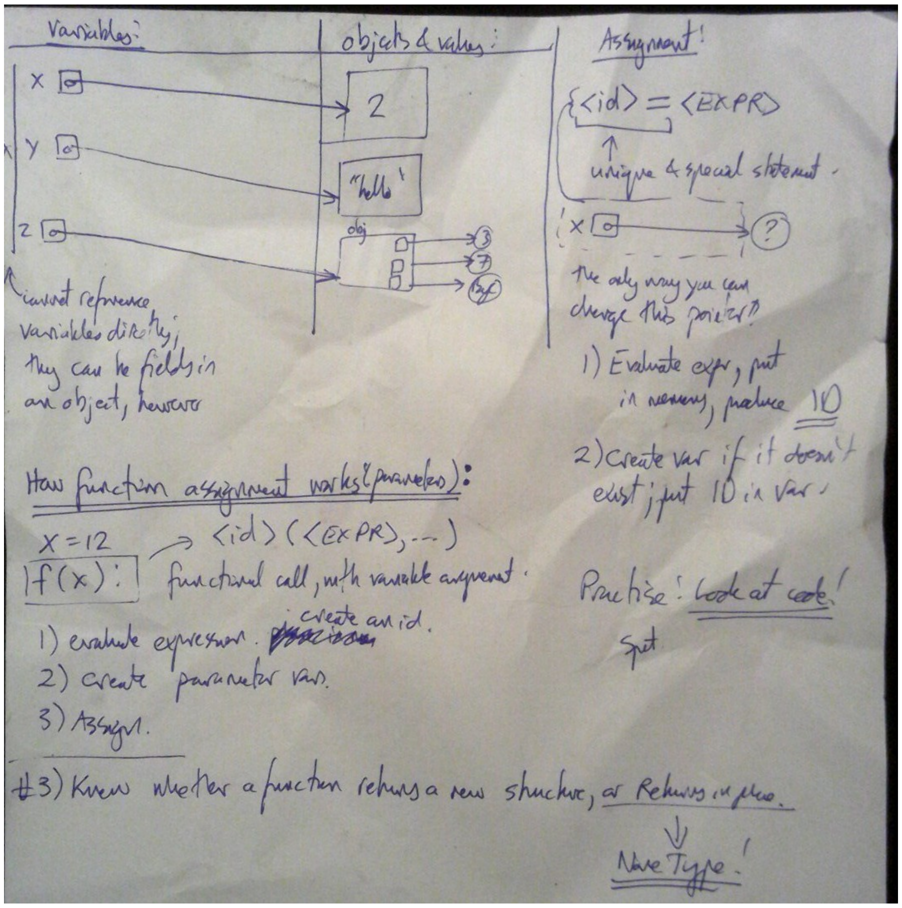
Python Review Notes:

Glossary:

- **Namespace:** aka Context; a system of naming to avoid name clashes; similar or common names are associated or prefixed with a particular namespace to avoid clashing. Three NS's include: *local* (function block), *global* (for the entire package or program) and *built-in* names.
    - The effect of turning a module into a package is that its name is used to implement a prefixed namespace for all functions in the package – to avoid clashing.
- Type System: One that assigns types to data objects in order to reduce logical and run time programming errors. Python is dynamically typed (I.e: type safety is checked at RT).
- Type: A category of object, that has encoded constraints and rules for how it is used.
- Scope: The context in which a particular binding can be used in a program. There is global and local scope (at the function level). Scopes are **separated, in order to protect the unwanted mutation/corruption of data. Global scope data can be passed to local scope through variable passing, or the usage of *global* symbol.**

Variables and Objects:

Gary's Model:



cannot reference variables directly; they can be fields in an object, however

unique & special statement.

The only way you can change this pointer!!

1) Evaluate expr, put in memory, produce ID

2) Create var if it doesn't exist; put ID in var.

How function assignment works (parameters):

x = 12

f(x):

→ <id>(<EXPR>, ...)

functional call, with variable argument.

1) evaluate expression. ~~[illegible]~~ create an id.

2) create parameter var.

3) Assign.

Practise! Look at code!

Spot

#3) Know whether a function returns a new structure, or Returns in place.

↓

None Type!

Python’s variable passing paradigm is technically pass-by-value. It is described (denotationally) as pass by shared object.

In python, everything is an object (there are no primitive and complex data types, like in C++ or Java).

An import**ant distinction in guessing behavior is whether an object is mutable or not.**

Numbers and strings are immutable; whereas lists and objects are generally not.

