

Lecture 1: 1/10

- Compiled vs. Interpreted
 - **Compiled** - tries to compile the entire code before it executes
 - **Interpreted** - reads each line at a time and runs it after it reads the line

Strings

- There are two ways to concatenate strings together

Option 1: Using the + symbol

Option 2: Using the ,

- The comma adds a space for you rather than the + option from having to use a space

```
print("Hello " + "World")
print("Hello", "world")
```

Escape Characters

- It doesn't follow what they would normally do in Python
- `\n` = new line
- `\t` = new tab

Commenting

There are 2 ways to comment in Python. There is the option for a single line comment withh "#" and then there is the multiple line comment: `''' This is a multiple line comment '''`

Lecture 2: 1/12

Variable Data Types

- Integer **int**
- Real Numbers **float** - any number with a decimal point

- Character Strings **str**
- Boolean **bool**

Variables are case sensitive, these are all different forms: (Use of camel case)

- age
- AGE
- Age

Strings

You can perform methods on strings with

Method	Description
<code>s.upper()</code>	Returns the uppercase version of the string.
<code>s.lower()</code>	Returns the lowercase version of the string.
<code>s.swapcase()</code>	Returns a new string where the case of each letter is switched.
<code>s.capitalize()</code>	Returns a new string where the first letter is capitalized and the rest are lowercases.
<code>s.title()</code>	Returns a new string where the first letter of each word is capitalized and all others are lowercase.
<code>s.strip()</code>	Returns a string where all the white space (tabs, spaces, and newlines) at the beginning and end is removed.
<code>s.replace(<i>old</i>, <i>new</i>)</code>	Returns a new string where occurrences of the string <i>old</i> are replaced with the string <i>new</i> .

Input

Getting user input through `input()`. Example below:

```
name = input("What is your name?")
```

The `input()` operation will **ALWAYS** return a string. If you have numbers, you can cast them

```
number = int(input("Enter a number"))
```

There are other casting functions:

Function	Description	Example	Returns
float(x)	Returns a floating-point value by converting x	float("10.0")	10.0
int(x)	Returns an integer value by converting x	int("10")	10
str(x)	Returns a string value by converting x	str(10)	'10'

Lecture 3: 1/17/17

There are different arithmetic operations that you can perform on integers:

Operator	Description	Example	Evaluates To
+	Addition	7 + 3	10
-	Subtraction	7 - 3	4
*	Multiplication	7 * 3	21
/	Division (True)	7 / 3	2.333333
//	Division (Integer)	7 // 3	2
%	Modulus	7 % 3	1
**	Exponent	7 ** 3	2401

There are also other operations that you can do on a float:

Operator	Description	Example	Evaluates To
+	Addition	7.0 + 3.0	10.0
-	Subtraction	7.0 - 3.0	4.0
*	Multiplication	7.0 * 3.0	21.0
/	Division (True)	7.0 / 3.0	2.333333
//	Division (Integer)	7.0 // 3.0	2.0
%	Modulus	7.0 % 3.0	1.0
**	Exponent	7.0 ** 3.0	2401.0

If you do any operations with an int and a float, it becomes a float!

Division

There are 2 different ways of dividing in Python.

- **True Division** (/)
 - Answer is **always a float**
 - Keeps the decimal
- **Integer Division** (//)
 - Only gives you the whole number part and throws away the decimal

Modulo

- Modulus - gives you the remainder from division

There is an order of operations that Python will follow when it is given an equation with multiple of them and they are:

Category	Operators
Parentheses (grouping)	()
Exponent	a**b
Positive, Negative	+a, -a
Multiplication, Division, Modulus	a * b, a / b, a // b, a % b
Addition, Subtraction	a + b , a - b

Branching (Conditionals)

Flow of Control - the order a program performs actions, done with branching statements

Conditionals

Conditionals evaluate to **True** or **False**. Example: `if number > 1:`

Block - one or more consecutive lines indented by the same amount

Parts that make up a condition:

- `if [condition]`
- `elif [condition]`
- `else:`

There are different comparison operators that you can use in the conditional statements:

Operator	Meaning	Sample Condition	Evaluates To
<code>==</code>	equal to	<code>5 == 5</code>	True
<code>!=</code>	not equal to	<code>8 != 5</code>	True
<code>></code>	greater than	<code>3 > 10</code>	False
<code><</code>	less than	<code>5 < 8</code>	True
<code>>=</code>	greater than or equal to	<code>5 >= 10</code>	False
<code><=</code>	less than or equal to	<code>5 <= 5</code>	True

Lecture 4: 1/19/17

Conditionals (cont.)

