CS571 Advanced Programming Techniques

Python

Today's agenda

- Select answers to Midterm
- make
- Python
- Solving Assignment 4 in class (bash)

Select Answers to Midterm

1. Consider the following two lines from some script. In which line(s) does wc receive input from standard input?
   - ○ wc -l a.txt
   - ✓ cat a.txt | wc –l
2. Which of the following files will be displayed by this command

   cat *ch*
   - ○ .ch
   - ○ catch
   - ○ patch
   - ○ none of the above
   - ✓ all of the above
3. In a regular expression **.?** matches exactly one character
   - ○ true
   - ✓ false

Select Answers to Midterm (Cont'd)

4. Which **awk** command outputs all lines where the second field is larger than the line number:
   - ○ $NF < $2 { print }
   - ✓ NR < $2 { print }
   - ○ $2 > $LN { print $0 }
   - ○ $2 > LN { print

   NR is the number of records (lines) read so far

5. Suppose that the current directory contains the following files:

   `a2.txt ab1.doc ab123.pdf b1.tex b12.exe`

   How many files will be listed by the following command?

   ls [ab]?[123]*

   - ○ 5
   - ○ 4
   - ✓ 3
   - ○ 2
   - ○ 1
   - ○ 0

   a2.txt ab1.doc ab123.pdf b1.tex b12.exe

Python 3

Python

- A general-purpose language that can be used as:
  - a scripting language (like awk)
  - a procedural language (like C)
  - an object-oriented language (like Java)
- Applications for Python:
  - scientific and numeric computing
  - machine learning (scikit-learn, tensorflow, keras, etc. library)
  - Web and internet development
  - Tk GUI library is included with most distributions
- Python is often used as a support language for developers, for build control and management, for testing, in many other places

Python 2 vs Python 3

- As of January 2020, Python 2.x is on EOL (End-Of-Line) status
  - will no longer be supported
  - EOL planned for the last 10 years
  - almost everything has been ported
- Recommended version is Python 3
  - Tux has two versions
    - Python 2.7.17 (invoke using `python`)
    - Python 3.6.9 (invoke using `python3`)
  - Latest version is 3.8.1
- To check version running use

  `python -V` or

  `python --version`
- There are some differences between Python 2 and Python 3
- Download Python3 from *https://python.org/*

Interpreter

- These notes refer to Python 3.6
- Python has a very convenient interactive interpreter can be used as a calculator
- documentation is handy

```
$ python
Python 3.6.1 ...
>>> 4 + 3
7
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

- Use ^D to exit

Useful Tidbits

- Newlines separate statements in Python
  - Use ; to separate statements on the same line
- Escape the newline with \ to continue a statement

```
>>> x = 40 + \
…2
>>> x
42
```

- # introduces a line comment

```
>>> i = 5 # Newtons
>>> j = 3*i # Very clever calculation
```

- `None` is the sentinel reference (the `NULL` pointer)

Running HelloWorld.py

```
% ls

hello_world.py

% cat hello_world.py

print("Hello World")

% python hello_world.py

Hello World!
[Finished in 0.1s]
```

Variables and Simple Data Types

hello_world.py

```
message = "Hello world!"
print(message)
```

Variables

- Python variables are not declared explicitly
  - implicit declaration by usage
  - they are dynamically typed
- Everything is a reference in Python
- Everything is an object in Python
  - some types are immutable

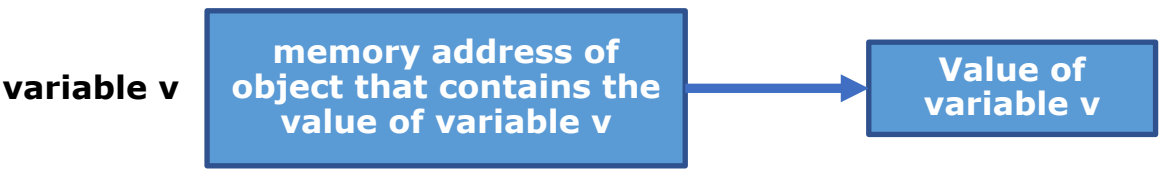
```
>>> x = 12
>>> type( x )
<type 'int'>
>>> x = 1.732050807
>>> type( x )
<type 'float'>
>>> x = "Zaphod Beeblebrox"
>>> type( x )
<type 'str'>
```

Parenthesis: Value vs Reference Types

From ECMA International Standard 335 defining the Common Language Infrastructure (CLI):

- Value type: A type such that an instance of it directly contains all its data. The values described by a value type are self-contained."

- Reference type: A type such that an instance of it contains a reference to its data. A value described by a reference type denotes the location of another value."



Basic Types in Python

- Strings
- Numbers
- Lists - arrays defined using []
- Tuples - immutable lists defined using ()
- Dictionaries– associative arrays defined using {}
- Sets

- Which can be combined too, e.g., in lists of dictionaries and dictionaries containing lists, etc.

Strings - `str`

- Anything in quotes is a string
- You can use single or double quotes

