Lecture 2

Variables and Expressions (and Printing); Assignments and Statements; Basic Numerical Operators; `float` Weirdness and Binary; Functions, Randomness and Modules; String Basics

1. Variables and Expressions (and Printing)

A ***variable*** is a *name* that is associated with a *value*. (Warning: this is actually a simplification. But it wouldn't be helpful to describe the truth right now.)

The name part of a variable is called the variable's ***identifier***. Rules for Python identifiers:

- Only letters, digits, and underscores (_), no spaces or other punctuation
- May not begin with a digit
- May not be a reserved word (`if`, `import`, `def`, etc.)

So, these are legal: `hey123`, `AHHH`, `_x_y_z`

And these are not: `1st_thing`, `what?`, `no spaces`

Also:

- Identifiers are case-sensitive: `hello` is not the same as `hElLo`
- Probably should be *meaningful*
- We'll start variable names with lowercase letters usually

The purpose of variables is to hold data values for reference later on. This "obvious" point takes a while to click for most beginners. Variables aren't just things you throw in your program because whatever; their purpose is to remember data that you'll need later on.

To *create* a variable: simply write a line assigning it a value, of the following form:

In [ ]:
```
VARIABLE CREATION SYNTAX (do not run me):

<variable name> = <value>
```

After that, the variable name will be associated with that value, until it is assigned a different value.

(Note: In these notes, I will use `<this notation>` to represent a blank to be filled in. I use this to highlight ***syntax***: this refers to the order in which variables and operators are placed to make meaningful statements. For example, `x = 17` and `17 = x` are very different statements -- the latter causes a *syntax error*, because that's not the order you're supposed to write things in -- we'll discuss why soon.)

An ***expression*** is one or more literals or variables, connected with **operators** (e.g., `+`, `-`, etc.) and functions, which can be ***evaluated*** to produce a value. For example:

```
3 + 4.5
"Hello" + "Goodbye"
len("Hello")
"Hello" * 3
```

When these are encountered in a program, the interpreter respectively computes values of `7.5`, `"HelloGoodbye"`, `5`, and `"HelloHelloHello"` for the first four. (Notice that "value" doesn't mean *number* -- it means any type of data, and so strings are perfectly good end values.)

These are ***not*** (just) expressions:

```
x = 5 + 7
print(5+7)
```

Sure, `5+7` on its own is an expression -- it has a value. But `x = 5 + 7` and `print(5+7)` are really *instructions* about what to do with values, rather that *values* themselves.

Also, `3 + xyz` is also a (probably) valid expression. Since the `xyz` is **not in quotes, it is treated as a variable!**

To evaluate this expression, the interpreter reads the *current* value of the variable `xyz`, and (assuming it's numerical) produces that value plus 3. But: if

- `xyz` has a non-numeric value, or
- `xyz` has not previously been assigned a value

then this will cause an error!

Let's talk about the very basics of printing, which you may have caught on to.

In [ ]:
```
SYNTAX FOR THE PRINT FUNCTION:

print(<whatever>, <another>, <yet another>, <as many of these as you like>)
```

Each of those arguments (`<whatever>`, `<another>`, etc.) must be a single expression -- either a single literal, or a single variable, or a more complex formula. When your program reaches the `print` line, each of the expressions will be printed out on the same line, with their current values at that line, in order, with spaces between each value. The next `print` statement will be printed starting on a new line.

In [ ]:
```
# EXAMPLE 1a: Basic print

variable_x = 4 + 5
print("Hello", True, variable_x, 5*variable_x  - (variable_x+2), "Goodbye")
print("Next line")
# That looks like two expressions to me, so there's got to be a comma between them!
# If you put a comma in between, then this code will run.
print("Here's a problem" variable_x)
```

You can also suppress the automatic newline at the end of a print by including `end = ""` at the end of the print statement, as follows.

In [ ]:
```
# EXAMPLE 1b: Suppressing the automatic newline

print(10)
print(11)
print(12, end = "") # This print statement won't be followed by an automatic newlin
e...
print(13)           # ...which means that this is displayed IMMEDIATELY after the 1
2, not even a space in between.
print(14)
```

2. Assignment and Statements

The single equals sign = is called the ***assignment operator***. We've already seen it with variable creation, but it works more generally:

In [ ]:
```
ASSIGNMENT SYNTAX:

<variable name> = <expression>
```

(Note the order!!) When you write this line, three things happen:

1. ***FIRST***, the right side gets evaluated.
2. ***THEN***, the value you gets stored to the variable `<variable name>`
3. Until `<variable name>` gets reassigned, all subsequent references to it will produce that value.

