A cheat sheet for digital world. Save a local copy so that you can refer to it.

Digital World: Cheat Sheet

Manipulating strings

Single quote or double quote? It doesn't matter. Just **be consistent**

```
print "Hello World"
print 'Hello World'
print 'He turned to me and said \"Hello there\"'
```

Common string formatting tricks

```
newLine = 'Backslash n creates a new line\n'
symbols = 'Use backslash for symbols such as \%s and \" that would otherwise screw
up the string'
formatter = 'String uses %s, decimals use %d, and fixed point uses %f (with a
default precision of 6)'%(aString, aDecimal, aFixedPoint)
padding 'This gives left padding %5s'%(aString)
```

Slicing strings

```
s = '0123456789'
s[1:] # from index 1 to end. Gives you '123456789'
s[:7] # from start to index 7 - 1, much like how range works. Gives you '0123456'
s[:-1] # from start to end - 1. Gives you '012345678'. THIS IS VERY USEFUL
```

Strings are immutable. That means you can't replace characters like lists

```
s = 'Hello!'
s[0] = 'B' # this will return an error
s = 'B' + s[1:] # this is the proper way to do it
```

Other cool methods

```
s = 'test'
s.upper() # gives you TEST
s.find('e') # gives you 1 because 'e' is in index 1 of 'test'
s.find('z') # gives you -1 when it fails to find anything
'e' in 'Test' # gives you True
s.islower() # gives you False as 'T' is capitalised
up = s.upper()
up.isupper() # gives you True as s.upper() gives you TEST
```

Variables

Names

Don't touch these names! These are Python ***keywords***

```
"and as assert break class continue def"
"def elif else except exec finally for"
"from global if import in is lambda not or"
"pass print raise return try while with yield"
```

Types

Checking the type of variable is quite simple

```
type(variable)
```

Type conversion

```
strToInt = int(string)
intToFloat = float(integer)
anythingToString = str(anything)
```

Using **eval** converts a string to whatever it should be

```
string = '[1, 2, 3, 4, 5]'
listString = eval(string) # listString will be a list
```

Operations

Division is weird. Dividing integers will not give you floats

```
print 10/3 # gives you 3
print float(10/3) # you may think this gives you 3.33, but no. This gives you 3.0
```

Use modulo (%) to your advantage. Find out if something is a multiple of another this way

```
def isEven(num):
    return num % 2 == 0

def isMulipleOfSeven(num):
    return num % 7 == 0
```

Booleans. Look at the example above. Anything that uses >, < or = is gives you a boolean. So is a wealth of other things

```
3 in [1, 2, 3, 4, 5] # gives you True. 3 is found within the list
3 > 4 # gives you FalseÂ
30 / 10 == 3.0 # although 3 is int and 3.0 is float, this will return True
'abc' < 'xyz' # gives you true. Imagine if letters have values. This is like saying
123 < 456
```

Let's chain these together using **and** or **or**!

```
# and
# both must be True to return True
True and True # gives you True. This is the only scenario that gives you True.
True and False # gives you False.
# or
# if either is True, it will return True
True or False # gives you True.
False or False # gives you False. This is the only scenario that gives you False
```

Remember to float the numerator/denominator when you do division

```
floatPnt = 3/2.0 # method 1: adding a .0
floatPnt = 3/float(2) # method 2: surround with float()
floatPnt = (integer + 0.0)/2 # method 3: adding a + 0.0
```

To round down a float, use **math.floor**, or simple type cast it to an **int**. To round up, use **math.ceil***

```
math.floor(3.14) # gives you 3.0
```

```
int(3.14) # gives you 3
math.ceil(3.14) # gives you 4.0
```

You can multiply a string or a list

```
'a' * 3 # gives you 'aaa'
[False] * 3 # gives you [False, False, False]
```

Functions

print VS return

print only prints to console. **return** allows you to assign stuff to variables

```
def giveOne():
    print 1

case1 = giveOne() # 1 will be printed to console, but case1 will be None
print case1 # just in case you don't believe me

def giveTwo():
    return 2

case2 = giveTwo() # nothing is printed, but case2 will be quietly assigned 2
print case2 # just in case you don't believe me
```

