Topic 2: Conditionals, Operators, and User Input Cont'd.

Goals: By the end of this topic, we will discuss…

- *conditional (i.e. "if" statements) and control flow*
- *number representations and how computers store data (booleans, numbers)*
- *formatting print statements*

Acknowledgements: These class notes build on the content of my previous courses as well as the work of R. Jordan Crouser, Jeffrey S. Castrucci, and Dominique F. Thiébaut.

Conditional

[http://sites.psu.edu/mgeppingerpassionblog/wp-content/uploads/sites/32731/2015/09/roads-diverging.jpg]

The Road Not Taken b*y Robert Frost*

Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;

Then took the other, as just as fair,
And having perhaps the better claim,
Because it was grassy and wanted wear;
Though as for that the passing there
Had worn them really about the same,

And both that morning equally lay
In leaves no step had trodden black.
Oh, I kept the first for another day!
Yet knowing how way leads on to way,
I doubted if I should ever come back.

I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I—
I took the one less traveled by,
And that has made all the difference.

"Whichever way they go, they're sure to miss something good on the other path."
[Miles, Jonathan (May 11, 2008). "All the Difference". New York Times. Retrieved June 13, 2015.]

Which road did he take? Why did he choose this road?

If you want to walk on a grassy cover then walk down the second road, otherwise take the first.

If (desire == grassy cover) then second path else first.

```
if grassy:
     print("Take path 2.")
else:
     print("Take path 1.")
```

Cooking Example:

"It is easy to check a cake for doneness by using the toothpick test. Just stick a toothpick into the center of your cake or cupcake. If there is wet batter still on the toothpick, it needs more time in the oven. If it comes out clean, your cake is done!"

[https://www.youtube.com/watch?v=x5GMCIF9en0]

If batter on toothpick then more time in oven else remove cake.

```
if batter_on_toothpick:
     print("Leave cake in oven.")
else:
     print("Remove cake from oven.")
```

If toothpick comes out clean then remove from oven else leave in for a few more minutes.

```
if no_batter_on_toothpick:
     print("Remove cake from oven.")
else:
     print("Leave cake in oven.")
```

What are `grassy, batter_on_toothpick, no_batter_on_toothpick` ???

Answer: *Boolean Expressions*

Booleans

- In Boolean Logic, every statement is either True or False. There are no other possibilities.
- You may have encountered this sort of logic in math class or philosophy class.
- To accommodate these situations,
  - Python has a type `Bool`,
  - which only has two possible values: `True` or `False`.

The Road Not Taken

```
grassy = True
if grassy:
     print("Take path 2.")
else:
     print("Take path 1.")
```

Lb to Kg Example

```
lb_TO_kg = False
if lb_TO_kg:
    lb = eval(input("Enter the weight in lbs: "))
    kg = lb / 2.2
    print(lb, " lbs is ", kg, " kgs!")
else:
    kg = eval(input("Enter the weight in kgs: "))
    lb = kg * 2.2
    print(kg, " kgs is ", lb, " lbs!")
```

Remind: Data Types

- `bool`: boolean

  Two possible values: `True` or `False`.
- `int`: integer

  Whole Numbers: For example: 3, 4, 894, 0, -3, -18
- `float`: floating point number (an approximation to a real number)

  Decimal Numbers: For example: 5.6, 7.342, 53452.0, 0.0, -89.34, -9.5

Conditional Statements

Example:

1. I passed the course. (No Condition)
2. If I do well on the exam then I will pass the course.
3. If I do well on the exam then I will pass the course, otherwise I will fail.

Conditional Statements or "If" statements allow a program to select a particular execution path from a set of alternatives.

Statement: '`if`'

`if` statements can be used to **control** which instructions are executed by **creating a "branch" in the code**. The `if` statement evaluates a Boolean expression, and **if it is True**, then it runs the code under it, otherwise it skips it.

