Python for Informatics

LESSON 3

Iteration

"The wheel has come full circle."

— William Shakespeare

- In our previous two lessons, we looked at variables and data types, and learned how these can be used to perform simple computations.
- While storing a value in a variable is useful since it allows us to come back to that variable and retrieve that value at a latter time, it is the updating of a variable—the changing of its value—that yields the full power of a variable.
- ***It is through the varying of their values that variables fully realize themselves.***

Iteration

- Because the updating of variables is such a common yet powerful activity, there are idiomatic syntactic structures that facillitate it.
- Assignment operations are paired with other operations.

```
x = 0
x = x + 1
print(x)
1
```

Iteration

- ***x = x + 1*** is a simple way of incrementing the value of ***x***.
- The syntax can be simplified even further, by employing a short-hand notation:

  x += 1

- Because this short-hand version requires less typing, it is generally preferred.

Iteration

- Our other binary operators follow suit:

    x -= 1

    x *= 1

    x /= 1

    x %= 1

- The right operand doesn't need to be 1.

    x += 5

    x += y

Iteration

- The ability to update a variable with a new value is powerful.
- While updating a variable once is powerful, updating a variable multiple times is especially powerful.
- Imagine performing a mind-numbingly tedious task. Doing it once is bad, twice… cruel, thrice… maddening,… a thousand times!…
- These are circumstances for which the massive number-crunching power of our computers brilliantly shines.

Iteration

- Try this…

```
n = 0
while n < 1000:
    n += 1
print('done!')
```

- Now, try it again, but instead of 1,000 make it 10,000 (don't forget to reset n to zero).
- Continue to add an extra zero until you finally notice a time delay.
- On my computer, I finally notice a slight delay with a value of 1,000,000. It isn't until 10,000,000 that the delay is significant!

Iteration

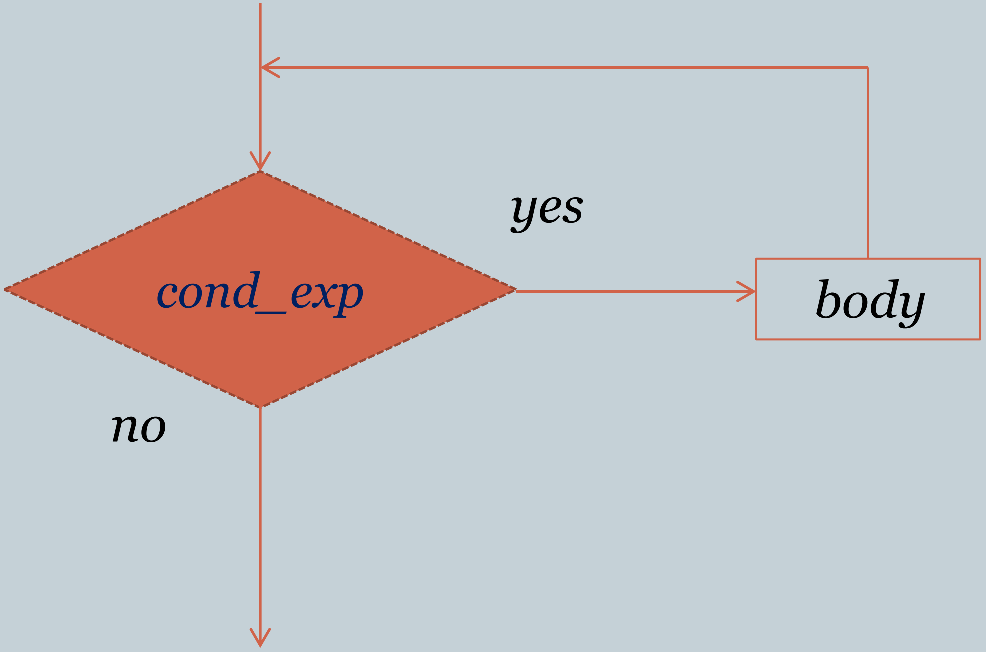
- Let us look at the general syntax:

```
while conditional_expression:
    update_control_variable
```

- The ***conditional_expression*** is an expression that evaluates to either ***True*** or ***False***.
- If the ***conditional_expression*** evaluates to ***True***, then the body of the ***while*** is executed, and the program flow loops back to the top of the while statement.
- The ***conditional_expression*** will be evaluated again, and the body will be executed again if the ***conditional_expression*** evaluates to ***True***.
- This process of evaluation/execution will repeat or iterate until such time that the ***conditional_expression*** evaluates to ***False***.
- When the ***conditional_expression*** evaluates to ***False***, the body will be skipped, and the program flow will drop down and continue past the while statement.

