Objectives

- Review: string format
- Functions
- Import
- Intro to design patterns
- Definite loops

Review: Formatting

- What data type does string formatting give you?
  - For example, what data type would `"%6.2f" % expense` give back?
- What is the format specifier's *code* for ints? Floats? Strings?
- What is the format specifier for right-justifying a number within 10 spaces that displays 3 decimals?

Parts of an Algorithm

- Input, Output
- Primitive operations
  - What data you have, what you can do to the data
- Naming
  - Identify things we're using
- Sequence of operations
- Conditionals
  - Handle special cases
- Repetition/Loops
- Subroutines
  - **Call**, reuse similar techniques

FUNCTIONS

Using Built-in Functions

- Functions perform some task
  - May take **arguments/parameters**
  - May **return** a value that can be used in assignment
- Syntax
  - `func_name(arg0, arg1, …, argn)`

Argument/parameter list

- Depending on the function, the arguments may or may not be required
  - [ ] indicate an optional argument
- Semantics: depend on the function

Example Built-in Functions

Known as function's "*signature*"

Template for how to "call" function

Optional argument

- `raw_input([prompt])`
  - If prompt is given as an argument, prints the prompt without a newline/carriage return
  - If no prompt, just waits for user's input
  - **Returns** user's input (up to "enter") as a **string**
- `input([prompt])`
  - Similar to `raw_input` but returns a **number**

More Examples of Built-in Functions

- **round(x[,n])**
  - Round the `float` **x** to **n** digits after the decimal point
  - If no **n**, round to nearest `int`
- **abs(x)**
  - Returns the absolute value of **x**
- **type(x)**
  - Return the type of **x**
- **pow(x, y)**
  - Returns x^y

Using Functions

- Example use: Alternative to Exponentiation
  - Goal: compute -3^2
  - Python alternatives:
    - **pow**(-3, 2)
    - (-3) ** 2
- Typically, we use functions in assignment statements
  - Function does something
  - We save the result of function in a variable

Python Libraries

- Beyond built-in functions, Python has a rich **library** of functions and definitions available
  - The library is broken into **modules**
  - A ***module*** is a file containing Python definitions and statements
- Example modules
  - `math` -- useful math functions
  - `os` -- useful OS functions
  - `network` -- useful networking functions

Example Library: Math Module

- Defines constants (variables) for `pi` (i.e., π) and **e**
  - These values never change, i.e., are ***constants***
  - Remember: *we* name constants with all caps
- Defines functions such as
  - **ceil(x)**
    - Return the ceiling of `x` as a `float`
  - **exp(x)**
    - Return e raised to the power of `x`
  - **sqrt(x)**
    - Return the square root of `x`

Using Python Libraries

- To use the definitions in a module, you must first **import** the module
  - Example: to use the **math** module's definitions, use the the import statement: **import math**
  - Typically import statements are at *top* of program
- To find out what a module contains, use the **help** function
  - Example:

```
import math
help(math)
```


Using Definitions from Modules

- Prepend constant or function with "**modulename**."
  - Examples for constants:
    - `math.pi`
    - `math.e`
  - Examples for functions:
    - `math.sqrt`
- Practice
  - How would we write the expression $e^{i\pi} + 1$ in Python?

Alternative Import Statements

```
from <module> import <defn_name>
```

- Examples:
  - `from math import *`
    - Means "import everything from the math module"
  - `from math import pi`
    - Means "import pi from the math module"
- With this `import` statement, don't need to prepend module name before using
  - Example: e**(1j*pi) + 1

Python Libraries

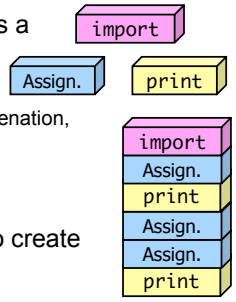
- Python has a rich library of functions and definitions available for your use
  - The library is broken into **modules**
  - A **module** is a file containing Python definitions and statements
- **Benefits** of functions/definitions in modules
  - Don't need to rewrite someone else's code
  - If it's in a module, it is very efficient (in terms of computation speed and memory usage)

Finding Modules To Use

- How do I know if some code that I want already exists?
  - Python Library Reference:
  - `http://docs.python.org/lib/lib.html`
- For example, `string` module has functions/ constants for manipulating strings
- For the most part, to practice, in the beginning you will write most of your code from scratch

Programming Building Blocks

- Each type of statement is a building block
  - Initialization/Assignment
    - Arithmetic, string concatenation, functions
  - Print
  - Import
- We can combine them to create more complex programs
  - Solutions to problems

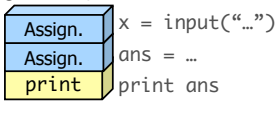


Design Patterns

- General, repeatable solution to a commonly occurring problem in software design
  - Template for solution

Design Patterns

- General, repeatable solution to a commonly occurring problem in software design
  - Template for solution
- Example (Standard Algorithm)
  - Get input from user
  - Do some computation
  - Display output