The general form of an `if` statement in Python is as follows:

```
if condition:       # general form
     block
```

`if` statements are always followed by a colon (:), this is how Python knows you are going to create a new block of code. Indenting four spaces (or tab) tells Python which lines of code are in that block. **You must indent a block!**

Examples:

```
if (today is a weekday):
     go to class

if (today is Tuesday):
     then it will rain
```

Statement: '`if - else`'

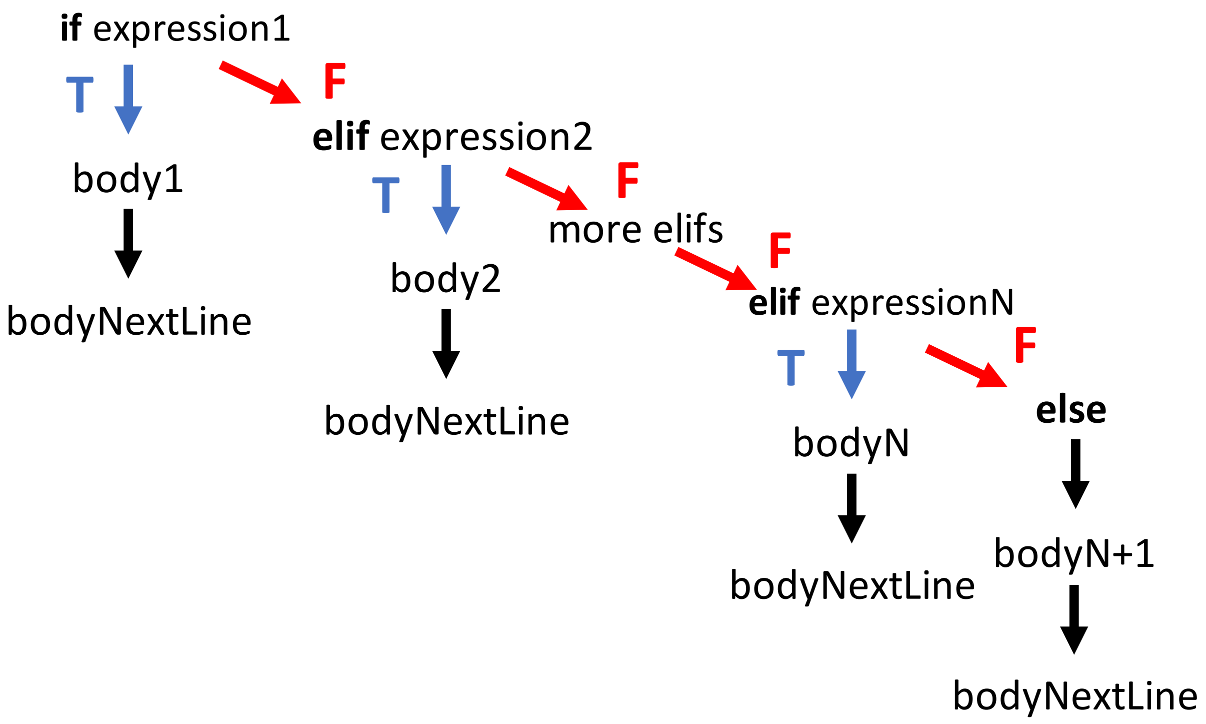
You may want to complete a different task if the condition is false. `else` statements occur when no `if` condition is satisfied.

```
if the conditional is true
     perform action(s) following conditional
otherwise (else) the conditional must be false
     perform action(s) following the else statement
```

The general form of an `if` statement in Python is as follows:

```
if condition:
     block1                   # when condition is True
else:
     block2                   # when condition is False
```

Again both blocks must be indented (four spaces or tab) to tell Python which lines of code are in the block.



Examples:

```
if (today is a weekday):
     go to class
else:                         # (today is a weekend)
     sleep in

if (food at Tyler looks good):
     eat at Tyler
else:                         # food at Tyler doesn't look good
     order Domino's
```

Statement: ‘`if - elif`’

`elif` stands for "else if", so this forms a chain of conditions. To execute an `if` statement, evaluate each expression in order from top to bottom. If an expression produces `True`, execute the body of that clause and then skip the rest. If there is an `else`, and none of the expressions produce `True`, then execute the body of the else.

The general form of an `if - elif` statement in Python is as follows:

```
if condition1:
     block1                   # when condition1 is True
elif condition2:
     block2                   # when condition2 is True
```

`if - elif` statements can be combined with `else` statements:

```
if condition1:
     block1                   # when condition1 is True
elif condition2:
     block2                   # when condition2 is True
else:
     block3                   # when both conditions are False
```

Statement: General ‘`if - elif - else`’

The most general form for `if` statements is shown below:

```
if expression1:
    body1
elif expression2:
    body2
      .
      .
      .
elif expressionN:
    bodyN
else:
    bodyN+1

bodyNextLine
```

IN OTHER WORDS:
For an If structure, *at most one* block will be executed
For an If/Else structure, *one or the other* block will be executed
For an If/Elif structure, *at most one* block will be executed
For an If/Elif/Else structure, *exactly one* block will be executed
The order of your test expressions matters