Iteration

- Here is a flow chart visualization of the while loop:



- Each execution of the loop is called an iteration.

Iteration

- Consider this:

```
m = 0
n = 1000
while n > 0:
    m -= 1
```

- Under what condition would this loop stop iterating?
- The control variable ***n*** is not modified in any way that would allow the loop to stop iterating.
- A loop that never stops iterating like this is called an ***infinite loop***.
- Go ahead and try this code. You'll need to either restart the Kernel, or restart Canopy.

Iteration

- Sometimes an infinite loop can serve as a convenience, when you don't know exactly how many iterations you want your loop to make.
- An ***infinite loop*** combined with a ***break*** statement can be an elegant solution:

```
while True:
    line = raw_input('> ')
    if line == 'Ni!':
        break
    print line
print('Oh, what sad times are these when passing
ruffians can say Ni at will to old ladies.')
```

Iteration

- Note, however, that this code could be written to not rely upon a ***break*** statement.

```
line = ''
while line != 'Ni!':
    print line
    line = raw_input('> ')
print('Oh, what sad times are these when passing ruffians can say Ni at will to old ladies.')
```

- As a general rule, you should be sparing in the use of break statements.
- ***break*** statements force your code to skip and jump in a manner that can be confusing and error prone.
- If at all, only consider using ***breaks*** when the code is clear and well-contained.

Iteration

- The ***continue*** statement is another way of redirecting program flow within a loop.
- With ***continue***, the program flow stays within the loop, but skips any remaining statements between the ***continue*** statement and the end of the loop.
- Whereas ***break*** forces an exit from the loop, the ***continue*** forces skipping to the next iteration of the loop.

```
while True:
    line = raw_input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print line
print 'Done'
```

Definite vs. Indefinite Loops

- The ***while*** loop is an example of an ***indefinite loop***.
- The number of iterations performed by a while loop depends upon a condition (the evaluation of a boolean True/False expression).
- The while loop is indefinite because we cannot know *a priori* how many times the loop will iterate.
- The ***for*** loop is an example of a ***definite loop***.
- The ***for*** loop is used to ***iterate through an entire set of items***.
- ***If you need to process each item within a set, the for loop is preferred***, as it precludes the possibility of erroneously skipping any item within the set.

Iteration

- Here's an example of the ***for*** loop at work:

```
comedians = ['Graham Chapman', 'John Cleese',
             'Terry Gilliam', 'Eric Idle', 'Terry Jones',
             'Michael Palin']
for comedian in comedians:
    print(comedian + ' is hilarious! ‘)
print('A flying circus!')
```

Graham Chapman is hilarious!
John Cleese is hilarious!
Terry Gilliam is hilarious!
Eric Idle is hilarious!
Terry Jones is hilarious!
Michael Palin is hilarious!
A flying circus!

Iteration

- Note that ***comedian*** is the ***iteration variable*** within the loop.

```
comedians = ['Graham Chapman', 'John Cleese',
             'Terry Gilliam', 'Eric Idle', 'Terry Jones',
             'Michael Palin']
for comedian in comedians:
    print(comedian + ' is hilarious! ‘)
print('A flying circus!')
```

- The value of ***comedian*** is guaranteed to be that of each successive item with the set being iterated over.

Common Looping Practices

- Common steps or operations that are typically performed as part of a loop construct include:
  - Initializing one or more variables before the loop begins.
  - Performing an operation or computation upon each item within the loop body, and perhaps modifying variables.
  - Making use of the resulting values of your variables, after completion of the loop.

