time.sleep      type
help       help(__builtins__)
```

See below for sequence and list functions, as well as zellegraphics classes and functions.

7. ***Defining functions, parameters***

   a. The *def* keyword lets you define your own functions. See the example to the right for the notation.

   b. Functions can have *parameters* that are used in the body of the function.

   ✓ When the function is called, actual values are substituted for the formal parameters.

   ✓ The parameter names are local to the function definition; the same name used outside of a function has no relationship to the parameter name.

   c. Functions can *return values*, per the example to the right.

   d. Indentation denotes the body of the function (i.e., where the function definition begins and ends)

   e. You can put a *documentation string* as a string immediately after the *def* statement. Such strings are displayed by the *help* function. For example, typing **`help(factorial)`** produces the documentation string shown in the example to the right.

Example:

```
def factorial(n):
    """Returns n! (n factorial)"""
    product = 1

    for k in range(2, n + 1):
        product = product * k

    return product

print factorial(8)
print factorial(4)
print factorial(factorial(4))
```

Output from the above:

```
40320
24
620448401733239439360000
```

8. ***Modules*** and ***import***

   a. Some functions are built-in, others aren't

   b. **`import blah`** lets you refer to functions in the module *blah*

      ✓ e.g., **`import math`** lets you say **`math.sin(…)`**

   c. **`from foo import *`** lets you refer to all the public functions in the module *foo*, without needing to precede the function name with the module name

      ✓ e.g. **`from zellegraphics import *`** lets you say **`Point(…, …)`**

      ✓ Use with caution, as this "pollutes" your namespace with all the names in the module.

Example:

```
import math
print math.sin(0.4), math.pi

from zellegraphics import *

win = GraphWin()
p = Point(45, 32)
p.draw(win)
```

9. Using variables and number ***types***

   a. Numbers can be of type:

      ✓ *int* – fixed-length whole numbers (typically 32 bits, in which case they range from -2^{31} to 2^{31}-1, i.e., from about -2 billion to 2 billion)

      ✓ *float* – fixed-length numbers stored in a form of scientific notation. Allows a far greater range than *int*, but values are only approximate (although the precision is very high – typically about 10 digits)

      ✓ *long* – unbounded-length whole numbers (as big as you need them!) Python switches from *int* to *long* whenever a *long* is needed

   b. Operations on *int*'s always yield *int*'s (or *long* if necessary). Operations that mix *int*'s and *float*'s yield *float*'s.

   c. You can attempt to force a conversion with the functions *int*, *float* and *str*.

Example: **Output from example**

```
x, y = 9, 5
a = x / y
b = float(x) / y
print a                       1
print b                       1.8

print int(5.8)                5
print float(3)                3.0

z = 10 ** 16
print z              10000000000000000
print type(z)        <type 'long'>
```

10. ***Comments*** and ***help***

   a. If you put a # in your code, everything to the right of that # symbol is a *comment*

      ✓ Comments are ignored by the compiler (hence play no role in what the program does), but are critical for human readers of the code.

   b. *Documentation strings* document functions, modules, classes, etc.

   c. You can do **`help(blah)`** to get help on *blah*.

Example:

```
def truncate(x):
    """Returns a float that is the argument
    truncated to a whole number"""
    return float(int(x)) # x should be a number

print truncate(3.9)
help(truncate)
```

Output from the above:

```
3.0
Help on function truncate in module __main__:

truncate(x)
    Returns a float that is the argument
    truncated to a whole number
```

11. *Sequences*

a. *Sequences* can be *strings*, *tuples*, or *lists* (see below for details)

- ✓ There are other types of sequences too.

b. Use `x[k]` to refer to the kth element in the sequence *x*

- ✓ 0-based, so `x[0]` is the beginning element of the sequence, etc.
- ✓ `x[-1]` is the last element in the sequence *x*, `x[-2]` is the next-to-last, etc.

c. `x[m:n]` is a new list with elements from the mth element of *x* up to but not including the nth element of *x*

- ✓ So `x[:s]` is a new list with the elements of *x* up to but not including the sth entry
- ✓ And `x[r:]` is a new list with the elements of *x* from the rth entry to the end of the list

d. `x[m:n:k]` is a new list with every kth element in *x*, starting at the mth element of *x* up to but not including the nth element of *x*

e. Important functions/operations include:

len **index** **+** *****

12. *Strings*

a. Notation: elements in quotes (single or double), separated by commas

b. Immutable
(can't change the characters after the string is constructed)

c. Important string functions include:

capitalize **lower** **upper** **count**

find **replace** **split** **join** *lots more!*

13. *Tuples*

a. Notation: elements in parentheses, separated by commas

b. Immutable

14. *Lists*

a. Notation: elements in square brackets, separated by commas

b. Mutable – can change elements and add or remove elements

c. Important functions include:

range **append** **reverse** **sort** **count**

d. List comprehension – constructs a list from a list, see example

Example:

```
>>> list = [10, 20, 30, 40, 50]
>>> list[0]
10
>>> list[1]
20
>>> list[-1]
50
>>> list[1:3}
[20, 30]
>> list[3:4]
[40]
>>> list[0:5:2]
[10, 30, 50]
>>> len(list)
5
>>> list.index(30)
2
>>> list.index(900)
Error message
>>> list + [9, 7]
[10, 20, 30, 40, 50, 9, 7]
>>> list[1:3] * 4
[20, 30, 20, 30, 20, 30, 20, 30]
```

All the above works the same way with strings and tuples.

