Scripting Languages

- One way scripting languages contrast from low-level language is in the way they implement built-in features that save programming effort (i.e. Python defines its own hash tables via the Dictionary structure)
- Scripting languages also handle low-level resource management, such as pointers (Object-Oriented scripting languages like Python utilize garbage collection to handle memory leaks)

Python Cheat Sheet

- Python is an object-oriented language, meaning that every value is an object (even numbers).
  - Objects are characterized by (1) an address, (2) a type, and (3) a value. (1) and (2) cannot change (as this would not make sense). (3) can only change if the object is mutable. Otherwise, the object is immutable.
- `is` and `is not` are used to compare objects by identity (address). Do not get this confused with equality or comparison operators, which compare value.
- Python Types:
  - `None` - null
  - Numbers (All Immutable):
    - `Int`
    - `Float`
    - `Complex`
  - Sequences:
    - `String` (Immutable)
    - `List` (Mutable)
    - `Tuple` (Immutable)
  - Mapping:
    - `Dictionary` (Mutable)
  - Callable:
    - `Function` (Mutable)
    - `Class` (Mutable)

Sequence Operations

- `x in s` Checks if x is in the sequence
- `s[i]` accesses the ith element of the sequence. Valid indices are 0 <= i <= len(s) - 1 OR -len(s) <= i <= -1. Negative indices go backward, starting from the very end of the sequence.
- `s[i:j]` Selects a subsequence of s from [i, j)
- `s[i:]` Selects a subsequence from i to everything after
- `s[:j]` Selects a subsequence from the start to j (not including j)
- `len(s)` Returns the length of s
- `min(s)` Returns the minimum (compared) element of the sequence
- `max(s)` Returns the maximum (compared) element of the sequence

- `list(s)` Constructs a List whose elements are the same as the sequence
- For mutable sequences:
    - `s[i] = v`
    - `s[i:j] = s1` Assigns the subsequence of s with s1 (which MUST be a sequence)
    - `del s[i]` Deletes the ith element from s
    - `del s[i:j]` Deletes the subsequence [i, j)

List Operations

- All sequence operations also apply to lists
- Lists in Pythons are analagous to vectors in C++
- List comprehension: `s = [(OUTPUT) for VARIABLE in COLLECTION if CONDITION]`
    - Example: `s = [x for x in range(10) if (x % 2 == 0)]`
- `s.append(v)` Adds the element v to the *end* of the list
- `s.extend(s1)` Adds the *sequence* s1 to the end of the list
- `s.count(v)` Counts the number of elements v in the list
- `s.index(v)` Returns the first index of v in the list or an error
- `s.insert(i, v)` Inserts v at the ith element in s
- `s.pop(i)` Deletes the last element in the list and returns it; if an argument i is provided, then it deletes the element at i and returns it
- `s.remove(v)` Finds and removes the first instance of v
- `s.reverse()` Reverses the list in place
- `s.sort()` Sorts the list in place
- `s.copy()` Returns a copy of the list

String Operations

- All sequence operations (that are not for mutable sequences) apply to strings
- `s.index(sub, start, end)` Looks for the index of substring sub, with the optional arguments start and end specifying the bounds to search
- `s.join(t)` Joins all strings in t with s as a separator. Note that s must be either a single string or a collection of strings
    - i.e. `".".join("hey")` -> "h.e.y"
    - i.e. `" ".join(["ABC", "DC"])` -> "ABC DC"
    - i.e. `"".join(["ABC", "DC"])` -> "ABCDC"
- `s.split(sep, maxsplit)` Splits the string s at separator, optionally at maxsplit times, returning a list of separated strings
    - i.e. `"Help ME".split(" ")` -> ["Help", "ME"]
- `s.format()` Replaces empty braces placeholders in a string with its arguments
    - i.e. `"Message {one} and Message {two}".format(one = "HELLO", two = "BYE")` -> "Message HELLO and Message BYE"
- `s.upper()` Converts all lowercase letters to uppercase
- `s.lower()` Converts all uppercase letters to lowercase