Understanding assignment isn't too hard, but it is utterly utterly crucial to understanding programming. Read those 3 lines carefully.

Also, be aware that we use the equals sign in (at least) two different ways in mathematics: in addition to assignment (as in "set $x = 3$, and then tell me what x^2 is"), there is also the logical operator (as in "is 2 + 2 = 5? No, that statement is False"). The first one is a command to do something, the second one expresses a potential fact; pay attention to the distinction.

If I write

```
x = 4 + 2
```

1. 4+2 evaluates to 6.
2. The value 6 is assigned to `x`.
3. If I saw, e.g., `print(x+4)`, then `10` would be displayed.

But, imagine if I wrote

```
4 + 2 = x
```

1. x evaluates to... I'm not sure. Maybe it has a value already? Huh.
2. Whatever that value is gets assigned to 4+2...what? The program is trying to change the value of 4+2? That's nonsense!

So, order matters deeply here.

Variables can change, which means that you have to pay attention to assignments really closely to predict program behavior. Look at the following example. Try to figure out what happens in this program; then, when you've got an idea, add print statements to the bottom to confirm. (Remember: lines execute in order, and on any line, a variable has the value of its most recent assignment, and the right side of an assignment is evaluated **first**.)

In [ ]:
```
# EXAMPLE 2a: Walkthrough

x = 0
y = 1
z = 2
a = 3
y = x + 10
z = z + y
z - 2
x = 3 * x # DON'T WRITE "3x" -- actually put the "*" explicitly.
a = "x"

# When you're ready, print out x,y,z,a.
```

Notice how Line 9 contains no assignment, and so it does nothing! *Simple, but important, point*: expressions are dandy, but if you don't print them or assign them to a variable, then they amount to nothing but a waste of processor time.

Notice also that Line 11 changes `a` to the `str` "x", not the variable `x`. In fact, notice also that `a` changes type, from an `int` to a `str`! That's fine in Python (if somewhat ill-advised), although many other languages would not allow such sheninanigans.

Finally, notice that `y` is assigned to be `x+10` on line 7, but `y` doesn't get updated when `x` changes later on line 10!

You've noticed by now that a Python program is typically a list of lines. Some of those lines begin with `#`'s -- those are ***comments***, which give human readers clues to how the nearby code works. Each one of the non-comment lines is called a ***statement***. You should type each statement on a single line, *with no leading spaces*. It is common to put exactly one space between each variable, operator, and/or literal appearing in your statement, although you have some freedom in doing this.

When a program runs, it performs the *actions* specified by those statements, in order; we say that the program ***executes*** those statements. You should think of a statement as an instruction, and executing a statement as the program trying to follow the instruction.

As we've just noted: a statement that doesn't either change the value of a variable or print something out is legal, but probably isn't very useful! A statement that is just an expression without a print or an assignment is like going up to your brother and just saying "the dog" and nothing else. If you want your brother to take the dog out for a walk, you can't just give him a noun -- you need to specify to him what you want done with that noun.

Let's make a slightly less dumb (still pretty dumb) program.

Create four variables:

- a variable called `name`, which you should assign to be your name
- two variables called `score1` and `score2`, which you should assign to be `90` and `100`, respectively
- a variable called `average`, which should be assigned the average of the two scores. (Have Python do the computation, even though we all know the average is 95.)

Then, using these variables, have the program print out:

```
Hi <name variable here>
Your average is: <average variable here>
```

You should type your name, 90 and 100 each only once, and never type 95.

In [ ]:
```
# EXAMPLE 2b: Average calculator
# See instructions above.

name = "Evan"
score1 = 90
score2 = 100
average = (score1 + score2)/2
print("Hi", name)
print("Your average is:", average)
```

Notice the parentheses in `(score1 + score2)/2`. That's necessary because of order of operations.

3. Basic Numerical Operators

This is a math class, so let's talk about math!

With `ints` and `floats`, the basic operators are `+`, `-`, `*`, `/`, and `**` (the last one being exponentiation). The first two can be used with a single number, to indicate positive or negative, and all of them can be used with two arguments (which is a fancy word for "inputs"). They mostly work the way you'd expect, but there are some things to be aware of.