Parameters and arguments

Here, the variable **num** exists only inside the function. Whatever variables created in the function disappears when the function ends

```
def repeatA(num):
    abc = 3
    return 'A' * num

print repeatA(3) # gives you 'AAA'
print abc # gives you an error: name 'abc' not defined
```

Use global to modify a global variable. Don't use it too often though!

```
count = 0

def increCount():
    print count # you don't need to global count before you read it
    global count # but doing this allows count to be modified
    count += 1
    # you need not return anything in a function
```

return as the end point of a code

```
def code():
    return 3 # the code stops HERE.
    a = "test" # these lines
    print a # will not be executed
```

Conditionals

Booleans in Conditionals

Please be efficient in your code!

```
# don't do this
if boolean == True:
    ...
# do this instead
if boolean:
    ...

# Similarly, don't do this
def compareAB(a, b):
    if a > b:
        return True
    else:
        return False
# do this instead
def compareAB(a, b):
    return a > b # (a > b) IS A BOOLEAN!
```

Not all **if** needs an **else**. If **else** means "don't do anything", don't even bother with using **pass**

```
if (boolean):
    a = "something"
else:       # these two lines
    pass    # are totally unnecessary
```

Recursion

Understanding range

range gives you a list!

```
range(10) # gives you [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(start, end) # gives you [start, start+1, start+2, ..., end-1]
range(start, end, step) # gives you [start, start+step, start+step*2, ..., end-1]
assuming end-1 is start+step*n
range(0, 21, 5) # gives you [0, 5, 10, 15, 20]
range(0, 20, 5) # gives you [0, 5, 10, 15]
```

for loops

for loops always work this way

```
for item in iterable:
```

Don't forget: range is a list, which is a iterable. Strings and tuples are iterables too!

for VS while

Use **for** if you are absolutely sure how many steps you want to take. Use **while** if you aren't quite sure when the program will end

```
condition = True
while condition:
    ...
    if (somethingHappens): # loop will stop when somethingHappens is met
        condition = False

count = [False] + [True] * 4 # gives you [False, True, True, True, True]
for i in range(4): # range(4) gives you [0, 1, 2, 3]. i will loop through this list
    # you may or may not need to use i
```

```
    count.pop(len(count) - 1)

print count # since you looped through the list exactly 4 times, all the True will
be popped
```

break

Stop a recursion immediately by using **break**

```
while True: # while True is typically dangerous: it runs an infinite loop
    line = raw_input('> ')
    if line == 'done':
        break # however, this allows the program to escape the infinite loop if
line == 'done'
    print line
```

Recursive Functions

```
# a function that doesn't require any evaluation of numbers
def printTo10(num):
    print num
    if num < 10:
        printTo10(num+1)
# eg printTo10(7)       >> 7        >> calls printTo10(8)
#    printTo10(8)       >> 8        >> calls printTo10(9)
#    printTo10(9)       >> 9        >> calls printTo10(10)
#    printTo10(10)      >> 10

# the classic fibonacci example
def fibonacci(n):
    if n == 0:
        return 0 # function stops recursion here
    elif n == 1:
        return 1 # function stops recursion here
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
# eg                            fibonacci(3)
#             fibonacci(2)           +           fibonacci(1)
#       fibonacci(1) + fibonacci(0)                 return 1
#         return 1       return 0
```

Input

raw_input vs **input**. Think of it this way. The raw_input gives you a string of what you write. input gives you eval(your_input) instead.

```
raw = raw_input("Enter raw_input here: ")
print type(raw) # this should give you a string based on whatever you typed

noRaw = input("Enter input here: ")
print type(noRaw) # Python will interpret whatever you input. eg if you input '[1, 2, 3]', this will be a list
```

Reading file

Opening a file

Make sure the file is in the same directory as your .py file

```
fread = open('words.txt', 'r') # open for reading
fwrite = open('words.txt', 'w') # open for writing
```

