Week 3 (Wednesday) [No lecture on Monday - Labor Day]

Let's summarize what we have done so far to get to the programming stage:

Variables

x, abcd, Abcd, ABCD, Joe ….

(Use variables that have meaning, i.e., total = sum1 + sum2 + sum3)

Algebraic Expressions and Functions

```
>> 3+8
11

>> 3*2
6

>> 3*2.0
6.0
```

>> 5/2
2.5

>> 4/2
2.0

5 has no decimal point, so it is type int (int means “integer”)

5.0 has a decimal point, so it is type float (float means “floating point”)

Why call it “floating point”? Because the decimal point can be moved anywhere
Convenient (say, to represent the number) without changing its value.

5.5 is same as 0.55 x 10

You can keep moving the decimal point right or left

In a computer word, 5.5 will be represented as 0.55 x 10

You already know how to represent 0.55 in binary. Use the 48 right-most bits (called “lower order

bits") to represent 0.55 in binary,
and use the 16 left-most bits (called "Higher order bits") to contain the number 1. The understanding is that 1 is the exponent of 10.

This is how floating point numbers are represented .... by splitting a 64-bit word
Into two pieces, the low 48-bits holding the fractional part ("mantissa"), and the
high 16-bits holding the exponent (with the understanding that it's the exponent of
10).

Accuracy

This means the size of the exponent is constrained by 16 bits. So how large or how small can the exponent be?

Let's see. Suppose a computer word had only 3 bits. How many numbers could it hold?

000 = 0
001 1
010 2

011	3
100	4
101	5
110	6
111	7

So 3 bits can hold 8 numbers. Now these are all positive numbers. So a total of 8 positive numbers can be held in 3 bits.

2**3 = 8

So 12 bits could hold 2**16 positive numbers.

0, 1, 2, 3,……, (2**16)-1.

(2**12)-1. = 65535

So the largest positive exponent we can have in 16 bits is 4095

I.e., 10**65535

What about negative exponents?

If there was a way we could signal that the number contained in the 16 bits is a negative number (and

yes, there is;
we'll see this way a bit later), then we could store the negative exponents

-65536, ……., -1

That means you can represent a number such as

395 x 10**(-100)

(Python will evaluate this as 3.9500000000000004e-98)

If the number is too small for Python to print, it will print 0.0

Notice this "trade-off" with accuracy, depending on how we split the 64 bits into the two parts that contain the mantissa and the exponent. We don't
have a choice. The computer hardware/software decides.

Reduce the 16 bits to 12 bits or 8 bits and the exponent range gets smaller, but the accuracy of the number gets larger because we have more bits on the other side to represent the mantissa (the

number of digits after the decimal point).

Increase the number of bits for the exponent to 20 (say), and the accuracy of the mantissa representation gets smaller, because fewer bits are left to hold it.

Final point:

To represent 5.5, we would put

0.55 in the mantissa bits and

1 in the exponent bits

So that we have the number (10**1) * 0.55

which is 5.5

Making sure the number after the decimal point is not 0 is called the “standard form”.

In other words, we would not use a representation like

(10**2)* 0.055, or

(10**3)*0.0055 etc.

even though these are mathematically correct. We want to avoid losing valuable bits to the 0's after the decimal point, since we have only 48 bits to hold that fraction.

Data Types

3, 99, 17537531.	Int
3.3, 99.76	float
"Gosh! He talks a lot!"	String (just a run of characters between quotes)

Algebraic expressions ALWAYS evaluate to a number:

```
>> 3*99
297
```

What kind of number (int, float) you get as a result depends on what the inputs are. You saw this in the lecture on using *, /, //, % operators.xpressions

BOOLEAN EXPRESSIONS:

An important class of expressions do not evaluate to numbers. They evaluate to
True or False.

These are called Boolean expressions. We use these to make a program take one of two or three or four or umpteen different directions depending on what a particular expression evaluates to. We use the word “conditional” to describe that expression.

This is not Python code. Just demonstrating the idea.

```
If (temperature > 90F) then
    print(“Dress like Tarzan!”)
Else
    If (temperature < 42F) then
        print(“Dress like an Eskimo!”)
    Else
        print(“Dress normally!”)
```

Expressions such as “temperature > 90f” have only one of two values: True or False.
They are Boolean expressions because a Boolean

variable can take on only one of two values: True or False.

Python understands Booleans.

```
>> 5 < 6
True

>> 6 < 5
False

>> 600 + 500 > 1200
False

>> 3 == 3
True

>> 3 == 714
False

>> 5 + 6 + 7 == 7 + 6 + 5
True
```

When the expression evaluates to “True”, the indented block under the “if” is executed.
When the expression evaluates to “False”, the indented block under the “else”

Is executed.

So this is called an “if-else” statement, or an “if-then-else” statement. It makes your program go one way or another way depending on the conditions.