Activity: What is the difference between each code snippet.

```
if (it is sunny):
     go to the beach
if (it is snowy):
     go skiing
else:
     stay home
```

```
if (it is sunny):
     go to the beach
elif (it is snowy):
     go skiing
else:
     stay home
```

```
if (it is sunny):
     go to the beach
     if (it is snowy):
          go skiing
     else:
          stay home
```

```
if (it is snowy):
     go skiing
elif (it is sunny):
     go to the beach
else:
     stay home
```

In each case,
- What will happen if it is both sunny and snowy?
- What will happen if it is neither sunny or snowy?
- What will happen if it is sunny but not snowy?
- What will happen if it is snowy but not sunny?

Nested `if` Statements

It is possible to place an if statement within the body of another if statement. For example:

```
if precipitation:
    if warm_temperature:
        print('Bring your umbrella!')
    else:
        print('Wear your snow boots and winter coat!)
```

The message `'Bring your umbrella!'` is printed only when both of the `if` statement conditions are True. The message `'Wear your snow boots and winter coat!'` is printed only when the outer if condition is `True`, but the inner if condition is `False`. The following is equivalent to the code above:

```
if precipitation and warm_temperature:
     print('Bring your umbrella')
elif precipitation:  # and not warm_temperature
     print('Wear your snow boots and winter coat!')
elif sun_out:
     print('Wear your hat and sun screen!')
else:
     print('No specific extreme weather gear required.’)
```

But this is a bit ugly because you are testing the value of precipitation twice. It is clearer to write this as a nested-ifs.

```
if precipitation:
     if warm_temperature:
          print('Bring your umbrella')
     else:
          print('Wear your snow boots and winter coat!')
elif sun_out:
     print('Wear your hat and sun screen!')
else:
     print('No specific extreme weather gear required.')
```

How do I know which ‘`if`’ to use?

In a group, discuss how you know when to use if, elif, and else? What metaphor(s) help you think through this?

Mathematical Operators Con’t

Logical Operators

Reminder: In Boolean Logic, every statement is either `True` or `False`.

The real power of Booleans becomes apparent when they are used together with comparison operators. Three logical operators are particularly important for our purposes:
`and` - The `and` operator evaluates to True if and only if both of its operands are True.
`or` - The `or` operator evaluates to True if one (or both) of its operands are True.
`not` - The `not` operator evaluates to True if and only if the operand is False.

x	**y**	**x and y**	**x or y**	**not x**
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

The order of precedence for logical operators is: not, and, then or.

Numerical Operator (Comparing Values)

These operators take two numerical values and produce a boolean value.

Description	**Operator**	**Example**	**Result of example**
less than	<	3 < 4	True
greater than	>	3 > 4	False
equal to	==	3 == 4	False
greater than or equal to	>=	3 >= 4	False
less than or equal to	<=	3 <= 4	True
not equal to	!=	3 != 4	True

Note the difference between = and ==. = is assignment whereas == is comparison.

Examples using comparisons operators:

```
>>> 1 == 1
True
>>> 2 > 3
False
>>> 1!=1
False
>>> 2*3 == 5
False
>>> 2*3 !=5
True
```

Examples of comparisons on variables:

```
>>> x = 5
>>> y = 3
>>> x == y
False

>>> x > y
True
```

Activity: How old are you?

Write a program that ask the user for his/her/their name and their age in years and give one or more answers from the following ones below:

- if the age of the user is less than 16, the program should print on the screen “You are not allowed to drive at the moment”.
- if the age of the user is less than 18, the program should print on the screen “You are not allowed to vote at the moment”.
- if the age of the user is less than 25, the program should print on the screen “You are not allowed to rent a car at the moment”.
- if the age of the user is greater than or equal with 25, the program should print on the screen “You can do anything that is legal”.

[https://www.quora.com/Can-any-one-give-a-real-life-example-of-if-else-and-nested-if-else-or-any-others-like-switch-cases]

Boolean as Input

If you are trying to input a Boolean value, you might run into trouble.
Instead use a compactor with a character value.

```
val = input("Enter a boolean (\'t\' or \'f\')")
if (val == 't'):
     bool_val = True
else:
     bool_val = False
print(bool_val)
```

Boolean as Int (Advanced Topic)

`bool` is also a subtype of `int`, where `True == 1` and `False == 0`. What happens when you enter the following code?

```
>>> True + 1
2
>>> False - 1
-1
>>> (False + 3) * 4 - True
11
>>> bool('stuff')
True
```

You may run into hard to detect errors where you accidentally do an integer arithmetic operation on a boolean variable, changing its type to integer. Then when you pass that integer to a function expecting a boolean value, you will get the error at that line instead of the line where the type change occurred.