Dictionary Operations

- Dictionaries are effectively just hash tables. The key to a dictionary must be immutable (as changing the key would no longer allow for you to search for it properly)
  - Objects can be keys, and they are typically hashed using the object address (which cannot change)
- `d = {}` Creates a unique empty dictionary
- `d = {"hello" : 12}` Assigns the value 12 to the key "hello"
- `d[k]` Looks up the entry in the dictionary with key k and returns its value or an error if it is not in there
- `d[k] = v` Stores the element v for the key k or replaces its existing value
- `len(d)` Returns the length of the dictionary
- `d.clear()` Clears out the dictionary
- `d.copy()` Creates a copy of the dictionary
- `d.items()` Returns a list of (key, value) tuples
- `d.keys()` Returns a list of keys
- `d.values()` Returns a list of values
- `d.update(d1)` Updates d with key/values from d1, overriding values of d or adding new key/value pairs if necessary
- `del d[k]` Deletes the key/value pair
- `d.get(k, v)` Gets the key/value pair for k. If there is no value, it returns v.
- `d.popitem()` Deletes a random key/value pair from the dictionary (since dictionaries are unordered)

Functions

-
  ```
  def f(x): // Defines a named function
    BODY
  ```
  - Doing a = f is valid
- `f = (lambda x: x * x - 1)` is an example of a nameless Lambda function, which you typically use for functions that take other functions as arguments

Classes

-
  ```
  class c(a, b): // Defines a class c with parents a, b
     def m(self, x, y): // Class methods have a self argument
        BODY
  ```
  - Classes are objects too, so doing `d = c` is valid
- With multiple inheritance in Python, a class will look for a method that it does not already have via a depth-first search starting from its first parent. If it finds the method, then it does not look any further - meaning if there are any differences in methods between a class's ancestors, the first parent in this depth-first search wins

  -
    ```
    class c(a, b)
    class a(w, x)
    class b(x, z)
    class b(x, z)
    class x(q, r)
    class z(q, r)
    ```
  - Search Path: c -> a -> w -> x -> q -> r -> b -> z
- Classes have special members, typically denoted with "__" at the beginning:
  - `__dict__` is a dictionary that maps names (i.e. method names) to values (i.e. functions) in the class (so `c.__dict__[m]` is equivalent to doing c.m)
    - Since this dictionary is mutable, it is entirely possible to change a class at runtime by altering the mappings of the dictionary
  - `def __init__(self, a, b):` is the constructor of the class, and can be redefined
  - `def __repr__(self):` is a method that returns a string representation of the object
  - `def __str__(self):` is a method that returns a shorthand string representation of the object, and is typically called when the object is attempted to be converted into a string
  - `def __comp__(self, other):` is a method that returns -1 for self < other, 0 for equality, and 1 for self > other
  - `def __lt__(self, other):` is a more specific comparison operator, which is typically useful for floating points (since they can deal with NaN and so forth)
    - If `__lt__`, `__gt__` are defined, then they will be called instead of `__comp__` whenever possible
  - `__nonzero__(self):` handles checking if the class object is "nonzero" - i.e. when doing `if obj`
  - `def __hash__(self)` handles the hashing of the object (defaulting to using the object's address)
  - `def __add__(self, other):` handles addition with the object

Modules

- Modules in Python can be thought of similarly to source files in C++/C.
  - A module is typically a file that ends in `.py`
  - A module may contain the construct `if __name == '__main__':`, which is the code to be executed if the module is called at the top level (i.e. directly through the shell)
  - Modules are used through `import` statements. Importing will create a new namespace bound to the name of the module and will execute any unblocked code in that module file.

    -
      ```
      import math
      math.log // math namespace
      ```
- Both modules and classes are constructs used to organize source code, but modules, through packages, are more developer friendly since they allow for programmers to work on only specific parts

(files) of an overall package

  - Packages and modules tend to provide an API for other packages to use

  -
    ```
    /code directory/
    math.py
    net/
      connection/
          ...
      io/
          read.py
          write.py

    import math
    import net.io.read
    ```
  - Packages may contain an init.py file that performs any initializations for the entire package

- The environment variable PYTHONPATH is analagous to the shell $PATH variable and is where Python looks for modules to import
  - i.e. `PYTHONPATH = /home/pylib:/usr/share/python`
    - This will look first at /home/pylib and then at /usr/share/python, so if there are modules with the same name, whichever comes first is favored