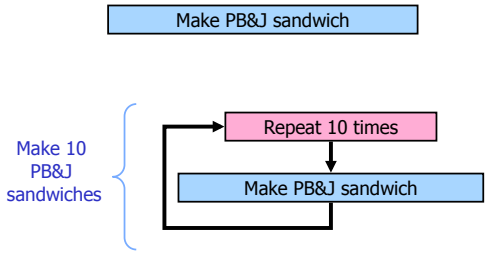


- Learn new building block, new design pattern

Parts of an Algorithm

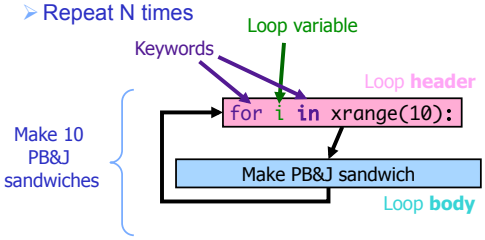
- Input, Output
- Primitive operations
  - What data you have, what you can do to the data
- Naming
  - Identify things we're using
- Sequence of operations
- Conditionals
  - Handle special cases
- Repetition/Loops
- Subroutines
  - Call, reuse similar techniques

Looping/Repetition



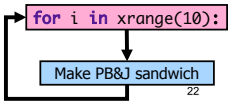
The `for` Loop

- Use when know how many times loop will execute
  - Repeat N times



What Goes in the Loop Body?

- Make PB&J Sandwich
  - Gather materials (bread, PB, J, knives, plate)
  - Open bread
  - Put 2 pieces of bread on plate
  - Spread PB on one side of one slice
  - Spread Jelly on one side of one slice
  - Place PB-side facedown on Jelly-side of bread
  - Close bread
  - Clean knife
  - Put away materials



What Goes in the Loop Body?

- Make PB&J Sandwich
  - Gather materials (bread, PB, J, knives, plate)
  - Open bread

Initialization

Loop Body

  - Put 2 pieces of bread on plate
  - Spread PB on one side of one slice
  - Spread Jelly on one side of one slice
  - Place PB-side facedown on Jelly-side of bread

  - Close bread
  - Clean knife
  - Put away materials

Finalization

Using the `for` Loop

- Use when know how many times loop will execute
  - Repeat N times

Times to repeat

```
for i in xrange(10):
    statement_1
    statement_2
    …
    statement_n
```

"Body" of `for` loop
- Gets repeated
- Note indentation

Using the for Loop

- If only ***one*** statement to repeat

```
for i in xrange(5): print "Hello!"
```


Analyzing xrange()

- **xrange** is a built-in function
- What does **xrange** do, exactly?
  - Simulate on paper

xrange([start,] stop[, step])

- What does the above signature mean?

xrange([start,] stop[, step])

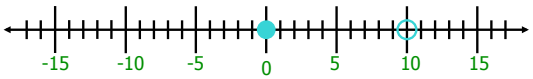
- 1 argument: xrange(stop)
- 2 arguments: xrange(start, stop)
- 3 arguments: xrange(start, stop, step)

xrange([start,] stop[, step])

- 1 argument: xrange(stop)
  - Defaults: start = 0, step = 1
  - Iterates from 0 to stop-1 with step size=1
- 2 arguments: xrange(start, stop)
  - Default: step = 1
  - Iterates from start to stop-1 with step size=1
- 3 arguments: xrange(start, stop, step)
  - Iterates from start to stop-1 with step size=step

xrange()

- xrange is a built-in **function**
  - 1 argument: xrange(stop)
  - 2 arguments: xrange(start, stop)
  - 3 arguments: xrange(start, stop, step)

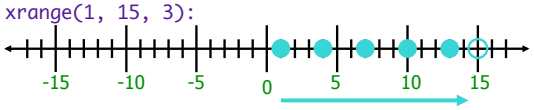


[start, stop)

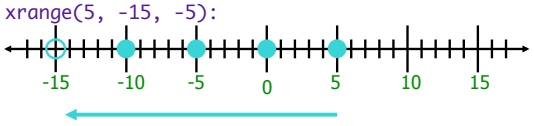
xrange(10)
xrange(0,10)
xrange(0,10,1)

xrange()

xrange(1, 15, 3):



xrange(5, -15, -5):



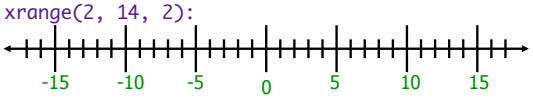
new_for.py

Practice

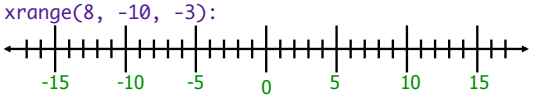
Place these: ● ○

Which direction?

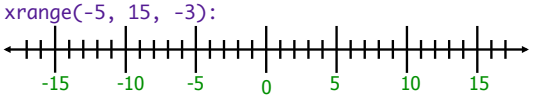
xrange(2, 14, 2):



xrange(8, -10, -3):

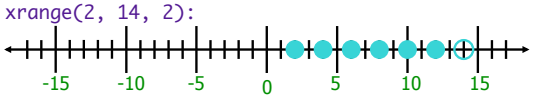


xrange(-5, 15, -3):

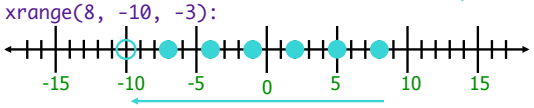


Practice Solution

xrange(2, 14, 2):

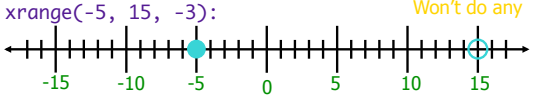


xrange(8, -10, -3):



Won't do any

xrange(-5, 15, -3):



Programming Practice

- Add 5 numbers, inputted by the user
  - After implementing, simulate running on computer

sum5.py