It behaves as follows (in terms of functions):

Actions in/with Functions:	**Mutable Object:**	**Immutable:**
Passing	A → [1,2,3] f(B ← A) A, B → [1,2,3]	A → “hello” [1] f(B ← A) B → “hello” [2] A separate copy is made.
Assigning	A → [1,2,3] f(B ← A) B → [q,x,c] A → [1,2,3] untouched, and B → [q,x,c] within function scope	A → “hello” f(B ← A) B → “bye” A → “hello” “”, “”

Alteration	A→ [1,2,3]
	f(B ← A)
	B.Append(4,5,6)
	Now A → [1,2,3,4,5,6]
		N/A for original or copy of immutable object!

Is vs ==: Equality checks to see that the internals of two objects are the same, whereas is:= “id(x) == id(y)”. Caveat: in python memory optimization, separate strings that are the same are stored in the same memory location! So X is Y with two identical (but separate) declarations will return true; there is only one instance (shared) in memory for both.

Bindings and Namespaces:

- When a script is interpreted, python creates objects and binds the names of variables to the data they reference.
- For a given script, a table of bindings in made.
- When different modules and packages are imported into a python script, there is always the risk of name collisions. How is this avoided? Via **namespaces**.
- Namespace: a grouping of bindings, prefaced with a unique name. It is done to avoid name collisions, and differentiate bindings that share the same name, but are independent of one another.

- **Practically:** For every namespace, there is is a separate bindings table. When your main script imports mod1, mod2, pack1...there is a separate bindings table for each of them.

Modules and Packages:

Path System:

- Python searches a number of locations in order to locate modules, when interpreting a new script.
- You can view these locations by running print(sys.path) in the interpreter.

Module:

- This is a file, that contains functions, classes variable definitions. The name of the module is the name of the file + ".py"
- Object notation is used to access variables and functions in modules.
- Unless specified with a magic/system variable, the name of the module file is used to create a namespace for the definitions inside.
- Modules can be run on their own; a main method is executed in a global code block by the following: *if "__name__" == "__main__":*
- To run a module in another module, we **import** it. Some importing examples are below:

Import Syntax (from Chris Yeh's Blog):

`import <package>`

1. `import <module>`
2. `from <package> import <module or subpackage or object>`
3. `from <module> import <object>`
4. `from <module> import <object>`

Examples:

import mod1
mod1.func1(arg) #all functions need to ref the mod name to run:

from mod1 import func1, func2 #import functions directly
func1() #we can just run directly

import mod1 as m1 #gives us a reference short hand

from mod1 import func1 as f1 #short hand for directly accessed functions.

import mod1
func = mod1.funct1 #treating imported function as first class.

- Module Search Path: The Full Search Path is as follows:

  1) modules in the Python Standard Library (e.g. `math`, `os`)

  2) modules or packages in a directory specified by `sys.path`:

  1. If the Python interpreter is run interactively:
     - `sys.path[0]` is the empty string `''`. This tells Python to search the current working directory from which you launched the interpreter, i.e. the output of `pwd` on Unix systems.

     If we run a script with `python <script>.py`:

     - `sys.path[0]` is the path to `<script>.py`
  2. directories in the `PYTHONPATH` environment variable
  3. default `sys.path` locations

  Import Types:

  - Absolute: An import that references the location of a resource from the top of the project folder. These can even be used to run resources deep in the project tree.
    - Benefits: One point of reference, no relative confusion. Simple to understand.
    - Drawbacks: For deeply nested scripts, long name imports become tedious, even for modules that are adjacent to one another.
  - Relative: An import that uses the scripts position in the tree as a point of reference.
    - Benefits: Less long
  - Implicit: written as if the script is located in one of the sys.path locations. These are banned in Python3 (security issues)
  - Explicit: written from the relative position of the script doing the import.
    - Note: You cannot import things from the parent directory of the main script. You either use absolute imports, or refactor your project.

- Dir(modname) function lists what a module defines (functions, variables, etc). Without argument, it lists what a user has defined in an interpreted session.

Packages:

- A collection of modules, grouped together in a directory structure.
- “dots” in import statements indicate increasing subdirectories.
- “.”: current directory, “..” one level up, “...” two levels up, etc.
- Regular Packages(Py<3.3): Each subdirectory includes an __init__.py file; they are mandatory and force python to consider directories as packages.
- Name-space Packages (Py>=3.3): __init__.py is not required. Python will auto-infer packages, these auto-generated ones are called *namespace packages*.

Meta-Options of Python and Modules:

- dir(arg): No argument: Return the list of names in the current local scope. At the interpreter level, it will just be at global scope level. With an argument, return “relevent” information for the object/type.
    - Dir(modname): will return all of the bindable names in a given module.
- help(module): Returns a summary of a module; including global vars, functions and there comment strings (above)
- Globals(): Return dictionary of current global system table.
- Vars(): Returns the __dict__ attribute
- id(): returns object pointer address.
- Underscore Conventions:
    - _var: Convention: A suggested internal/hidden variable or method; it can still be called if you wish.
    - Var_: Overrides a keyword and allows you to use it as a name for a variable.
    - __var: For “name mangling”
    - __var__: “Dunders”: Reserved for python magic methods – avoid using for your naming schemes.