Reading content of file

There are two ways to go about it. Reading the whole thing, or reading line by line.

```
wholeFile = fread.read() # this dumps the entire file content into the string variable wholeFile
test = fread.read() # if you try to read again, it doesn't work any more
fread.close()
```

Line by line. There will be a \n at the end of every line though, so use .strip() to remove it

```
fread = open('words.txt', 'r') # let's try again
firstline = fread.readline() # reads the a string until \r\n
firstline = firstline.strip() # strip removes the trailing \n
secondline = fread.readline() # reads from where it stopped until the next \r\n
fread.close() # remember to close it when you are done
```

Use a **for** loop to loop through the file line by line!

```
fread = open('words.txt', 'r')
for line in fread:
    word = line.strip()
    print word
```

Writing

```
fwrite = open('words.txt', 'w')
line1 = "Line 1\n"
line2 = "Line 2\n"
fwrite.write(line1) # whatever written in a file MUST be a string
fwrite.write(line2)
fwrite.close() # remember to close the file!
```

Lists

Tricks

```
l = [3, 2, 1]
l.append(4) # l is now [3, 2, 1, 4]
m = [6, 5]
l.extend(m) # extend takes all the elements of m and append them to l
# l is now [3, 2, 1, 4, 6, 5]
l.sort() # l is now [1, 2, 3, 4, 5, 6]
sum(l) # gives you 21
popped = l.pop(0) # removes the index 0 from l, and returns what it popped, so
popped = 1
# l is now [2, 3, 4, 5, 6]
l.remove(5) # this removes the element with the value 5, NOT index 5!
# l is now [2, 3, 4, 6]
l *= 2 # l is now [2, 3, 4, 6, 2, 3, 4, 6]
del l[1:5] # this removes index 1 to index 4 of l, from l. ie [3, 4, 6, 2] is
removed
# l is now [2, 3, 4, 6]
print l[::-1] # iterates from start to end. Backwards. This gives you [6, 4, 3, 2]
```

Pointers

When you assign a list to a variable, the variable is merely a point to the string object

```
a = [1, 2, 3]
b = a
b[0] = 3 # modifying b modifies the object that it is pointing to.
print a # this will give you [3, 2, 3], since a and b are pointing to the same
object
```

Deep Copy

Deep Copy creates ANOTHER object with the exact same contents. This will prevent modifying the original object when the new object is modified

```
import copy

a = [1, 2, 3]
b = copy.deepcopy(a)
b[0] = 3
print b # b is now [3, 2, 3]
print a # a is still [1, 2, 3]
```

Dictionaries

Dictionaries are **key-value** pairs. Keys and values can be anything, really. Just remember, keys have to be unique!

```
eng2sp = {'one': 'uno', \
            'two': 'dos', \
            'three': 'tres'} # use black slashes to make your code more readable

eng2sp.keys() # gives you ['one', 'two', 'three']
eng2sp.vals() # gives you ['uno', 'dos', 'tres']

t = [('a', 0), ('b', 1), ('c', 2)]
d = dict(t)
# d is a dictionary: {'a': 0, 'b': 1, 'c', 2}
```

Lookup

dict[key] returns you the value of the corresponding key

```
eng2sp['one'] # gives you 'uno'
```

```
eng2sp['one'] = 1 # this changes the value of 'one' to 1
```

Iteration

If you want to do a reverse lookup, you have no choice but to iterate through every key, value pair

```
for k in eng2sp:
    v = eng2sp[k]
    print k, v

for k, v in eng2sp.iteritems():
    print k, v

def reverse_lookup(value):
    output = []
    for k, v in eng2sp.iteritems():
        if v == value:
            output.append(k)
    return output
```

Classes

A typical class

```
class Object():
    count = 1 # this is a static variable. You can read/modify only through
Object.count

    def __init__(self, x, y): # this assigns the arguments to the respective
attributes upon initialisation
        self.x = x
        self.y = y

    def getX(self): # always create a get method for each attribute in an object
        return self.x

    def getY(self):
        return self.y

ball = Object(1, 2)
```

```
print ball.x # this gives you x
print ball.getX # but please use this method to get the x value instead
Object.count += 4
print Object.count # this gives you 5
```