- Obviously, you can't divide by 0.
- `floats` can be weird. I'll elaborate shortly.
- If you add or subtract or multiply two `ints`, Python will produce an `int` value as the result. If you divide two `ints`, or perform any operation involving at least one `float`, Python will *always* provide a `float` answer -- even if you divide 12 by 6, the result will be a `float`.
- Python has built-in support for complex numbers, but it uses the letter `j` for $\sqrt{-1}$ instead of `i`.

In [ ]:
```
# EXAMPLE 3a: Arithmetic
# What do these evaluate to?

print(5 + 2)     # Notice the output types for these four lines:
print(6.0 * 4)   # if it has a decimal point, Python's producing a float,
print(12 / 6)    # if not, almost certainly Python's produced an int.
print(4 * 3 + 2) #

print(2 ** 10)     #
print(2 ** 0.5)    # Exponents
print((-2) ** 0.5) #

print(3 / 0)        # I think you know this is an error
```

For `ints`, there are two other operations worth knowing about: mod `%` and floored division `//`.

The first one gives *remainders*. That is, `x % y` gives the remainder when `x` is divided by `y`.

The second one gives *integer part of division*. That is, `x // y` divides `x` by `y`, but simply removes the decimal part.

In [ ]:
```
# EXAMPLE 3b: % and //
# What are the answers?

print(29//7)
print(29 % 7)
print(123 % 10)

# Check this one out: what am I doing?
x = 1896
print( (x//100) % 10 )
```

Having introduced all these operators, let's now talk about operator precedence. Imagine you see an expression like

`1 + 2 * 4 % 5 - 1` or `5 - 3 - 2 ** 2`

The computer can only do one operation at a time, which means it needs to choose an order to do them in. How does it choose?

The basic rules are:

1. `**` is evaluated before {`*`, `/`, `//`, `%`}, which are evaluated before {`+`, `-`}.
2. Within each class, appearances of any of the symbols are evaluated as they are encountered in the expression, from left to right.

So, in the first expression, what happens? What about in the second?

First one:

`1 + 2 * 4 % 5 - 1` --> `1 + 8 % 5 - 1` --> `1 + 3 - 1` --> `4 - 1` --> `3`

Second one:

`5 - 3 - 2 ** 2` --> `5 - 3 - 4` --> `2 - 4` --> `-2`

Of course, parentheses can override these rules, and obviously should be used when there is any doubt:

`10 - (2 - 1)` --> `10 - 1` --> `9`

In [ ]:
```
# EXAMPLE 3c: Precedence

print(1 + 2 * 4 % 5 - 1)
print(5 - 3 - 2 ** 2)
print(10 - (2 - 1))
```

4. `float` Weirdness, Binary

I said that `floats` can be "weird". Here's what I mean.

In [ ]:
```
# EXAMPLE 4a: Floats misbehaving

print(0.1 + 0.2) # Wait, seriously?

print(2 ** 2000)     # A really big int
print(2.0 ** 2000)   # A really big float
```

So, the two basic "weird" things to be aware of when it comes to `floats` are that they are imprecise, and that they cannot express arbitrarily large numbers -- numbers greater that about 10^{300} in absolute value can't be stored as `floats`. (On the off chance that you need to deal with numbers that large, there are special packages that can assist you).

The reason why both these phenomena happen, in short, is that `floats` are stored using a fixed number of binary digits.

Pause: I think I have to explain binary now. All data on a computer is stored using binary in one way or another, but let's stick with numbers (both `floats` and `ints`), for now.

Humans use the decimal system for writing our numbers. Everything is based around the number ten: ten digits, powers of ten. We use the decimal system because ... why?

Anyhow, when I write down a number like 307.4, each of those four digits stands for something. The "3" stands for 3 hundreds, the "0" stands for 0 tens, the "7" stands for 7 ones, and the "4" stands for 4 tenths. Each digit counts the number of a quantity that is a power of 10; which power of 10 depends on the position of the digit from the decimal point. Oh, and of course, you add those quantities together: $3 \times 10^2 + 0 \times 10^1 + 7 \times 10^0 + 4 \times 10^{-1}$ is the quantity we mean when we write down 307.4.

Now, computers have to store data -- each digit of each number it needs to compute with -- using tiny physical "switches". (What exactly this "switch" looks like in physical terms depends on the type of memory we are talking about.) It's hard to engineer a tiny switch that can have ten different positions; it's much easier to engineer a switch that has two positions. So, computers don't use base ten; they use base two -- binary.