Strings are evaluated to true. So if you had:

```
mystery = "chicken"
if mystery:
    print("This is true")
```

The result would be "This is true".

Numbers

- 0 and 0.0 are **False**
- ALL other numbers are **True**

Strings

- The empty string `""` is **False**
- Everything else (that is a string) is **True**

Logical Operators

- The 3 bottom logical operators ust be a combination of simple conditions between 2 boolean expressions
- **not** - the opposite of something
- **and** - where *BOTH* of the conditions must be true
 - `number > 5 and number < 10`
- **or** - where *EITHER* of the conditions can be true
 - `number == 5 or number == 10`

A	B	A and B	A or B	not A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Importing Modules

Module = Libraries

- You `import moduleName` the different modules at the top of the program file
- To access a function you use: `moduleName.function()`

Random Module

- To import the random module you type `import random` at the top of your code
- When using `random.randrange(5)` it will give you a number from 0 to 5.
- To get a range from 1 - 6, you would just shift the range to `random.randrage(5) + 1`

Lecture 5: 1/24/17

Lecture 6: 1/26/17

For Loops

It repeats your code, but it's not based on a condition. Instead, it's based on a sequence.

You have to follow the key word `for` with a **new** variable that will **only** be used within that for loop. Then you use the reserved word `in` and then there is the **sequence** that you want to loop through.

The variable that you declared will be assigned the different variable possibilities defined by the sequence.

Ex: `for var in sequence:`

Creating sequences w/ range()

- **range()**, goes up to but doesn't include the stop number. It also has many different functionalities:
 - `range(6)` = 0,1,2,3,4,5
 - `range(int start, int stop, int step)` => `range(10,25,5)` = 10,15

Sequential access - going through a sequence 1 element at a time **Random access** - you can get any element in the sequence directly

String as a Sequence

```
msg = "spamalot"
for letter in msg:
    print(letter.upper(), end=" ")
```

Other things that are a sequence i the range() function

Length of String

You can get the lenght of a string with `len()`

Example: `python msg = "Hello" length = len(msg)` This will set length to 5, b/c there are 5 characters in the string msg.

in Operator

The `in` operator returns **True** if the element is a member of the sequence & **False** if it is not.

```
msg = "spamalot"
if "spam" in msg:
    print("Found")
else:
    print("NOT found")
```

This will print out "Found"

You can also build a string, one character at a time, using the concatenation (+)

ASCII Conversion

- `ord(<letter>)` will convert the letter that you have in the parenthesis into an integer of what it is in ASCII
- `chr(num)` this will convert the integer into a character

Lecture 7: 1/31/17

Indexes

- The first item has an index of 0
- The last index has an index of (num of items) -1
- **Random Access** - when you want to be able to find the certain element at that index. If you try and go out of the index, you will get an "error"
- You can also access a word using negative indices, where it will go backwards

```
msg = "spamalot"
print(msg[2]) = a
print(msg[10]) = ERROR
print(msg[1:6]) = pamal
```

The above code will print "a". The second line will give you an error b/c you are trying to access an index outside of the amount of words.

Slicing

- You can get a range through the [] operator: `variable[startPosition: endPosition]`
- Start @ beginning up to that point `print(msg[:3]) = "spa"`
- Go to end from point `print(msg[4:]) = "alot"`
- Print the whole word `print(msg[:]) = "spamalot"`

Differences

```
word = "barista"
a = word
b = word[:]
print(a)
print(b)
```

Both of the lines do the same thing, but a is a pointer, where b is a copy of it.

