1 What to Expect

1.1 Major Themes

• Symbol Tables
  – A symbol table for the function exists while the function is executing (and goes away after the return)
  – There is a symbol table in each module, class definition, and instance object
  – There is a global or main symbol table when you run in the interpreter

• Important Computational Constructs As Exemplified by Python Language Features
  – Variables and assignment statements
  – Functions (parameters as input, return values as output)
  – Conditional Statements (if, elif, else)
  – Loops (for)
  – Data Types (ints, floats, bools, lists, strings, dictionaries)
  – Class Definitions and Instance Objects

• Top-Down Design
  – Break down the problem by focusing on the verbs – break it into steps that must be accomplished
– Pass data from one function to another

• Object-Oriented Design

– Break down the problem by focusing on the nouns – think about objects, their data, and their responsibilities

2 Symbol Tables

A symbol table maps a name to a value. Values that are numbers, booleans, or strings are drawn in the table (these are called immutable objects in Python parlance – they are called this because you cannot change just a part of a number, boolean, or string). Values that are class instances, classes, modules, functions, lists, dictionaries, or any other mutable objects are drawn outside the table – the entry in the table is an arrow to the value.

Symbol tables are drawn for modules, class instances, classes, and functions. We label the table based on the type of object containing the symbol table or, for functions, the name of the function that is executing.

• Functions: When a function is executing, it has a symbol table. To label this symbol table, we write “a” at the top.

• Interpreter: When running in the interpreter, there is a “global” or “main” symbol table that stores all the symbols used at the command line.

• Modules: Each module contains a symbol table. For the tree module we write “tree <module>” at the top of the table.

• Classes: Each class definition contains a symbol table. This symbol table is the template for any instance of that class. For a Person class, we write “Person <class>” at the top of the table.

• Class Instance Objects: Each class instance contains a symbol table. The symbol table for an instance of the Person class is labeled “<Person>” to indicate the type of object that contains this symbol table is a “<Person>”.

Entries in a class instance object symbol tables are called attributes. Attributes fall into two categories:
- methods: functions defined in classes are called methods. A method’s entry in an object’s symbol table is an arrow to the corresponding entry in the class’s symbol table. For Shape instance objects, these include \_init\_ and draw.

- fields: Another name for a field is “instance variable”. These are entries for the data that the object needs to contain. For Shape instance objects, these include distance and angle.

We use the dot notation to access symbols in “other” symbol tables. For example, if we are executing in a main function

```python
import turtle
turtle.right(90)
```

Then the first line adds a symbol to the main symbol table. That symbol is named turtle and it refers to a module. To access the function right in the turtle module’s symbol table, we use the dot notation.

### 3 Important Computational Constructs As Exemplified by Python Language Features

#### 3.1 Modules

A module is simply a collection of executable statements, function definitions, and class definitions. Modules are used to organize code. When we program using a procedural design, we use modules to organize our functions. When we program using an object-oriented design, we use modules to organize our classes.

The difference between

```python
import turtle
```

and

```python
from turtle import *
```

3
is related to symbol tables. If you are running in the interpreter, and type (1), then a single entry will be added to the main/global symbol table. That entry will be named “turtle” and its value will point to a module, which contains a symbol table with all of the turtle functions as its entries. If you type (2), then the turtle module will be loaded and the contents of its symbol table will be copied into the main symbol table. So the main symbol table will contain an entry for each of the turtle functions.

3.2 Assignment Statement

Assignment statements are used to store values in symbol tables. They are evaluated in two steps.

1. Evaluate the right hand side (this may involve calling a function, creating an instance object, evaluating an expression, or some combination of these operations)

2. Update the symbol table. If there is already an entry for the symbol on the left hand side of the assignment statement, then just update its value. Otherwise, add a new row to the table.

For local variables, we need to look at only one symbol table. Suppose we have a main symbol table with no entries in it.

We execute the line

```python
a = 'hi'
```

and a new entry is added for a.