Special methods

```
class Time():
    def __init__(self, hour, minute, second):
        self.hour = hour
        self.minute = minute
        self.second = second

    def __str__(self): # this allows the object to be printable
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)

    def __add__(self, other): # this allows you to add two SGD objects together
        second = (self.second + other.second) % 60
        minute = (self.minute + other.minute) % 60 + (self.second + other.second) /
60
        hour = (self.hour + other.hour) % 24 + (self.minute + other.minute) / 60

        return Time(hour, minute, second)
        # remember to return an OBJECT. Time + Time gives you a new Time after all

    def __call__(self, hour, minute, second): # this allows you to use the object
as if it's a function
        # in this example, this reinitalises the Time to the values given in the
arguments
        self.hour = hour
        self.minute = minute
        self.second = second
        # it also returns the new time in a string
        return self.__str__()

    def __cmp__(self, other): # this allows you to use the >, < or == boolean
operators on the object
        if self.hour > other.hour: return 1 # return 1 to show that self > other
        if self.hour < other.hour: return -1 # return -1 to show that self < hour
        # if self.hour == other.hour, proceed
```

```
        if self.minute > other.minute: return 1
        if self.minute < other.minute: return -1
        # if self.minute == other.minute, proceed

        if self.second > other.second: return 1
        if self.second < other.second: return -1
        # if self.second == other.second, proceed

        return 0 # return 0 to show that self == other
```

Inheritance

A simple example. Inheritance allows you to define a template for whatever that inherits it.

```
class Account:
    def __init__(self, initialBalance):
        self.currentBalance = initialBalance
    def balance(self):
        return self.currentBalance
    def deposit(self, amount):
        self.currentBalance = self.currentBalance + amount
    def creditLimit(self):
        return min(self.currentBalance * 0.5, 10000000)

class PremierAccount(Account): # This INHERITS all the methods from Account
    def creditLimit(self): # This OVERWRITES the creditLimit method from Account
        return min(self.currentBalance * 1.5, 10000000)

class EconomyAccount(Account):
    def creditLimit(self): # This OVERWRITES the creditLimit method from Account
        return min(self.currentBalance*0.5, 20.00)
```

Let's use the SM class as example

```
class SM:
    def start(self):
        self.state = self.startState

    def step(self, inp):
        (newState, output) = self.getNextValues(self.state, inp)
        self.state = newState
```

```
        return output

    def transduce(self, inputs):
        self.start()
        return [self.step(inp) for inp in inputs]
```

Wait a minute, where is self.state and self.startState? What is self.getNextValues()?

We shall define them in the class that inherits SM.

```
class Accumulator(SM):
    def __init__(self, value): # since SM has no __init__, you must have one here
        self.startState = value # since SM has no self.startState, you must declare
it here

    # SM references to a self.getNextValues method, you must define it here
    def getNextValues(self, state, inp):
        return state + inp, state + inp
        # the (newState, output) tuple in the SM class step method receives this
return value
```

Note: when you are actually using it, you will call only the method from the **SM** class, not the classes that inherit from it. This is a good code structure.

Here is a different example:

```
class Elevator(sm.SM):
    startState = 'First' # declare your starting state

    def getNextValues(self, state, inp):
        # declare the default nextState and output values
        nextState = state # nextState is state by default, only modified if certain
conditions happen
        output = False # output is False by default, only modified if certain
conditions happen

        if state == 'First':
            if inp == 'Up':
                nextState = 'Second'
                output = True
```

```
        elif state == 'Second':
            if inp == 'Up':
                nextState = 'Third'
                output = True
            elif inp == 'Down':
                nextState = 'First'
                output = True

        elif state == 'Third':
            if inp == 'Down':
                nextState = 'Second'
                output = True

        return (nextState, output) # try to use only ONE return in the entirety of
getNextValues
```