Built-in functions:

```
>> abs (-5.5)
5.5

>> min(5,  6,  7)
5

>>max(-5, -6, -7)
-5
```

Functions that are not built-in must either come from existing libraries (e.g., like the sqrt() function that we import from the math module):

```
import math                #at the to of your Python file

>> x = math.sqrt(256)
```

```
>> x
16
```

or functions you must write yourself. Say your teacher catches you giggling loudly in class and makes you stay after class and write some string (e.g., “I will not giggle in class!”) 100 times on a sheet of paper. Being clever you learn Python and write a function to do this:

```
def  me_smart(s, n):     #s is the string to be written (printed) n times
    for k in range(n):      #using k as index in a loop going from 0,1,2 …. to n (stop at n-1)
        print(s)
return
```

```
>> me_smart(“I will not giggle in class!”, 100)
```

will print the string 100 times.

Type Conversions

```
>>  5 + 6       #algebraic expression
11
```

>> (5 < 6) #logical expression …. evaluates to True or False. (Boolean)

Important:

True is treated as the integer 1

False is treated as the integer 0

So, if in the shell, you type:

>> 10 + True
11 #Python does IMPLICIT TYPE CONVERSION because of the “+”
  #Converts True to the int 1 and does integer addition

>>10 + False #False is converted to 0 because of the “+”
10

This means in the SET of integers (all the whole numbers), the Boolean values

True and
False

are a SUBSET because they are 1 and 0.

The SET of integers are a subset of the FLOATS (because any integer becomes a float if you place a decimal point after it).

>> 4 + .4545.   # 4 is int, but Python converts it to float before addition
4.4545

The set of ints is contained in the set of floats. The latter is a larger set.

BUT, the range of values that int objects can have is much larger than the range of values that float objects can have.

That means you cannot always convert an int value to a float value (because the number of digits is too large and
causes OVERFLOW)

>>> 2**10000 + 5
19950631168807583848837421626835850838

8234968318861924548520089498529438830
2219466319199616840361945978993311294 2
3209124271556491349413781117593785932 0
9632395785573004679379452676524655126
6059895520550086918193311542508608460
6181046855090748660896248880904898 94
8380092539416332578506215683094739025
5691238806522509664387444104675987162
6985453222868538161694315775629640762
8368807607322285350916414761839563814
5896946389941084096053626782106462142
7333394036525565649530603142680234 96
9400335934316651459297773279665775606
1725820314079941981796073782456837622 8
0037302885487251900834464581454650557
9296014148339216157345881392570953797 6
9119277800826957735674444123062018757 8
3632550272832378927071037380286639303
1428133241401624195671690574061419654 3
4232463880124885614730520743199225961 1
7962501309928602417083408076059323201
6126849228849625584131284406153673895
1487114256315111089745514203313820202 93
1640957596464756010405845841566072044
9628670165150619206310041864222759086 7
0900574606417856951911456055068251250
4060075198422618980592371180544447880

72906395242548339221982707404473162376760846613033778706039803413197133493654622700563169937455508241780972810983291314403571877524768509857276937926433221599399876886660808368837838027643282775172273657572744784112294389733810861607423253291974813120197604178281965697475898164531258434135959862784130128185406283476649088690521047580882615823961985770122407044330583075869039319604603404973156583208672105913300903752823415539745394397715257455290510212310947321610753474825740775273989834829849834075693795564663862187456949927901657210370136443313581721431179139822298384584733440270964182851005072927748364550578634501100852987812389473928699540834346158807043959118985815145779177143619698728131459483783202082147498217185801138907122825090582681743622057747592141765371568772561490458290499246102863008153558330813010198767585623434353895540917562340084488752616264356864883351946372037729324009445624692325435040067802727383775537640672689863624103749141096671855705075909810024678988017827192595338

128242195402830275940844895501467666838969799688624163631337639390337345580140763674187771105538422573949911018646821969658165148513049422236994771476306915546821768287620036277725772378136533161119681128079266948188720129864366076855163986053460229787155751794738524636944692308789426594821700805112032236549628816903573912136833839359175641873385051097027161391543959099159815465441733631165693603112224993796999922678173235802311186264457529913575817500819983923628461524988108896023224436217377161808635701546848405862232979285387562348655644053696262201896357102881236156751254333830327002909766865056855715750551672751889919412971133769014991618131517154400772865057318955745092033018530484711381831540732405331903846208403642176370391155063978900074285367219628090347797453332046836879586858023795221862912008074281955131794815762444829851846150970488802727472157468813159475040973211508049819045580341682694978714131606321068639151168177430479259670938 1

So no problem representing that integer. What if we want a float?

```
>>> 2**10000 + 5.0
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    2**10000 + 5.0
OverflowError: int too large to convert to float
```

EXPLICIT type-conversion is the conversion you do by yourself:

```
>> int (5.6)
5

>> int (-5.9)
-5

>> float(7)
7.0

>> int(“5.9”).   #makes no sense, so Python will give you a ValueError
```

There is also a str() function (actually a “string constructor”, just like int() is the “int constructor”) that converts an int into a string

```
>> x = str(5.9)
>> x
‘5.9’                    #x is the string “5.9”
```