Recall:

Two kinds of numbers in CS:

- `int`: integer
  Whole Numbers: For example: 3, 4, 894, 0, -3, -18
- `float`: floating point number (an approximation to a real number)
  Decimal Numbers or Floating Point: For example: 5.6, 7.342, 53452.0, 0.0, -89.34, -9.5

Basic operators:

- addition: +
- subtraction: -
- multiplication: *
- division: /
- integer division: //
- exponentiation: ** (power)
- modular arithmetic: % (modulo)

Note: If one of the numbers is a float the result will be a float.

Build in Operators

• abs(x)	# return the absolute value of x
• float(x)	# return x *parsed* as a float
• int(x)	# return x *parsed* as an int
• max(…)	# return the largest of a list of numbers
• min(…)	# return the smallest of a list of numbers
• round(x[, n])	# return x rounded to n digits after the decimal point.
	# If n is omitted, it returns the nearest integer value
• sum(…)	# return the sum of a list of numbers

Aside: What does parsed mean? What does return mean?

Examples:

```
>>> abs(-3.5)
3.5
>>> float(5)
5.0
>>> max([5,8,2,3,5])
8
>>> min([5,8,2,3,5])
2
>>> round(23.4567,2)
23.46
>>> round(23.4567)
23
```

Foreshadowing “Functions”

```
def do_something():
    # perform some operations, like:
    x = 2 + 3

    # send stuff back to the main program
    return x

y = do_something()
print(y) # y = 5
```

The Math Module

Lots of other things we might want to do with numerical values are available as functions in the math module.

- In Python, modules are just files containing Python definitions and statements (ex. `name.py`)
- These can be imported using `import name`
- To access name’s functions, type `name.function()`

`import math` # this allows you to access the math functions.

- `math.floor(f)` # round float f down
- `math.ceil(f)` # round float f up
- `math.sqrt(x)` # take the square root of x

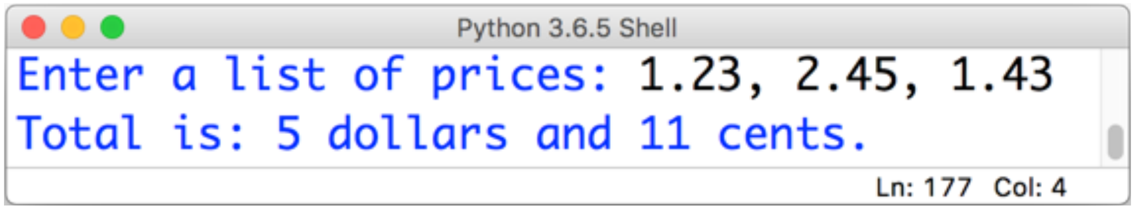
And more! Check out: https://docs.python.org/2/library/math.html

Examples:

```
>>> import math
>>> math.floor(5.6)
5
>>> math.ceil(5.4)
6
>>> math.sqrt(25)
5.0
```

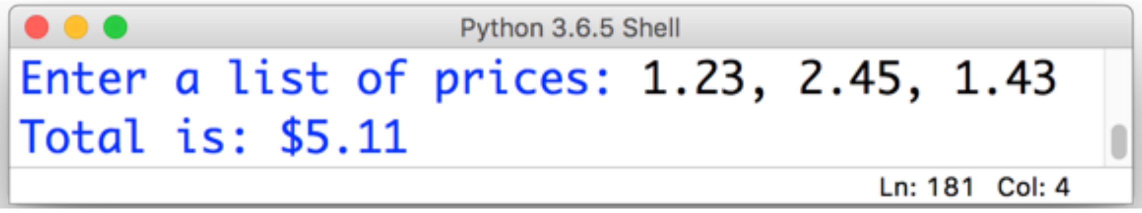
Exercise: Dollars and Cents - Picking up from Lab.

