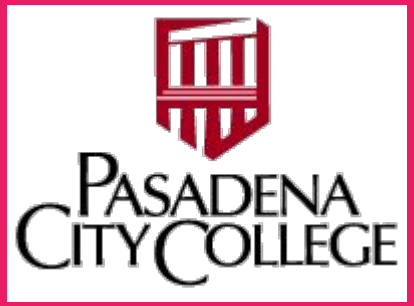


CIS 112

Intro to Programming Using Python

Module 1 Part 2

Agenda for the Day!

- Topics
- Syllabus
- Projects
  - Review of CIS 012

Functions

Modularity and Scalability

- Programs can quickly grow to be large and unwieldy, necessitating the need for modularity and segmentation to make them comprehensible.
  - Functions allow us to bundle self-contained fragments of code into reusable, referenceable code blocks
  - Classes (more on those!) allow us to create templates for items that bundle data and methods together and occur throughout our programs
  - Modules (more on those!) allow us to segment supportive functions in other files and reference as needed.

Reusable Blocks of Code

- In our study of control flow we've introduced the concept of code blocks, or multiple lines of code we want to link together to run in concert
- We also introduced the idea of controlling when these code blocks are invoked
  - Previously this was done automatically, as the result of an external condition
    - If statements
    - Loops
- But what if we want to create a code block that we can invoke on-demand?
- Introducing: functions
  - With functions you define a special variable that is associated with an executable block of code you can then reference and invoke on demand whenever you want it to run!
- We've already been using many base python functions! (print, input, int etc.) and Python libraries have countless more
- But we can also define our own custom functions too!

Putting the Fun in Function!

- write reusable pieces/chunks of code, called **functions**
- functions are not run in a program until they are "**called**" or "**invoked**" in a program
- function characteristics:
  - has a **name**
  - has **parameters** (0 or more)
  - has a **docstring** (optional but recommended)
  - has a **body**
  - **returns** something

Function Structure and Components

keyword: `def`

name: `is_even`

parameters or arguments: `( i ):`

specification, docstring:

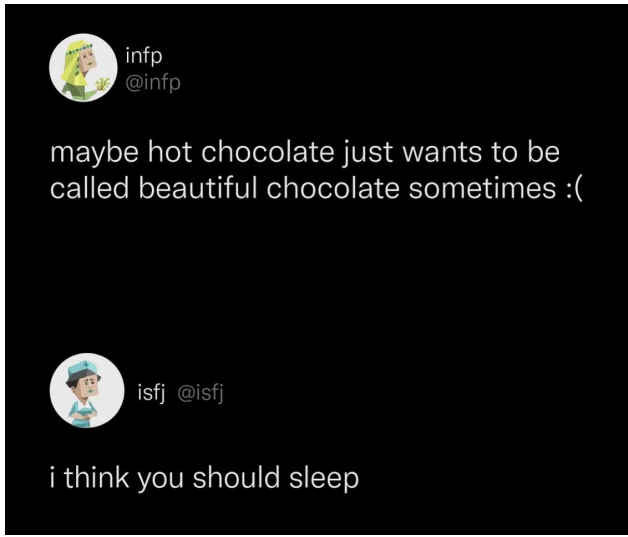
```
"""
Input: i, a positive int
Returns True if i is even, otherwise False
"""
```

body:

```
print("inside is_even")
return i%2 == 0
```

later in the code, you call the function using its name and values for parameters

```
is_even(3)
```



Functional Inputs

- Code blocks do not necessarily need inputs, and can run independently once invoked simply by referencing the function with an appended parentheses
- However most functions will take inputs
  - This should be quite intuitive for us as we've been using inputs from virtually day 1 to import data into a program
  - Functions work in much the same way, allowing the user to pass in specific values for placeholder functions
  - And just like inputs, there's no theoretical limit to how many inputs we can take

Functional Outputs

Jessie
@mommajessiec

People who camp are like, “But camping is so much fun!” and then tell you a story about how they had to fight a raccoon at 2 AM.

- Our overall program will often hand off, or ‘subcontract’ a process to a function and then take the result and continue on with the broader function
- We therefore need to talk about function outputs
  - The one we’ve seen already: print, outputs a final string to terminal
  - Problem is that output isn’t usable. We can’t pass it on to the rest of the program
  - To use it we need to instead use the **return** statement which exports the result and makes it visible externally

Nesting of functions

- Much like other control flow elements functions can be embedded within other control flow elements (e.g., other functions, loops etc.)
- In turn functions can also include their own nested control flow elements.

