homework
- put all function calls inside main function

Python Primer 2: Functions and Control Flow

Sareh Taebi

Program Structure

Python would distinguish the block of code with the tab/indent

- Common to all control structures, the colon character is used to delimit the beginning of a block of code that acts as a body for a control structure.
- If the body can be stated as a single executable statement, it can technically place on the same line, to the right of the colon.
- However, a body is more typically typeset as an indented block starting on the line following the colon.
- Python relies on the indentation level to designate the extent of that block of code, or any nested blocks of code within.

Conditionals

```
if first_condition:            if this condition is true:
(tab) ---> first_body          execute this line, if not skip
elif second_condition:         if this is true:
---> second_body               execute this, if not skip
elif third_condition:          if this is true:
---> third_body                execute this
else:                          if everything above is false:
    fourth_body                execute this instead
```

```
if 90 <= grade <= 100:
    final = 'A'
elif 80 <= grade :
    final = 'B'
elif 70 <= grade :
    final = 'C'
else:
    final = 'F'
```

removed other comparison, not needed

want your code to be fast / efficient
remove redundancies

Indent is important

if/else statements are boolean
- if statement is true
  - will execute block of code under it
- if false
  - will skip over block of code

```
door_is_closed =  True

if door_is_closed:
    open_door =  True
    door_is_closed =  False
advance = True
```

if the door is closed:
1. open the door
2. remove the flag on door_is_closed
   - will not repeat unless it becomes true again

advance
moves forward when the door is open

```
door_is_closed =  True
door_is_locked = True
unlock_door = False
advance = False

if door_is_closed:
    if door_is_locked:
        unlock_door = True
    open_door = True
advance = True
```

added another level to unlock

if the door is already unlocked
- skips to open door
- continues to advance

* Refer to the code ControlStrc_Functions.py

Actually called lecture 4

Loops

avoid infinite loops
- create a way to stop

- **While loop:** allows general repetition based upon the repeated testing of a Boolean condition

(boolean condition)

```
while condition:
    body
```

condition = true
execute body of code

- **For loop:** provides convenient iteration of values from a defined series (such as characters of a string, elements of a list, or numbers within a given range).

```
for element in iterable:
    body
```

iterable: can be anything
element: identifier
- come up with name

- very flexible
- always have for in
  or for not in

- **Indexed For loop:**

use when you want to find
where in series it is
j: counter
range: counts over range
of indices
- range of length (100) of data
- creates sequence 0-99 (not including 100)

range(n):
0 to n-1

```
big_index = 0
for j in range(len(data)):
    if data[j] > data[big_index]:
        big_index = j
```

while loop example

always use counter
- to end loop

len: length, built in function
- number of elements in the sequence

```
j = 0
while j < len(data) and data[j] != 'X' :
        j += 1
```

while within length of data
- and while X is not in the sequence

returns the length of a sequence

- if you reach the length it will end
    - even if you have not reached X yet
-----looking for the first false behavior
----- when short circuiting an and

short-circuiting behavior of the *and* operator

if written the other way around:
IndexError when 'X' is not found

and- both have to be true
- if there is no x- will keep going
- even past length

Index Error: exceeded bounds of sequence

for loops automatically stop at the end of the sequence
- doesn't need space for each item
- creating 1 object, val points to object, putting elements of data 1 by 1 into memory to check them

for loop example

val = identifier
data = sequence
- goes one by one
- adding val to total each time

```
total = 0
for val in data:
    total += val
```

sum of values in a sequence
- also have built in called sum

```
# assumes the first value is the largest
biggest = data[0]
for val in data:
    if val > biggest:
        biggest = val
```

max of values in in a sequence of data

using identifier (val) to browse data one by one
- comparing data to our assumption

- if it comes across a value in data that is larger than our assumption (biggest) then replace that value with the new found largest value

can usually avoid using with
elaborate condition instead
- conditions also break loops

Break and Continue

- Python supports a **break** statement that immediately terminate a while or for loop when executed within its body.

Use sparingly

```
found = False
for item in data:
  if item == target:
    found = True
    break
```

looking for target in data set
- finds--> ends loop

- Python also supports a **continue** statement that causes the current iteration of a loop body to stop, but with subsequent passes of the loop proceeding as expected.

#function call
#fun_name([data], target)
count([1, 2, 3], 2)

<--- actual arguments
not formal parameters
- objects created
- can also use identifiers that
point to objects

Functions

- Functions are defined using the keyword **def**.

signature/heading/declaration --->

```
def count(data, target):
    n = 0
    for item in data:
        if item == target:
            n += 1
    return n
```

(formal_parameters, input):
- not actual arguments/objetcs
- just pointers/aliases to
  objects/actual arguments

- iterating through data set
- finding an item, counting how many

- not copying data into function like C
--- just pointing to the data we want
to use in the function

return n to where the function was called

- This establishes a new identifier as the name of the function (count, in this example), and it establishes the number of parameters that it expects, which defines the function's **signature**.
- The **return** statement returns the value for this function and terminates its processing.