The binary system works in exactly the same way as decimal, except the only digits allowed are 0 and 1, and all the quantities are powers of 2.

So, for example,

111001 (binary)

is the decimal number $1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 32 + 16 + 8 + 0 + 0 + 1 = 57$.

What would 1010101 (binary) be? What would 110.101 (binary) be?

The latter would be $1 \times 4 + 1 \times 2 + 0 \times 1 + 1 \times \frac{1}{2} + 0 \times \frac{1}{4} + 1 \times \frac{1}{8} = 6.625$.

Now, `ints` are stored in the computer exactly, as finite sequences of "0"s and "1"s (where "0" would be represented by a down switch, and "1" would be represented by an up switch). And if you do arithmetic with `ints`, you will always get exactly correct answers. In fact, that's the primary reason for the existence of `ints`: if you need an answer that you can be confident is *exactly* right -- often you do! -- you can have that with `int` arithmetic.

`floats`, on the other hand, are harder to store, because numbers can have infinitely many digits to the right of the decimal point. Furthermore, it turns out that many numbers we would think are very tame become infinite decimals in binary. For example:

0.3 (decimal) $= 0.0100110011001100110011001100110011 \ldots$ (binary)

A finite computer can't store an infinite number of bits, and so it cuts off after a finite number of places -- leading to weird rounding errors.

Technically, a `float` is stored as an integer (in binary), together with a power of 2 to multiply it by (also stored in binary). An analogue in base 10 would be: 1234.56 can be written as 123456×10^{-2}, so 123456 and -2 would be stored. Only a fixed number of bits are allowed for each part: if the exponent is too large to fit in its allotment, you get an error; if there are too many significant digits, the least significant ones just get rounded off. This explains both of the oddities we saw before.

Don't get too hung up on this -- these rounding errors usually show up after more than 10 significant (decimal) digits, and so they rarely affect practical calculations. The biggest thing you need to watch out for is *comparisons*. I think you'd agree that

In [ ]:
```
# EXAMPLE 4b: Comparisons

print(1 + 2 <= 3)       # Would you say that 1 + 2 <= 3 is a True statement?
print(0.1 + 0.2 <= 0.3) # Would you say that 0.1 + 0.2 <= 0.3 is a True statement?
```

5. Functions, Modules, and Randomness

In general, a ***function*** in Python is any expression that has the general form

In [ ]:
```
FUNCTION SYNTAX:

<fn name>()

or

<fn name>(  <inputs>  )
```

So a function is something that you write with parentheses, that often (not always) has inputs, and which produces some action or value. (Cripes, what a horrible definition. Let's stick with it anyway.)

`print()` is a function: you give it a `str` or a number, and it displays it on the screen.

`len()` is a function: you give it a `str`, and it produces a value: the length of that `str`.

And of course, your favorite math functions are Python functions too! They look slightly different in Python than they do in Calculus class, but I'm sure you can get the gist easily. But: to use these functions, be sure to **import the `math` module**. See below. (For other math functions, use a textbook or Google!)

In [ ]:
```
# EXAMPLE 5a: The math module

# The following line IMPORTS A MODULE: the math module, which contains
# functions beyond basic arithmetic

import math

# Now you can use functions and variables in the math module; just write math.
# in front of whatever you need.

y = 25.0
z = (3.14159)/6 # This is a crappy way of writing PI/6 -- see below.

# math.exp() is the function e^x
print(math.exp(1))
# math.sqrt() is the square root function
print(math.sqrt(y))
# I think you can guess this one
print(math.sin(z))

# math.pi isn't a function -- it's an imported variable.
# Indeed, notice the lack of parentheses.
print(math.pi)
```

Computers are great at following directions, but they are suprisingly bad at picking random numbers. Fortunately, the creators of Python have made tools so that Python can do a pretty convincing job at faking it. To use random numbers, we'll have to import another module: the `random` module.

For now, we'll use two basic ways to generate random numbers:

- `random.random()` will return a random float between 0 and 1. Each float has a roughly equal chance of being chosen.
- `random.randrange(<int1>, <int2>)` will return a random integer that is $\geq$ `<int1>` but less than `<int2>`. Pay attention to the boundaries: `<int1>` can be chosen, but `<int2>` cannot!