Use built-in functions and functions from the math module to take a list of prices, calculate their sum, and output their total formatted like this:



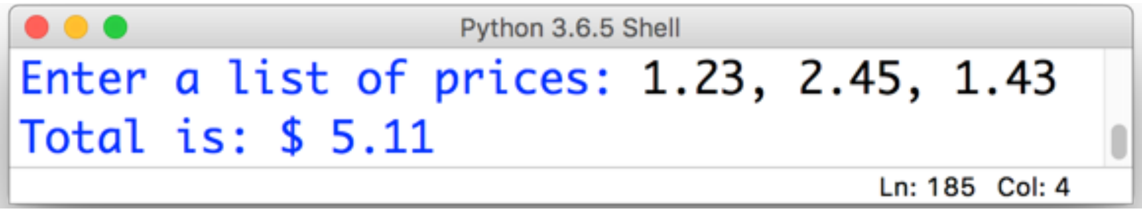
```
Enter a list of prices: 1.23, 2.45, 1.43
Total is: 5 dollars and 11 cents.
```

What if we want the output to look like this:



```
Enter a list of prices: 1.23, 2.45, 1.43
Total is: $5.11
```

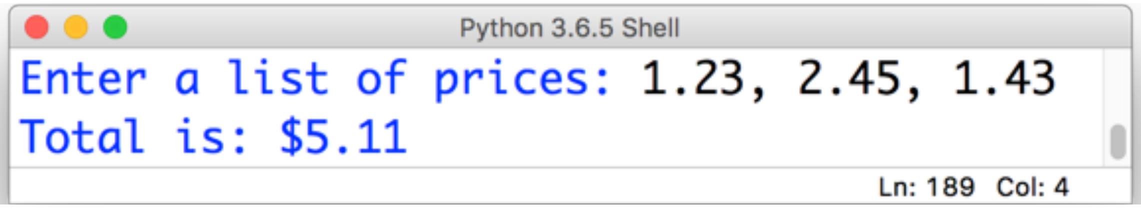
Attempt #1:
`print("Total is: $", my_sum))`



```
Enter a list of prices: 1.23, 2.45, 1.43
Total is: $ 5.11
```

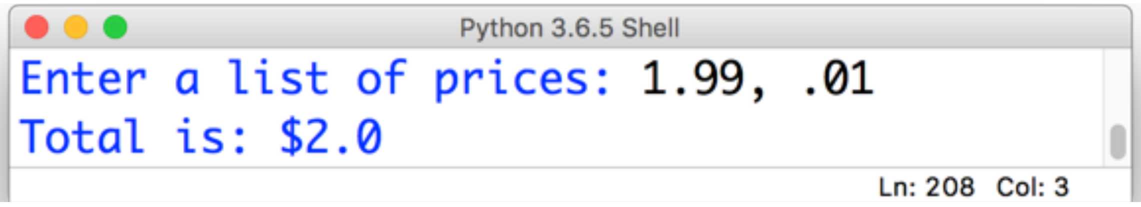
…. but we have an extra space.

Attempt #2:
`print("Total is: $" + str(my_sum)))`



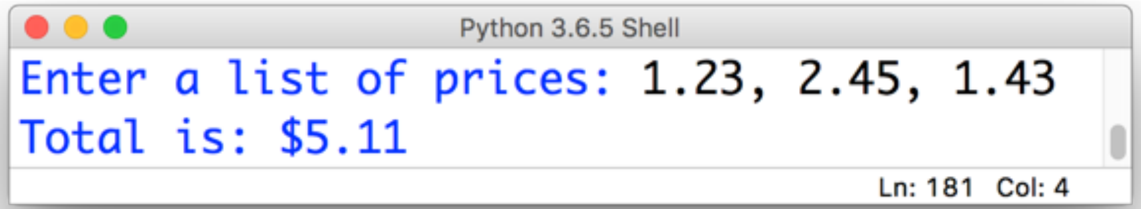
```
Enter a list of prices: 1.23, 2.45, 1.43
Total is: $5.11
```

…. but this my result in the wrong number of decimal places, as below:



```
Enter a list of prices: 1.99, .01
Total is: $2.0
```

Attempt #3: The solution is to use `.format()`
`print("Total is: ${0:2.2f}".format(my_sum))`



```
Enter a list of prices: 1.23, 2.45, 1.43
Total is: $5.11
```

String.`format()`

The `.format()` operation can to applied to a String.

    <template-string>.format(<values>)

Inside the template string is `{#}` with the index of the value in the parameters of `.format()`.