2 types of sequences:

- **Mutable** - (changeable)
 - can modify a single item in the sequence
- **Immutable** - (unchangeable)
 - you **CANNOT** modify a single item in the sequence
 - Ex: Strings

Lists

- Lists are like sequences, but they are **mutable**
- You can create a list with both strings and numbers
- Ex: `myList = [item1, item2, ...]`
 - item1 can be any variable type:
 - String
 - int
 - float
 - another list
- You can add 2 lists together

- You can add an additional item after the list has already been declared with the function

```
.append(value)
```

- `myList.append("hello")`
- Will add the value to the end of a list
- You can check to see if there is something that exists within a list with an if statement
 - if "dog" in stuff:

```
things = ["emu", "pig"]
stuff = ["dog", "cat", "boa"]
things += stuff #adding the 2 lists together
```

You can also do the other operators on lists. Like indexing & slicing.

Lecture 8: 2/2/17

There are many different operations you can perform on a list:

Method	Description
<code>someList.append(value)</code>	Adds value to end of a list.
<code>someList.sort()</code>	Sorts the elements, smallest value first.
<code>someList.reverse()</code>	Reverses the order of a list.
<code>someList.count(value)</code>	Returns the number of occurrences of value.
<code>someList.index(value)</code>	Returns the first position number of where value occurs.
<code>someList.insert(i, value)</code>	Inserts value at position i.
<code>someList.pop([i])</code>	Returns value at position i and removes value from the list. Providing the position number i is optional. Without it, the last element in the list is removed and returned.
<code>someList.remove(value)</code>	Removes the first occurrence of value from the list.
<code>del someList[i]</code>	Removes the element at the specified index

- **.sort()** - will **NOT** return a new list
- You can use the **.remove()** to remove a certain value from the list
- **del** - will delete some random value, you aren't searching for the value you want to delete:

```
del someList[index]
```

Delimiter

It's how you can join some list together

Join

```
wordList = ["Always", "look", "on", "the", "bright", "side" , "of", "life"]
delimiter = " "
quote = delimiter.join(wordList)
print(quote)
```

The delimiter can be anything

Split

```
quote = "spam-spam-spam"
delimiter = "-"
wordList = quote.split(delimiter)
print(wordList)
```

Lecture 9: 2/7/17

Functions

You define a function with `def` , which is followed by the **function name**, then a pair of parenthesis, a colon, and the indented block of statements

```
def functionName(parameters):
    statements
```

You just **call** a function by using the name of the function and the parenthesis

`functionName(parameters)` . You **must** define the function before you call it.

Parameters

You can pass in parameters into the function call.

Default Parameter Values

You can assign default values to your parameters (if there was no value passed in).

If you assign default values to a parameter in a list, you have to assign default values to all the parameters after it. If you want to use default values, put them at the end of your list.

```
#default parameters
def birthday(name = "Cooper", age =1):
    print("Happy Birthday " + name + "! You are " + str(int(age)))

birthday()
birthday(name = "Carter")
birthday(age = 6)
birthday("Nicole")
```

- 1) Happy Birthday Cooper! You are 1
- 2) Happy Birthday Carter! You are 1
- 3) Happy Birthday Cooper! You are 6
- 4) ERROR

Return Values

```
def functionName(parameters):
    statements
    return value

var = functionName(parameters)
```

The var is the variable that you are using to store the value that gets returned by the function.

You can return **multiple** values. They are separated by commas, but be sure to have enough variables to catch all of the returning values.

Lecture 10: 2/9/17

main()

- Take no arguments
- Returns no arguments
- You can define your functions in any order, as long as you call on `main()` at the end of the file

namespaces

- also called **scopes**
- represent different areas of your program that are separate from each other
 - function cannot access another variable that is not in its namespace

constants

- **Constant** is a variable that can not change
- Style: constants are ALLCAPSWITH_UNDERSCORES
 - Ex: SALES_TAX

Global constants

- constants that are created in a global namespace
- They can be accessed anywhere
- Their values cannot be changed once they are assigned

Lecture 11 & 12: 2/14 & 2/16

Files

If you want to be able to access data after the program has ended, you can save it to a file for permanent storage.

File - collection of information (stored in bytes), they live on the computer's hard drive, phone memory, etc.

Kinds of Files

- Files are either stored as:
 - **Text** files store data in human-readable formats
 - Ex: Simple text files (.txt) or web pages (.html)
 - **Binary** files store data in computer-readable formats
 - Ex: Pictures (.jpg), music (.mp3), or word doc (.docx)

We will be using files to: save the data from the program, share information, and to write programs that use multiple data files.