In [ ]:
```
# EXAMPLE 5b: Random numbers

import random

# Every time we run this, we'll get different results, but all 3 numbers should be
between 0 and 1.
x = random.random()
y = random.random()
z = random.random()

print(x,y,z)

# Here are some random "dice rolls" -- numbers between 1 and 6 (inclusive).

roll_1 = random.randrange(1,7)
roll_2 = random.randrange(1,7)
roll_3 = random.randrange(1,7)

print(roll_1, roll_2, roll_3)
```

6. String Basics

Basic string operations include +, `len()`, and * as we discussed before, and also the *index operator*, represented by a pair of square brackets. It works as follows: suppose that we write the lines

```
x = "Hi!  Ho? "
print(x[0])
print(x[1])
print(x[2])
print(x[6])
```

`x` is a `str` containing 9 characters (note the 2 spaces in the middle and the one at the end). Then `x[0]` will produce the character `"H"`, while `x[1]` will produce the character `"i"`, and `x[2]` will produce the char `"!"`. And `x[6]` will produce the seventh character (counting spaces), which is `"o"`. Notice the **zero-based indexing**: the first character is labelled 0 rather than 1. This is for historical reasons, but most major programming languages use it, and Reddit will make fun of you if you try to start counting with 1.

Also be aware that if you see numbers in quotation marks, they are being treated by the program as sequences of symbols, rather than numbers. See the next example.

In [ ]:
```
# EXAMPLE 6a: Strings
a = "Hello"
b = "Goodbye"
print(a+b)
print(len(a+b))
print(3*a)

x = "Hi!  Ho? "
print(x[0], x[1], x[2], x[6])
print(x[-2]) # Here's a bonus tip -- if you put in a negative index, it counts from
the end (but not zero-based)

# Watch out here! It doesn't matter that us humans see numbers --
# if they are in quotes, they are treated as strings, NOT numbers.
num1 = "10"
num2 = "20"
print(num1 + num2)
print(num1 - num2)
```

Imagine that I wanted to store the following literal in a single string variable:

`Evan said "3300 is fun" and we nodded politely.`

Do you see why that could be tricky?

In [ ]:
```
# EXAMPLE 6b: A troublesome string literal

# Look at the coloring of the code below -- it gives a big hint
# about what the problem is.
sentence = "Evan said "3300 is fun" and we nodded politely."
print(sentence)
```

The quotes, which we want to be characters within the string, end up accidentally ending the string!

There are a few characters that, for one reason or another, are tricky to type. To type these, Python (and many languages) have the so-called ***escape characters***. They are all typed as a backslash(\) and one more keystroke; and they are used to represent these tricky characters. Here are a few:

- \n: is the newline character. If this is included in a string literal, then when that literal is printed, the slash and n will not be displayed -- in their place, you will see your output with part of it on a new line (the part after the backslash n).
- \": if you want to see the quote mark character, you can use \", and then that quote won't be interpreted as the end of the string.
- \\: what if you actually truly want to see the backslash character? You use two backslashes.

In [ ]:
```
# EXAMPLE 6c: Escape characters

sentence = "Evan said \"escape sequences are helpful\" and we nodded knowingly."
print(sentence)

forget = "\nHe also said... \n\n ...actually, I can't remember what else he said.\n
"
print(forget)

slash = "I guess I wasn't \\\\listening//."
print(slash)
```

If you surround your string literals with pairs of double-quotes (`"my usual"`), they have to be typed on a single line of code. But there is an alternative: you can surround them with **triple** double quotes (`"""Like this"""`) -- in this case, your literal is allowed to span many lines. (This is also a trick used to temporarily disable big chunks of code without deleting them.)

In [ ]:
```
# EXAMPLE 6d: Multi-line literals

# This works
multi_line = """This multi-line literal is legal.
(The syntax coloring seems to back that up.)
That's because I started the literal with a triple quote."""
print(multi_line)


# This one won't work.
bad = "Let me just
use single quotes
for a multi-line literal
and see what happens"

print(bad)
```

By the way -- we learned how the computer stores numbers, but how does it store strings? Simplified answer: there is a special, widely used system called ASCII (American Standard Code for Information Interchange), which associates a number to each printable character. For example, `a` is 97, `b` is 98, `A` is 65, `0` (the symbol) is 49, `!` is 33, etc. So, characters can be represented by integers, which we know how the computer stores. So strings can also be stored in binary -- and indeed, all data stored by a computer is encoded in 0's and 1's somehow.