Function Call: Information Passing

- Parameter passing in Python follows the semantics of the standard assignment statement.
- For example

```
prizes = count(grades, 'A')
```

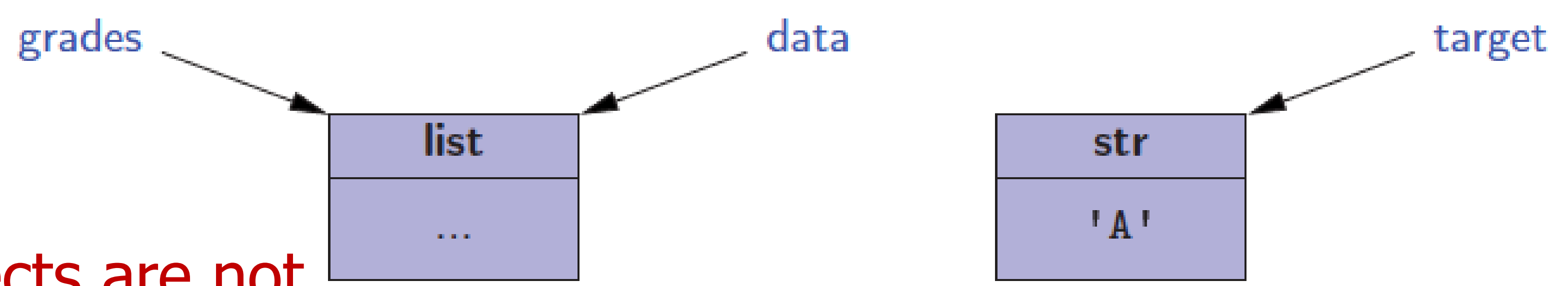
is the same as

```
data = grades
target = 'A'
```

Aliases are created!

- grades & data point to list

and results in



Objects are not copied!

Methods:
- work with only certain classes/objects of classes
- each class has functions (methods) that only work that class

example:
- objectName.sort()

Functions:
- stand-alone piece of code that does 1 thing

sorted(): function that can work with anything

Files

- Files are opened with a built-in function, **open**, that returns an object for the underlying file.
- For example, the command, fp = open('sample.txt'), attempts to open a file named sample.txt.
- Methods for files:

accessor/mutator

open excel file
- read()/(k)
- seek(k)
- tell()
- write(string)
- writelines(seq)

Calling Syntax	Description
fp.read()	Return the (remaining) contents of a readable file as a string.
fp.read(k)	Return the next *k* bytes of a readable file as a string.
fp.readline()	Return (remainder of) the current line of a readable file as a string.
fp.readlines()	Return all (remaining) lines of a readable file as a list of strings.
for line **in** fp:	Iterate all (remaining) lines of a readable file.
fp.seek(k)	Change the current position to be at the k^{th} byte of the file.
fp.tell()	Return the current position, measured as byte-offset from the start.
fp.write(string)	Write given string at current position of the writable file.
fp.writelines(seq)	Write each of the strings of the given sequence at the current position of the writable file. This command does *not* insert any newlines, beyond those that are embedded in the strings.
print(..., file=fp)	Redirect output of print function to the file.

Exception Handling

- Exceptions are unexpected events that occur during the execution of a program.
- An exception might result from a logical error or an unanticipated situation.
- In Python, exceptions (also known as errors) are objects that are raised (or thrown) by code that encounters an unexpected circumstance.
  - The Python interpreter can also raise an exception.
- A raised error may be caught by a surrounding context that “handles” the exception in an appropriate fashion. If uncaught, an exception causes the interpreter to stop executing the program and to report an appropriate message to the console.

Common Exceptions

- Python includes a rich hierarchy of exception classes that designate various categories of errors

Class	Description
Exception	A base class for most error types
AttributeError	Raised by syntax obj.foo, if obj has no member named foo
EOFError	Raised if "end of file" reached for console or file input
IOError	Raised upon failure of I/O operation (e.g., opening file)
IndexError	Raised if index to sequence is out of bounds
KeyError	Raised if nonexistent key requested for set or dictionary
KeyboardInterrupt	Raised if user types ctrl-C while program is executing
NameError	Raised if nonexistent identifier used
loops?-> StopIteration	Raised by next(iterator) if no element; see Section 1.8
TypeError	Raised when wrong type of parameter is sent to a function
ValueError	Raised when parameter has invalid value (e.g., sqrt(−5))
ZeroDivisionError	Raised when any division operator used with 0 as divisor