Reading from a File

3 Step Process:

1. Open the file for reading
2. Read from the file
3. Close the file

Reading Files

Reading is the process of getting data from a file that is on the computer.

We use the `open()` operator to open a file.

```
fileIn = open("words.txt", "r")
```

The "r" is the "file access mode" where we are only reading in the information from the file.

The operation returns a **file object**

- One method to read through all of the lines of a text file is to use a *for* loop.
- Each time 1 entire line is read from the file, there is a new line character at the end (`\n`).
- Remove the newline when reading with `string.strip()`
 - This will remove **any** white space at the beginning & end of the string
 - Whitespace:
 - A space
 - A tab (`\t`)

- A newline (\n)

```
fileIn = open("words.txt", "r")
for line in fileIn:
    print(line)
```

Closing the file

- After you are finished reading from the file you have to close it, this prevents the corruption of the file

```
fileIn = open("file.txt", "r")
for line in fileIn:
    word = line.strip()
    print(word)
fileIn.close()
```

- The "line" variable in the example above will **ALWAYS** be a *string*.
- If the file is of integers, we will need to convert line --> int (like we have to do with input)

```
for line in FileIn:
    line = int(line.strip())
    print(2*line)
```

Writing to a File

3 step Process:

1. Open the file for "writing"
2. Write to the file
3. Close the file

Open file for writing

- Use the built-in function `open()`. It comes to `fileOut = open("file.txt", "W")`

Writing to the file

- You will use the `print` function with the file argument


```
fileOut = open("results.txt", "w")
print("Hello World", file=fileOut)
```

Closing the file

- AFTER you are done writing to the file, you will have to close it

```
fileOut = open("results.txt", "w")
print("Hello World", file=fileOut)
fileOut.close() #this is the line that closes the file
```

This Diagram below shows the different File Access Modes

Mode	Description
"r"	Read from a file. If the file doesn't exist, Python will generate an error.
"w"	Write to a file. If the file exists, its contents are overwritten. If the file doesn't exist, it's created.
"a"	Append a file. If the file exists, new data is appended to it. If the file doesn't exist, it's created.

The Diagram below shows additional File Access modes

Mode	Description
"r+"	Read from and write to a file. If the file doesn't exist, Python will generate an error.
"w+"	Write to and read from a file. If the file exists, its contents are overwritten. If the file doesn't exist, it's created.
"a+"	Append and read from a file. If the file exists, new data is appended to it. If the file doesn't exist, it's created.


Comma-separated Value (CSV) files

- CSV files are commonly used to exchange data
- They are text files that can be used to represent the same data as a spreadsheet
- they are convenient b/c they can be read by any program, platform, etc.

CSV Format

- Each **row** represents one **line** in a table
- Commas separate each column
- The first line normally represents the headers

CSV Example



DEPT	COURSE_NUMBER	SEMESTER	NUMBER_OF_STUDENTS
ITP	115	Spring17	30
BUAD	101	Spring17	40
ITP	310	Spring17	35

class.csv

```
DEPT,COURSE_NUMBER,SEMESTER,NUMBER_OF_STUDENTS
```

- CSV files are processed using the `split()` function

Extra from this lecture Deck:

- **readline(n)** is the method where n = the # of characters you want to read from the current line
 - it returns chracters from the current line only
- **readlines()** - this is a method that willl read the entire textfile into a list
 - Each line within the file becomes a separate string element in the list

Writing Individual String to a Text File

- Use the `write(string)` that will write a string to a text file
- `write()` does not automatically intert a character at the endoof a string
- If you want there to be a newline, you will have to add the character (\n)

Writing a List of Strings to a Text File

- You can write a list of strings to a text file using: `writelines(someList)` where "someList" is a list of strings

Lecture 13: 2/21/17

Lists Review:

- They are an ordered sequence of things
- You access items by index

Dictionaries

- They store information in **pairs**
 - key and a value
- Use curly braces to define it { }, they are separated by commas
 - `myDictionary = {key1:value1, key2:value2, key3:value3}`
- It is not ordered sequentially, you access items by their *key*
- Keys must be unique & immutable (string, number or tuple)
 - The value does not have to be unique