So why Functions anyways?

- Repeatability
  - If you know you'll be reusing a set of functionality over and over again, it is often easiest to simply standardize the code once, and then reference as needed.
- Modularity
  - Programs can, very quickly, get very complex!
  - It is essential, for an ultimate application's success, to embrace modularity
    - Functions allow you to isolate and finalize discrete blocks of programming, you can then string together for broader applications
      - Micro-service type architecture.

Strings!

Referencing and indexing

- Indexing and referencing
  - As a reminder, strings are a datatype! Key features of strings:
    - They are Immutable but referentiable
      - We cannot directly iterate, but can reference via index
  - Ranges
    - We can use colons to index entire ranges of values!
  - Negative Indices
    - We can use a negative index to reference from the end
  - Incrementalization
    - We can also choose (much like with range!) our incrementalization step

String Methods

- The string datatype has a number of associated ‘methods’ or built-in functions
  - More on that when we get to Object-oriented programming!
- Find:
  - Can search a string for the presence of something
- Count:
  - Can count the number of occurrences
- Comparisons
  - ==, >, in
- Checks
  - A series of string, specific logic checks
- Create/Convert
  - Can make new strings with evolved string

Simon Holland
@simoncholland

Download an app, register as a user, load your credit card info, and scroll through seven pages of questions to avoid saying, “1 large pepperoni pizza please” into a phone.

Breaking up and Reconciling

- Can Split strings based on breaks into constituent elements
- Can also Join or concatenate string components

Lists

Lists! The workhorse of basic Python Data Structures

- Why do we love lists so much?
  - Least restrictive, and generally most useful data type
  - Fully mutable!
    - Allows for in-place edits, additions, and subtractions without having to reference and reassign
      - Why does this matter to us if we can simply do reassignments?
  - Referentiable
    - Easy to traverse and access
  - (multi-typed!) We can store any type of data in the list (including other lists!)

Data modifications

- Can Add data elements to the overall list
- Can Remove individual items
- Can modify items
- Can also reorganize items within the list
- And like strings…
  - Can conduct a count and index position search

Data Referencing

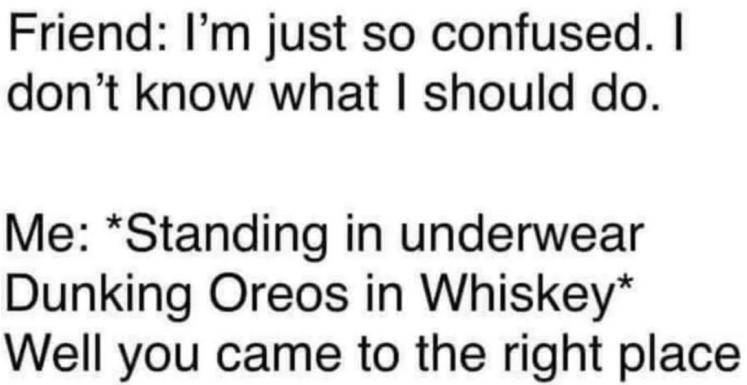
- Lists are fully iterable!
  - So we can use the item directly or…
  - We can use the index position
- Just like strings we can grab individual elements or entire ranges of options

Heinz to give new boat to man who survived on ketchup while lost at sea | US news

The Guardian · 11 h

help thats kinda funny. sending him back out to sea

List Comprehensions



- Sometimes we will want to uniformly perform a set of operations on a list.
  - We can do this in a for loop…
  - But there's a more compact representation: comprehensions
- Contain three elements
  - Variable assignment
  - Operation
  - Compacted for loop
- E.g., to create a new list of squared numbers:
  - `Squared = [i**2 for i in numbers]`

Dictionaries!

- Up until now keys were simply positional, but what if we want to take advantage of a pair-wise relationship to define a referential data structure
- Dictionaries allow this, where the Key is no longer positional, but rather a predefined element
  - Like a dictionary where the organizing keys are the words and the definitions are the associated values
- Keys and values can be multi-typed
- Both can be referenced and extracted
- Only values can be modified
- Only keys can be referenced

Program Structure

Putting it together

- Cobbling together a Python program often feels like constructing an assembly line
- We have to structure the overall process flow of the program including the various control flow elements
- Segment out modular components (e.g., functions and classes) and construct them independently before integrating them
- Often an iterative process, building and testing piecemeal