If we then execute the lines

```python
1 a = Person()
2 a.name = 'stephanie'
```

then

- Executing Line 1:

  1. Evaluate R.H.S.: a new Person instance object is created
  2. Update symbol table: The entry for a is updated to have an arrow to the `<Person>` object (which contains its own symbol table).
• Executing Line 2:

1. Evaluate R.H.S.: already done - the r.h.s. is the string literal 'stephanie'
2. Update symbol table: the lefthand side uses the dot notation, so we follow the arrow from a to the Person and add an entry to its symbol table with the name name and value 'stephanie' <str>.

Note: Ahhh!! This means that the code in the main function or interpreter is able to modify the symbol table for an object. Yes. But that isn’t good style. I should instead use a method for the Person to set the name field. I just used this as a quick example.

3.3 Boolean and Mathematical Operations

See lecture 6 for the boolean operations and lecture 4 for integer and float operations.

3.4 Functions

A function is a series of statements which returns some value to a caller. It can accept input via its parameters. For example

```python
def f1(a):
    return a + 1
```

is a function named f1 with one parameter named a. It has en explicit return statement that returns the sum of a and 1. In terms of symbol tables, a new symbol table is created when the function is called. There is an entry for a which contains the value passed in by the caller.

Not all functions need to have explicit return statements. For example,

```python
def f2(a):
    print "In f2, we compute " + str(a + 2)
```

has an implicit

```python
return None
```

at the end.

If we call these two functions with the following code:
\[ c = 5 \]
\[ b = f_1(c) \]
\[ \text{print } b \]
\[ d = f_2(c) \]
\[ \text{print } d \]

The following is printed to the screen:

6
In f2, we compute 7
None

It is also possible to define default values for function parameters, e.g.

\[ \text{def } f_3(x, y=0): \]
\[ \quad \text{return } x + 2*y \]

Here, \( y \) has a default value of 0. We can call \( f_3 \) with or without an argument (value) for \( y \), e.g.

\[ \text{print } f_3(1) \]
\[ \text{print } f_3(1, 2) \]

Python also allows us to use the \textit{keyword} notation for passing arguments. e.g.

\[ \text{print } f_3(x=1) \]
\[ \text{print } f_3(x=1, y=2) \]
\[ \text{print } f_3(y=2, x=1) \]

This allows us a little more flexibility (e.g. the order doesn’t matter) than the \textit{positional} notation (in which the first argument supplies the value for the first parameter, the second for the second, etc.). This notation also makes the code more readable because we don’t need to remember whether \( x \) or \( y \) came first – we can just read it.

### 3.5 Conditional Statements

If-elif-else statements are conditional statements. See lecture 6 if you would like a review.

### 3.6 File I/O

We open the systemE.txt file to read it like this:
fn = 'systemE.txt'
f = file(fn, 'r')
lines = f.readlines()
f.close()

Now lines is a list of strings, with one string for each line of the file.
For more about file I/O, see Lecture 20.

### 3.7 Sequence Types

#### 3.7.1 Strings

A string is sequence of characters delimited by single or double quotes. An example of a string literal is

'hi'

An example of a variable containing a string is this:

h = 'hi'

h is the symbol and 'hi' is its value. The type is `<str>`.

There are many operations that can be applied to strings including slicing them to get substrings out, e.g.

```python
s = 'abcdefg'
print s[0:]  # which prints abcdefg
print s[1:]  # which prints bcddefg
print s[2]   # which prints c
```

Strings are immutable, which means parts of string can’t be altered (which is one difference strings have from lists). We **cannot** change the string 'abcdefg' to 'abCdefg' using the following code:

```python
s[2] = 'C'
```

Instead, we must create a new string, e.g.

```python
s = s[0:2] + 'C' + s[3:]
```

or

```python
s = s.replace('c', 'C')
```
See Lecture 12 for more about strings.

### 3.7.2 Lists

A list is a sequence of objects (where object refers to *any* data type, including ints, bools, other lists, and instance objects) indexed by position (beginning at zero). Lists are mutable, which means we can change just one entry of the list, or grow or shrink the list.

A list is delimited by square brackets, e.g.

```python
  c = [0.1, 0.0, 0.5]
```

Elements are accessed using square brackets containing the index of the element, e.g.

```python
  print c[0]  # prints the first component
```

See Lecture 13 for more about lists.