Dictionary Operations

- Testing for a key within the **in** operator before retrieving the value
 - Use the **get()** method to retrieve a value
- Add a key-value pair
- Replace a key-value pair
- Delete a key-value pair

Operations listed:

Method	Description
<code>len(dict)</code>	Returns number of entries in dict
<code>dict.get(key, [default])</code>	Returns the value of key. If key doesn't exist, then the optional default is returned. If key doesn't exist and default isn't specified, then None is returned.
<code>dict.pop(key, [default])</code>	Removes the key and returns the value. If key doesn't exist, then the optional default is returned. If key doesn't exist and default isn't specified, then None is returned.
<code>dict.keys()</code>	Returns a list of all the keys in a dictionary.
<code>dict.values()</code>	Returns a list of all the values in a dictionary.
<code>dict.items()</code>	Returns a list of all the items in a dictionary. Each item is a two-element tuple (key, value)
<code>del dict[key]</code>	Removes the key. If key doesn't exist, then an error is generated.

```
info = {
    "name": "juan",
    "age": 47,
    "job": "doctor"
```

Reading from a Dictionary: `username = info["name"]` will return "juan"

You can print out everything within a Dictionary, but they might come out in a different order every time. You can read everything with a for loop

```
for key in info:
    print(info[key])
```

Adding Key/Values to Dictionary & Replacing

`info["kids"] = 2` Adding a value to a dictionary, is the same as changing the value associated with a key.

You can find out the size of a Dictionary: `size = len(info)`

Check if a key is in a dictionary:

```
if "age" in info:
    print("Found key age")
```

Deleting Keys

```
if "kids" in info:
    del info["kids"]
```

Checking to see if the key exists, before you try to delete it.

Getting a list

You can get a list of **all** the keys.

```
keyList = list(info.keys())
print(keyList)
```

You can also get a list of all the values.

```
valuesList = list(info.values())
print(valuesList)
```

Lecture 15: Web Scraping (won't be on the midterm)

- **Web scraping** - a technique used to collect data and other information from websites for further use, it's good for writing web scraping scripts and parsing websites
- This is used to detect fraudulent reviews
- HTML is used for creating languages and uses "tags" to translate text into a visual format
- "id" attribute value is unique
- "class" attribute is not unique

Beautiful Soup - python module made for parsing HTML and XML files

Import the necessary tools

```
from bs4 import BeautifulSoup
import urllib.request

url = "https://www.google.com/"
page = urllib.request.urlopen(url)
soup = BeautifulSoup(page.read(), "html.parser")
print(soup.prettify())
```

- `import urllib.request` will request the website
- `print(soup)` will show all of the html, but `print(soup.prettify())` will do all the proper indentation

Navigating through Tags

```
for tag in soup:
    # "tag" is a tag from the HTML doc
```

Accessing Tags

- `soup.title` will give you the title tag
- `soup.a` will give you the first a tag, but won't give you anything else

Getting Tag Information

- Get a tag's name: `tag.name`
- Get a dictionary of attributes: `tag.attrs`
- An attribute's value `tag[attrName]`

To learn more from the Documentation of the **Beautiful soup**, the website is [Beautiful suit Documentation](#)

Error Lecture

NOT going to be on the Midterm, but will e on future assignments

The different Error Checking calls

Function	Description
<code>string.isalnum()</code>	Returns True if string contains only letters and numbers Returns False otherwise
<code>string.isalpha()</code>	Returns True if string contains only letters Returns False otherwise
<code>string.isdigit()</code>	Returns True if string contains only digits Returns False otherwise
<code>string.isspace()</code>	Returns True if string contains only whitespace Returns False otherwise

Checking if input is a digit

Use the `.isdigit()` to check.

You deal with them using **try** and **except** clause.

There are many different types of exception Clauses

Exception Type	Description
IOError	Raised when an I/O operation fails, such as when an attempt is made to open a nonexistent file in read mode.
IndexError	Raised when a sequence is indexed with a number of a nonexistent element.
KeyError	Raised when a dictionary key is not found.
NameError	Raised when a name (of a variable or function, for example) is not found.
SyntaxError	Raised when a syntax error is encountered.
TypeError	Raised when a built-in operation or function is applied to an object of inappropriate type.
ValueError	Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value.
ZeroDivisionError	Raised when the second argument of a division or modulo operation is zero.

