Python Cheat Sheet

Variables

```
4/3
# Returns 1.3333333333333333

4//3
# Returns 1 (ignore remainder)

5/2
# Returns 2.5

5//2
#returns 2

4 % 2
# returns 0
```

int : The Python data-type used to represent integers is int. When we add or multiply two integers we always get an integer back. Not necessarily so when we divide.

float The Python data-type used to represent numbers with a fractional part is called `float`. When either operand in an addition, subtraction, or multiplication is a `float`, the result is a `float`. The result of floating point division is always a float.

str: A string is a sequence of characters. The Python data-type that represents strings is `str`. In Python we represent string literals as characters with single or double quotes around them. We can store strings in variables just like we do with numbers.

Assignment

= is assignment operator. NOT like math class, it assigns things.

```
my_int = 27
another_int = 5
a_float = 3.14159

my_int == 28
#False

my_int == 27
#True

my_int == 27.0
#True

"hello" == "hello"
#True
```

Variable Creation

When you first use a name, you add it to the namespace. Rules about the namespace:

1. Name may be of arbitrary length of ONLY letters, digits and underscores
2. Every name must begin with a letter or underscore, NO NUMBERS
3. CANNOT be keyword (like `type`)
4. Names ARE case-sensitive

Object and Types

```
x = [’a’,’b’] #This is a list. We will talk
    about these in mode detail later.
y = x  # assign a second variable name to the
    same list object

id(x)
# returns number from namespace (i.e.
    4472951232)

type(x) # get the type of the object, which is
    also an attribute
# returns list

len(x) # query the attribute length
# returns 2

x is y
#True

z = [’a’,’b’]

x is z
# False
```

Replacing values in same list objects:

```
x[0] = ’d’
x
# [’d’, ’b’]

y
#[’d’, ’b’]

y[0] = ’e’
y
# [’e’, ’b’]

x
#[’e’, ’b’]

z
[’a’, ’b’]
```

If two lists are the ”same” as in same values, but not the same list object, then you can check using `==` not, `=`

Conversion

Every object has a type and the type can never change! Variables however do not have a type and can point to any object. To covert variables, simply wrap them:

```
int("42")
# 42

float(27)
#27.0
```

Statements vs Expressions

Expressions: are evaluated and return a value. The value is displayed to the programmer on the console or can be assigned to a variable.
Statements Statements do not return a value, but they have side effects, such as printing to the terminal so the user can see the output.

```
(24 + 6) * 5 #expression, evaluates to 150

s2 = "hello 1006" #statement, doesn’t evaluate
    to anything. Just changes the state of the
    program.

s2 #expression, returns a value

s2 == "hi" # expression, returns a BOOLEAN
    value

print(s2) # this is an expression, but it
    returns None
```

Control

Selection: Making a decision about how to proceed in a program, based on the value of some variables
Repetition: Performing an operation over and over, either on different objects (for example each element of a list) or until some condition is met.
Bool The Python data-type used to represent the Boolean values True or False. We may use use the boolean operators not, and, or together with booleans.

If statement

```
a = 12
b = 10

if (a > b):  # compound statement
    print("Switch.")

    c = a
    a = b
    b = c

print(b-a)
```

Modules

We just used a module (‘random‘). Modules are collections of Python commands (functions and classes) that provide special functionality. We will encounter a number of different modules in this course.

```
import random #this statement is necessary
    because we will be importing the ‘random‘
    module

random.random() #This function in the ‘random‘
    module also has the name ‘random‘, hence
    the ‘random‘ _dot_ ‘random()‘ call. It
    returns a ‘float‘ between 0 and 1, not
    including 1. When we multiply that number
    by three, add 1 to it, and then convert it
    to an ‘int‘, we get a random number that is
     equally likely to be a 1,2, or 3. There
    are other functions for doing this without
    all the arithmetic and you are free to use
    them; all that is really required though,
    is ‘random.random()‘
```

If Else

```
 if boolean_expression1:
    indented stuff here. This block is only
    executed if the boolean
    expression evaluates True...
    more indented stuff here
    ...
elif boolean_expression2:
    if boolean_expression1 was False and
    boolean_expression2 evaluates
    True, run this stuff.
else:
    This block is only executed if neither
    boolean expression was True.
```

Break

Sometimes it is convenient to interrupt a while loop before we finish the suite of the statement. break immediately terminates the inner-most loop. Warning: some people consider this bad style (you can no longer predict the looping behavior by looking at the variables in the boolean expression only)

```
for i in range(3):

    x = 0
    while True:
        print(x)
        x += 1
        if x > 10:
            break

    print("done.")
```

Continue

The continue statement skips the remaining lines of the suite and jumps back to the header.

```
x = 0
while x < 10:
    x += 1
    if x % 2 == 1: # x is an odd number
        continue
    print(x)
```

Lists

```
li = []

li.append("hello")

li.append("hello")

li.append("test")

li
#['hello', 'hello', 'test']
```

For statement

A common operation is to perform some part of an algorithm on each element of a collection of items. A collection is a single object in Python that contains multiple elements, for example a list [0,5,8,9] or a string "hello". Most collections in Python are also iterables which means that the for statement can iterate over them. We will see how to deal with lists and other collections later in this course.

```
for x in ['some','stuff', -42, True, None]:
    print(x)

#same as:

li = ['some','stuff', -42, True, None]
for x in li:
    print(x)

#same as

i = 0
while i < len(li):
    print(li[i])
    i += 1

"""
some
stuff
-42
True
None
"""
```

Algorithms

String