Canonical Loop Forms

- Here's a loop that counts the number of items in a list:

```
count = 0
for cur_num in [5, 82, 35, 8, 27, 19]:
    count += 1
print('The number of items is ' + count)
```

Canonical Loop Forms

- Here's a loop that totals the number of items in a list:

```
total = 0
for cur_num in [5, 82, 35, 8, 27, 19]:
    total += cur_num
print('The total is ' + total)
```

Canonical Loop Forms

- Here's a loop that finds the largest value within a list:

```
maximum = None
for cur_num in [5, 82, 35, 8, 27, 19]:
    if maximum is  None or cur_num > maximum :
        maximum = cur_num
print('The maximum value is ' + str(maximum))
```

Canonical Loop Forms

- Here's a loop that finds the smallest value within a list:

```
minimum = None
for cur_num in [5, 82, 35, 8, 27, 19]:
    if minimum is  None or cur_num < minimum :
        minimum = cur_num
print('The minimum value is ' + str(minimum))
```

Canonical Loop Forms

- Here's a loop that finds the smallest value within a list:

```
minimum = None
for cur_num in [5, 82, 35, 8, 27, 19]:
    if minimum is  None or cur_num < minimum :
        minimum = cur_num
print('The minimum value is ' + str(minimum))
```

Canonical Loop Forms

- The previous loop examples demonstrate typical forms of applying loop-based operations.
- With the exception of the total counting example, you would not need to use those code snippets, because there are built-in functions that already perform the same task:

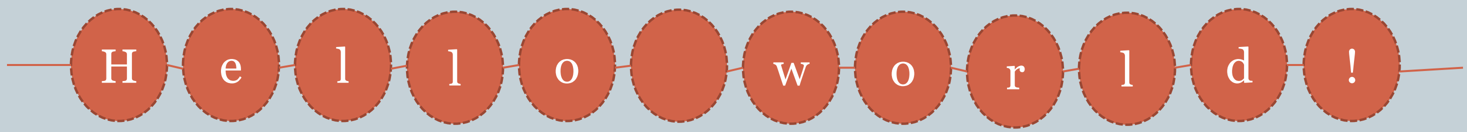
```
sum([5, 82, 35, 8, 27, 19])
176

max([5, 82, 35, 8, 27, 19])
82

min([5, 82, 35, 8, 27, 19])
5
```

Strings

- A String is a sequence of characters.
- Imagine a series of beads with a character carved into each one. By sliding them onto a string, you create a necklace.



- Syntactically, each character can be referenced by means of an integer index value.

H	e	l	l	o		w	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

greeting = 'Hello world!'

print(greeting[6])

w

Strings

- The built-in len function, allows us to get the length of any given string.

H	e	l	l	o		w	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

greeting = 'Hello world!'

print(len(greeting))

12

Strings

- Note that the last character in a string is always at the position of ***len(…) – 1***.

H	e	l	l	o		w	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

greeting = 'Hello world!'

print(greeting[len(greeting) - 1])

!

Strings

- Referring to a position that is beyond the end of the string results in an IndexError.

H	e	l	l	o		w	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

greeting = 'Hello world!'

print(greeting[len(greeting)])

IndexError: string index out of range

Strings

Just as with any kind of sequence, it is often useful to process a string one item (i.e. character) at a time.

H	e	l	l	o		w	o	r	l	d	!

```
greeting = 'Hello world!'
index = 0
while index < len(greeting) :
    letter = greeting[index]
    print(letter)
    index += 1
H
e
l
l
o
!
```

Strings

- A ***slice*** is a substring denoted by ***s[n:m]***, where for string ***s***, a string of characters is returned beginning at position ***n*** and extending up to but not including position ***m***.

P	y	t	h	o	n	e	s	q	u	e
0	1	2	3	4	5	6	7	8	9	10

word = 'Pythonesque'
print(word[0:6])
Python

print(word[4:7])
one

Strings

- Strings are immutable! Therefore, as much as you might want to, you cannot change them.
- You can, however, create new strings.

  Incorrect – attempting to modify an existing string…

  word = 'Pythonesque'

  word[0] = 'M'

  TypeError: 'str' object does not support item assignment

  Correct – creating a new string…

  word = 'M' + word[1:11]

  print(word)

  Mythonesque

Strings

- Counting across a string. Here’s an example of performing a count of the number of ‘e’s in a string.

P	y	t	h	o	n	e	s	q	u	e
0	1	2	3	4	5	6	7	8	9	10

```
word = 'Pythonesque'
count = 0
for letter in word :
    if letter == 'e' :
        count += 1
print('There are ' + str(count) + ' \'e\'s ' + 'in the word. ')
```

Strings

- The ***in*** operator takes two string operands and returns true if the left string is found as a substring within the right operand.