Raising an Exception

- An exception is thrown by executing the raise statement, with an appropriate instance of an exception class as an argument that designates the problem.
- For example, if a function for computing a square root is sent a negative value as a parameter, it can raise an exception with the command:

```
raise ValueError('x cannot be negative')
```

- can be used in your code to let user know they have made an error
- should be able to write code that works around the exceptions/errors

Catching an Exception

- In Python, exceptions can be tested and caught using a try-except control structure.

exercise:
- use try/except with all of the errors
  in the chart

```
try:
    ratio = x / y
except ZeroDivisionError:
    ... do something else ...
```

- does not terminate ---> try something else

- In this structure, the “try” block is the primary code to be executed.
- Although it is a single command in this example, it can more generally be a larger block of indented code.
- Following the try-block are one or more “except” cases, each with an identified error type and an indented block of code that should be executed if the designated error is raised within the try-block.

Iterators

- Basic container types, such as list, tuple, and set, qualify as iterable types, which allows them to be used as an iterable object in a for loop.

  for *element* **in** *iterable*:

- An iterator is an object that manages an iteration through a series of values. If variable, **i**, identifies an iterator object, then each call to the built-in function, **next(i)**, produces a subsequent element from the underlying series, with a **StopIteration** exception raised to indicate that there are no further elements.
- An iterable is an object, **obj**, that produces an iterator via the syntax **iter(obj)**.

Generators

- The most convenient technique for creating iterators in Python is using generators.
- A generator is implemented with a syntax that is very similar to a function, but instead of returning values, a yield statement is executed to indicate each element of the series.
- For example, a generator for the factors of n:

```
def factors(n):                  # generator that computes factors
  for k in range(1,n+1):
    if n % k == 0:               # divides evenly, thus k is a factor
      yield k                    # yield this factor as next result
```

if you have 1 simple if/else: you can write it all on one line

Conditional Expressions

- Python supports a conditional expression syntax that can replace a simple control structure.
- The general syntax is an expression of the form:

$$\textit{expr1} \textbf{ if } \textit{condition} \textbf{ else } \textit{expr2}$$

- This compound expression evaluates to expr1 if the condition is true, and otherwise evaluates to expr2.
- For example:

```
param = n if n >= 0 else -n      # pick the appropriate value
result = foo(param)              # call the function
```

- Or even

```
result = foo(n if n >= 0 else -n)
```

Comprehension Syntax

- A very common programming task is to produce one series of values based upon the processing of another series.
- Often, this task can be accomplished quite simply in Python using what is known as a comprehension syntax.

```
[ expression for value in iterable if condition ]
```

- This is the same as

```
result = [ ]
for value in iterable:
  if condition:
    result.append(expression)
```

automatically fills in values
- start with empty list
- for loop
--- if condition
----- append() #add to sequence

List Comprehension

- automatic comprehensions of a sequence

[k*k for k in range(1, n+1)] list comprehension [1, 4, 9, 16]
{ k*k for k in range(1, n+1) } set comprehension
(k*k for k in range(1, n+1)) generator comprehension
{ k : k* k for k in range(1, n+1) } dictionary comprehension

factors = [k for k in range(1,n+1) if n % k == 0]

- has a condition: if the remainder = 0 (whole numbers) use for assignments asking for mod & division together in one sequence

Packing

- If a series of comma-separated expressions are given in a larger context, they will be treated as a single tuple, even if no enclosing parentheses are provided.
- For example, consider the assignment

```
data = 2, 4, 6, 8
```

- This results in identifier, data, being assigned to the tuple (2, 4, 6, 8). This behavior is called **automatic packing** of a tuple.

Unpacking

- As a dual to the packing behavior, Python can automatically unpack a sequence, allowing one to assign a series of individual identifiers to the elements of sequence.
- As an example, we can write

```
a, b, c, d = range(7, 11)
```

- This has the effect of assigning a=7, b=8, c=9, and d=10.

Modules

- Beyond the built-in definitions, the standard Python distribution includes perhaps tens of thousands of other values, functions, and classes that are organized in additional libraries, known as **modules**, that can be imported from within a program.

```
import math
```

Existing Modules

- Some useful existing modules include the following:

Existing Modules	
Module Name	**Description**
array	Provides compact array storage for primitive types.
collections	Defines additional data structures and abstract base classes involving collections of objects.
copy	Defines general functions for making copies of objects.
heapq	Provides heap-based priority queue functions (see Section 9.3.7).
math	Defines common mathematical constants and functions.
os	Provides support for interactions with the operating system.
random	Provides random number generation.
re	Provides support for processing regular expressions.
sys	Provides additional level of interaction with the Python interpreter.
time	Provides support for measuring time, or delaying a program.