Built-In Functional Programming Concepts:

- **Filter(test function, Iterable) → FilterObject[elements that tested true].**
- **Map(function, Iterable) → MapObj[all elements with applied operations]**
- **Lambda x: <statement> : allows you to construct an anonymous function**

Things left to do:

- implement basic class based things: polymorphism, inheritance.

Objects and Classes in Python:

Object Oriented Programming:

- Idea: structure data into *objects*: data structures that contain information and functional methods in which to operate on the said data.
- Philosophy:
    - Minimal Code re-writes
    - use data structure and OO principles to model problem structures, create clean solutions with boundaries between data, types and methods.
- Objects: Have attributes and methods.
    - Access Control: Private, Public.
        - A private attribute can only be changed with a class method.
- Class: Is a formal definition for the attributes and methods of an object.
    - Contains: Object Constructor (with instance attributes), “global object” class attributes, object methods (class and instance level.
- Basic Principles:
    - Instantiation: When we use the class definition to construct an object. This is called an “instance” or “instance of” <class>.
    - Inheritence: Using details from an existing class to define a new one.
    - Encapsulation: Hide private details of a class from other classes, agents, etc.
    - Polymorphism: Use common operations or methods for different objects and data types. Example: “+” can add numbers, or join strings together.

Class Template:

```
class ClassName(classA,classB,…):
  attr1 = “” #public Class Attribute
  __attr2 = “” #private Class Attribute
  ...


  def __init__(self,arg1,arg2,...):
    super().__init__() #call to first superclass constructor; determined by MRO() order.
    Self.par1 = arg1 #public instance attribute.
    Self.__par2 = arg2 #private instance attribute.
    ...
```

Python Objects and Classes:

- Names and Namespaces:
  - A name is a lable that binds to a piece of data (object in memory). Recall that all data in Python is an object.
  - We reference data by name, in our program.
  - Namespaces are groups of names, for a given context. Namespaces allow different parts of a program to use the same name bindings, without reference clobbering occuring.
  - Namespaces (as a concept and implementation detail) also overlap with scope. Some common spaces are:
    - Built In (Top-Global NS/Scope), Global (the top level of our script), and Functional (inside a function).
  - In Python, Classes have their own namespaces.
- An object is a data structure with methods (functions) explicitly bound to it.
- A class, when instantiated, has its own local namespace for the name bindings used in methods.
- "_ _" Attributes: All classes have special attributes. Some include:
  - __doc__: Our doc string that describes our class
  - __dict__: All attributes for a class and object are encoded in a dictionary.
  - ???

Types of Objects:

- **Class Object:** Created after class definition interpreted by Python Interpreter.
  - Class attributes, Constructor, and __special__ attributes stored here.
- **Instance Object:** created by a call to a class object constructor.
  - Contains all instance attributes; can also access class attributes directly as well (if not private).
- **Function Object:** Accessed via ClassName; the bound name is a class attribute – but the function definition is not in the class.
- **Method Object:** This is a function where the body is defined in the class *(understand bound unbound and static methods to complete this)**

After Instatiating an Object:

- functions defined in the class (but not stored in an attribute) are bound to instance objects. These are treated as attributes that are passed arguments; the objectname.funname is just stores an ID to a particular function body, somewhere in memory

- Calling an object (bound) method: The first prototype parameter is (self), which references the object the method was called on. Python makes this the first parameter for a bound method, automatically.

Delete Attributes and Objects:

- Usage: *del object.method* or *del object.attr.* These are only removed from a particular instance.
- You can also delete methods and attributes from a Class.
- The same can be done for adding functions and attributes (to particular instances, or classes).

Python Inheritence:

- There are *base and derived classes*.
- The derived class inherits all class properties and methods. **It does not inherit the constructor.**
- A derived class must define its own constructor; it can reuse the base classes with the **super().__init__()** call.
- A derived class can use its own methods, or call those of the base class.
- Overriding: A derived class can define its own methods and attributes. If they have the same name as the base class, the derived class overrides them (these references stay in the class, and don't point to the base class with the same names).
- Two useful methods:
    - isInstance(object,classname)
    - isSubClass(derived,base)

Multiple Inheritence:

- Unlike (old) Java, Python allows multiple inheritence. This means that a derived class can have multiple base classes, and it inherits all of the methods and properties.
    - What if two base classes have methods that share names? Method Resolution Order() solves this ambiguity: the order of the base classes in the header definition define the order of presidence for multiple methods that share the same name.
        - MRO(class) can be used to view an objects order.