For example:

```
>>> print("1. {0}, 2. {1}, 3. {2}".format('a','b','c'))
1. a, 2. b, 3. c
>>> print("1. {0}, 2. {2}, 3. {1}".format('a','b','c'))
1. a, 2. c, 3. b
>>> c = 5
>>> print("1. {0}, 2. {1}, 3. {2}".format(7,6.5,c))
1. 7, 2. 6.5, 3. 5
```

We specify the format with the format-specifier:

    {<index>:<format-specifier>}

For example:

```
>>> print("{0:5} with five spaces".format('name'))
name  with five spaces
>>> print("{0:5} with five spaces".format('longname'))
longname with five spaces
>>> print("{0:10} with ten spaces".format('name'))
name       with ten spaces
>>> print("{0:>5} right justified with five spaces".format('name'))
 name right justified with five spaces
>>> print("{0:5} left justified with five spaces".format('100'))
100   left justified with five spaces
>>> print("{0:5} as a number with five spaces".format(100))
  100 as a number with five spaces
>>> print("{0:5.3f} as a number with three decimals with five
spaces".format(3.14159))
3.142 as a number with three decimals with five spaces
```

To add an alignment character inside the format-specifier:

- > Right justified… as in `{0:>5}`
- < Left justified… as in `{0:<5}`
- ^ Center justified…as in `{0:^5}`

There are other formatting modifiers, for example:

- `f` Fixed width… as in `{0:5.3f}`

Formatting Activity

How would your format the following table to store One Card Data:

```
+-------------------------------------------------------------------+
|First Name         |Last Name          |ID Number|  Status   |Balance |
+-------------------------------------------------------------------+
|Alicia             |Grubb              |991273053| faculty   |$  67.23|
+-------------------------------------------------------------------+
```

What assumptions do we need to make?

String Logical Operators

The equality and inequality operators can be applied to strings:

```
>>> 'a' == 'a'
True
>>> 'ant' == 'ace'
False
>>> 'a' == 'b'
False
>>> 'a' != 'b'
True
```

We can compare two strings for their *dictionary order*, comparing them letter by letter:

```
>>> 'abracadabra' < 'ace'
True
>>> 'abracadabra' > 'ace'
False
>>> 'a' <= 'a'
True
>>> 'A' < 'B'
True
```

Each character in a string is actually represented by integers following the ASCII encoding (show reference: http://simple.wikipedia.org/wiki/ASCII). Therefore when you compare two strings, what you are really doing is comparing their numerical representations. For example in ASCII the characters 'a' and 'w' are encoded as 97 and 119 respectively. The comparison `'a' > 'w'` would translates to `97 > 119` to give the result `False`.
Capitalization matters, and capital letters are less than lowercase letters:

```
>>> 'a' != 'A'
True
>>> 'a' < 'A'
False
```

We can't compare values of two different types for ordering:

```
>>> 's' <= 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>>
TypeError: unorderable types: str() <= int()
```

To obtain the ASCII (integer) representation, we can use the built-in function `ord()`:

```
>>> ord("a")
97
>>> ord("A")
65

>>> ord("ab")
Traceback (most recent call last):
  Python Shell, prompt 3, line 1
builtins.TypeError: ord() expected a character, but string of length
2 found
```

The build-in function `ord` is expecting a character, and will produce an error if used on a string of characters.

To convert from the ASCII integer representation back to a string, we can use the built-in function `chr()`:

```
>>> chr(97)
'a'
>>> chr(90)
'Z'
```

In Summary:

Description	Operator	Example	Result of example
equality	==	'cat' == 'cat'	True
inequality	!=	'cat' != 'Cat'	True
less than	<	'A' < 'a'	True
greater than	>	'a' > 'A'	True
less than or equal	<=	'a' <= 'a'	True
greater than or equal	>=	'a' >= 'A'	True
contains	in	'cad' in 'abracadabra'	True
length of string s	len(s)	len("abc")	3

We will discuss `contains` next week.

Reminder: Data Types

- `bool`: boolean

  Two possible values: `True` or `False`.
- `int`: integer

  Whole Numbers: For example: 3, 4, 894, 0, -3, -18
- `float`: floating point number (an approximation to a real number)

  Decimal Numbers: For example: 5.6, 7.342, 53452.0, 0.0, -89.34, -9.5
- `str`: strings

  Words and letters: For example: ‘dog’, ‘a’, ‘num’

We can change the type of a value or variable by *casting*:

```
     bool(), int(), float(), str()
```

- Analyze the **Problem**
- Determine **Specifications**
- Refine the ~~Create a~~ **Design**
- **Implement**
- Test & Debug

iterate many times