```
>>> s = “this is a string”
>>> t = (“this”, “is”, “a tuple”)
```

Split/Join example:

```
>>> s = “What is this stuff?”
>>> list = s.split()
[‘What’, ‘is’, ‘this’, ‘stuff?’]
>>> “ “.join(list)
‘What is this stuff?’
```

List comprehension example:

```
>>> list = [2, 4, 6]
>>> [k **3 for k in list]
[8, 64, 216]
```

15. ***Loops***

a. *Definite loops* are loops with a ***for*** statement

b. *Counted loops* when loops over a range

c. ***Accumulator pattern***, typical example:

```
total = 0
for k in range(100):
    total = total + math.sin(k)
```

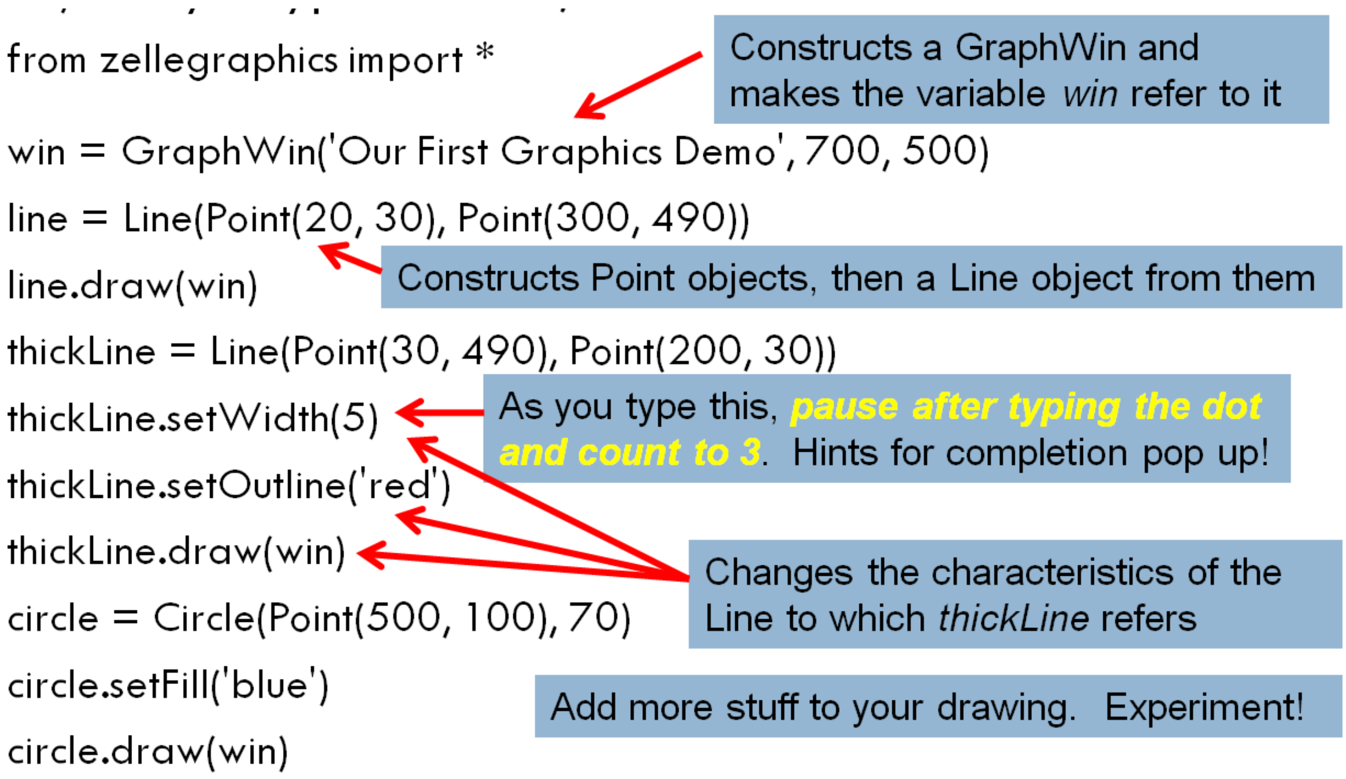
d. *Looping through a list, with a range statement:*

```
list = ...
for k in range(len(list)):
    ... list[k] ...
```

e. *Looping through a list, without a range statement:*

```
list = ...
for element in list:
    ... element ...
```

16. ***zellegraphics***



```
from zellegraphics import *

win = GraphWin('Our First Graphics Demo', 700, 500)
line = Line(Point(20, 30), Point(300, 490))
line.draw(win)
thickLine = Line(Point(30, 490), Point(200, 30))
thickLine.setWidth(5)
thickLine.setOutline('red')
thickLine.draw(win)
circle = Circle(Point(500, 100), 70)
circle.setFill('blue')
circle.draw(win)
```

Constructs a GraphWin and makes the variable *win* refer to it

Constructs Point objects, then a Line object from them

As you type this, ***pause after typing the dot and count to 3***. Hints for completion pop up!

Changes the characteristics of the Line to which *thickLine* refers

Add more stuff to your drawing. Experiment!