P	y	t	h	o	n	e	s	q	u	e
0	1	2	3	4	5	6	7	8	9	10

word = 'Pythonesque'

'on' in word

True

Strings

- The comparison operators allow us to compare two strings.

0	1	2	3	4	5	6	7	8	9	10
P	y	t	h	o	n	e	s	q	u	e

```
if word == 'Pythonesque' :
    print('Ni!' )
Ni!
```

Strings

- The comparison operators allow us to compare two strings.

0	1	2	3	4	5	6	7	8	9	10
P	y	t	h	o	n	e	s	q	u	e

word > 'Python'

True

'King Arthur' < word

True

Strings

- In Python, all uppercase characters are lexigraphically less than (or to the left of) all lowercase characters.

  'Zymurgy' < 'aardvark'

  True

- To account for this we can convert to a common format by using the ***lower()*** or ***upper()*** string functions.

  'Zymurgy'.lower() < 'aardvark'.lower()

  False

Strings

- Python strings are actually ***object***s.
- An object is a binding together of data and executable code.
- The idea is to keep data and the functions that operate upon that data close together.
- The binding of data and functions (or behavior) is accomplished by means of ***encapsulation***.
- ***Encapsulation*** effectively creates a skin around the data and its related functions—this is what grants objects their objectness.

Strings

- The functions that belong to objects are called ***methods***.
- The term method comes from the classical perspective of object-oriented programming wherein objects are said to send ***messages*** to each other.
- ***When an object receives a message, the method is the way that it responds to that given message.***
- *By the way, **in Python everything is actually an object—even ints and floats are objects!***

Strings

- By using the type function, we can determine what type or class of object we have.

  s = 'aardvark'

  type(s)

  str

Strings

- The ***dir*** function lists the methods that belong to an object.

  dir(s)

  ['__add__',

  ...

  'capitalize',

  ...

  'format',

  ...

  'zfill']

Strings

- Seeing that capitalize() is a string method, we can learn more about it by asking for help.

help(str.capitalize)

Help on method_descriptor:

capitalize(...)

S.capitalize() -> string

Return a copy of the string S with only its first character capitalized.

Strings

- To see how it works, we can call or ***invoke*** it.

  s.capitalize()

  'Aardvark'

- Note that objects are invoked by putting a dot after the object reference, and then specifying the method name.
- This syntactic use of a dot to invoke a method is called ***dot notation***.

Strings

- Now let's ***invoke*** the ***find()*** method.

```
s = 'shrubbery'
s.find('r')
2

s.find('r', 3)
7

s.find('er', 2)
6
```

Strings

- An important "clean-up" method is ***strip()***.

```
s = '     shrubbery     '
s = s.strip()
print(s[0:5])
shrub
```

- ***rstrip()*** is a similar method, but it only strips at the end, and it lets you specify a non-whitespace character to strip out.

```
s = '*****shrubbery*****'
s = s.rstrip('*')
print(s)
*****shrubbery
```

Strings

- ***startswith()*** is also a convenient method.

    s = 'dead parrot'

    s.startswith('dead')

    True

Strings

- We often use a series of operations to parse data.

```
s = 'http://www.stpacossecondchancedogrescue.org/user/login'
start_pos = s.find('://') + 3
end_pos = s.find('/', start_pos)
home_page = s[start_pos : end_pos]
print(home_page)
www.stpacossecondchancedogrescue.org
```

Strings

- Note that by refactoring our code, we can make it clearer and more reusable.

```
target_start_str = '://'
target_end_str = '/'
s = 'http://www.stpacossecondchancedogrescue.org/user/login'
start_pos = s.find(target_start_str) + len(target_start_str)
end_pos = s.find(target_end_str,  start_pos)
home_page = s[start_pos : end_pos]
print(home_page)
www.stpacossecondchancedogrescue.org
```

Strings

- The format operator, ***%***, allows us to construct formatted strings by means of ***string interpolation***.

```
s = '%s. Go on. Off you go.' \
% (raw_input('What... is your favourite colour? '))
print(s)
```

- The format operator has two operands, where the first is the string that contains one or more format sequences.
- Each format sequence is a place holder for the string being constructed.
- The second operand is a tuple (comma delimited value sequence) containing the values that will be inserted into the successive place holders.

Strings

- The various format sequences are based upon the type of value they are holding a place for.
- The format sequences for decimal, floating point, and string are, respectively:

%d
%g
%s