### 3.7.3 Tuples

A tuple is like a list, but it immutable. Tuples are useful for short lists whose entries do not need to be altered once the tuple has been created. For example, turtle positions and colors are often represented as tuples. Tuples are “lighter weight” objects than lists. In general it is a good idea to use the lightest-weight type that will do that job (why bother giving it methods it doesn’t need? it just wastes space).

A tuple is delimited by parentheses, e.g.

```python
  c = (0.1, 0.0, 0.5)
```

Elements are accessed using square brackets containing the index of the element, e.g.

```python
  print c[0]  # prints the first component
```

### 3.8 Dictionaries

A dictionary is a sequence of objects (where object refers to *any* data type, including ints, bools, other lists, and instance objects) accessed by a string key. Dictionaries are mutable, which means we can change a single entry, add entries, and remove entries. Dictionaries are used by python to store the symbol tables.
We use a dictionary when we find it more convenient to name each value rather than to just memorize its position. In our collages, it would have been better to use a dictionary or a class object rather than a list to store information about each image in the collage.

A tuple is delimited by curly braces, e.g.

```
c = {}  # make an empty dictionary
c['red'] = 0.1  # add an entry whose key is red and value is 0.1
c['green'] = 0.0  # add an entry whose key is green and value is 0.0
c['blue'] = 0.5  # add an entry whose key is blue and value is 0.5
```

Elements are accessed using square brackets containing the key to the element, e.g.

```
print c['red']  # prints the red component
```

An empty dictionary can also be created using object-oriented notation. The class name for the dictionary is `dict` and a new instance of a dictionary can be created using the code:

```
c = dict()
```

### 3.9 For loops and Ranges

Range is a function that takes one, two, or three integers as input and returns a list of integers, e.g.

```
>>> range(4)  # return list from 0 to 4-1
[0, 1, 2, 3]
>>> range(1, 4)  # return list from 1 to 4-1
[1, 2, 3]
>>> range(1, 8, 2)  # return list from 1 to 4-1 with step of 2
[1, 3, 5, 7]
```

For loops use the syntax:

```
for c in lst:
    # put body of code here
```

where `c` is called the loop control variable and `lst` is a list. The first time through the loop, `c` has the value of the first item in `lst`, the second time through the loop, `c` has the value of the second item in `lst`, etc. The loop body is executed length(`lst`) times.

The list in a loop can contain anything (integers, instance objects, etc.). It is common to name the loop control variable `i` if the list is of indexes (e.g. `[0, 1, 2, 3]`).

Consider the following example
```python
class Coin:
    def __init__(self, value):
        self.value = value

    def getValue(self):
        return self.value

if __name__ == '__main__':
    coins = []
    for i in range(5):
        coins.append(Coin(25))
    for i in range(len(coins)):
        c = coins[i]
        print(c.getValue())
    for c in coins:
        print(c.getValue())

Lines 12 and 13 together are equivalent to line 15.

4 Design

4.1 Hierarchical Design

Hierarchical (also called top-down) design is process-oriented. We break each task into smaller tasks.

Functions divide a problem into parts by looking at the actions required to achieve a solution. The keys to subdividing a problem into functional parts are:

- Identifying the steps required by the solution and looking for duplication
- Identifying the information that needs to be passed around between functions
- Identifying the input/output characteristics of each function
- Dividing the problem sufficiently so that each function is easy to write

Here is an algorithm for designing and implementing a program using a procedural design:

1. Define the task using natural language (understand the problem)
2. Identify the inputs and outputs of the task

3. Recursively break down the problem into smaller steps – this is called hierarchical decomposition

4. Organize the steps, noting the input and output requirements of each step

5. Identify what the individual functions should be, noting where in the steps there are similar input and output requirements. The textbook points out that functions should do “one and only one thing”. This is a good rule of thumb.

6. Generate some intermediate representation of the algorithm
   - Pictures
   - Flow chart
   - Pseudo-code style comments

7. Write code

8. Verify and test the code

### 4.2 Object-Oriented Design and Programming

An object-oriented approach to design looks at the problem differently. Instead of breaking down the problem into a series of steps, the problem domain is divided by the objects, or nouns that represent parts of the problem description. Both actions and data are then attributed to the critical objects.

- The objects for a particular problem represent the nouns
- The methods of the objects represent verbs in the problem description
- Adverbs and adjectives represent the data, or parameters required for the methods or objects.

A class defines a new object type. It can also be thought of as a template for a new instance object. E.g. the Ball class provides a template for any new Ball objects that are created.

A method is a function that is defined inside a class body.

See Lectures 23 and 25 for more about classes, objects, and symbol tables. See Lecture 27 for more about inheritance.
4.2.1 Terminology

Consider the code:

```python
class Root:
    def __init__(self, name):
        self.name = name

    def getName(self):
        return self.name

class Potato(Root):
    def __init__(self):
        Root.__init__(self, 'potato')

if __name__ == '__main__':
    p = Potato()
    print p.getName()
```

Lines 1 through 6 form a **class definition** for the Root class. Lines 2 through 6 are in the definition body. They define two methods. A **method** is a function defined within a class definition. The `__init__` method is a **special method** because it is one that Python pays special attention to.

Lines 8 through 10 for a class definition for the Potato class. The Potato class is **derived** from the Root class. The Potato class is a **child** or **subclass** of Root. Root is a **parent** or **superclass**. The Potato class **inherits** all of the methods from the Root class. The Potato class **overrides** the Root class’s `__init__` function with its own (lines 9 and 10).

Suppose this code is in a module named roots.py and it is executed from the Terminal command line using the command

```
python roots.py
```

Then it will be run with a `__name__` of '__main__'. The class definitions will be loaded into memory (i.e. one symbol table will be created for each). The lines 13 and 14 will be executed.

Line 13 creates an instance of the Potato class. We call it an **instance** or **instance object** or **object**. The type associated with it is `<Potato>`. When Python executes the expression `Classname()` , it follows a in three-step process:

1. Python creates the object, which contains a symbol table. That symbol table is created using the class definition’s symbol table as a template.
2. Python looks for an \_init\_ method. If it is there, then it is called on the newly created object.

3. Python returns a reference to the newly created object

In this case, the Potato has an \_init\_ method. It calls the \_init\_ method on its parent class using the notation \texttt{Classname.methodName(self, any other arguments)}. This method adds an entry to the instance object’s symbol table. This entry is called a \textbf{field} (i.e. \texttt{name} is a field).

The end result is the \texttt{Classname()} call is replaced by a reference to the new object.

Line 14 then calls the \texttt{getName} method on the Potato \texttt{p}. This method was inherited from the Root class.

Notice that line 14 as written uses the notation \texttt{instance.methodName(other arguments)}. We could have written it using the \texttt{Classname.methodName(instance, other arguments)} notation. It is equivalent, i.e. we could have said

\begin{verbatim}
    print Potato.getName(p)
\end{verbatim}

However, it is \textbf{not good style} to use the \texttt{Classname.methodName} notation unless it is \textbf{absolutely necessary}. In line 14, it is better to use the \texttt{instance.methodName} notation. In line 10, we were forced to use the \texttt{Classname.methodName} notation because we wanted to call the Root class’s version of \_init\_. In other words, we were making Python call a method other than the one in the instance’s symbol table.

\subsection*{4.3 Comparing the Two}

I generally use hierarchical design for highly mathematical code and object-oriented design for everything else. But it depends upon which language I am using. To write good Java code, I really should use an object-oriented design. Python is amenable to either. C is amenable to hierarchical, and C++ is amenable to either (it was designed to support OO but it retained all of the features of C).

\section*{5 Parsing}

Parsing is a process in which we analyze a string of characters by looking at them one at a time, and build up information about their content. In Lecture 30, we talking about parsing
parameters for interpreter symbols.

6 Recursion

Recursion is used to “divide and conquer” a problem. We used to do computation on lists and strings and to draw trees.

See lectures 32 and 35.