```
"This is a string."
'This is also a string.’


'I told my friend, "Python is my favorite language!"'
"The language 'Python' is named after Monty Python, not the snake."
"One of Python's strengths is its diverse and supportive
community."
```

Strings - str

- Concatenation with +
- Repetition with *

```
>>> f = 'Cookie'
>>> l = "Monster"
>>> n = f + l
>>> n
'CookieMonster'
>>> n = f + ' ' + l
>>> print n
Cookie Monster
>>> print('Spam ' * 3)
Spam Spam Spam
```

Strings methods

- String methods: `title()`, `upper()`, `lower()`

```
name = "ada lovelace"
print(name.title())

print(name.upper())

print(name.lower())
```

output

```
Ada Lovelace
ADA LOVELACE
Ada lovelace
```

- Stripping whitespace: `rstrip()`, `lstrip()`, `strip()`

```
>>> favorite_language = ' python '
>>> favorite_language.rstrip()
   ' python'
>>> favorite_language.lstrip()
   'python '
>>> favorite_language.strip()
   'python'
```

String concatenation & types

- Python is strongly typed
  - for example, you can't add a number to a string
- Every object has a `__str__` method
- called by the `str` operator

```
>>> print('Jupiter' + str(13))
Jupiter13
>>> print(str( 10 ) + 'Q')
10Q
>>> print("My list: " + str([1,2,3]))
My list: [1, 2, 3]
```

Strings are iterable

- Indexed, starting at 0
- use index operator, []
- `len` operator yields the length of an iterable
- strings are immutable (cannot modify once created)

```
>>> n = "CookieMonster"
>>> len(n)
13
>>> print(n[0])
C
>>> print(n[12])
r
>>> n[9]='b'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Formatting strings

- To insert a variable value in a string, place the letter `f` before the opening quotation
- `f` stands for formatted
- These strings are called `f-strings`
- `Can include`can include `\t(tab), \n(newline),` etc.

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
print(full_name)
print(f"Hello, {full_name.title()}!")
print(f"Hello {first_name.title()} {last_name.title()}")
print(f”Hello {full_name.title()”)
```

```
>>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
Languages:
    Python
    C
    JavaScript
```

`split` and `join` methods

- `split` takes a delimiter, returns a list of strings
- `split` takes an optional maximum number of elements to return in the list

```
>>> line='line,from,a,CSV'
>>> line.split(',')
['line', 'from', 'a', 'CSV']
>>> line.split(',',2)
['line', 'from', 'a,CSV']
```

- `join` join takes a list of strings, returns a string

```
>>> list=['Q1','Q2','Q3','Q4']
>>> ':'.join(list)
'Q1:Q2:Q3:Q4'
```

Use indices to get substrings

- Index operators can take a range
  - called a *slice*
- End position is a "one past the end" notion
  - that character not included
  - leave empty to indicate rest of the string

```
>>> s = 'Isaac Asimov'
>>> s[4:9]
'c Asi'
>>> len( s )
12
>>> s[11]
'v'
>>> s[6:11]
'Asimo'
>>> s[6:]
'Asimov'
```

More on slices

- Can leave start position blank to start at the beginning:

```
>>> s[0:7]
'Isaac A'
>>> s[:7]
'Isaac A'
```

- Use a negative position to count relative to the end

```
>> s[-1] # the last letter
v
>>> s[:-1] # All but the last letter
'Isaac Asimo'
>>> s[:-2] # All but the last two
'Isaac Asim'
```

dir

- `dir` lists all members of a class

```
>>> # Name the type:
>>> dir( str )
...
>>> # or, use an instance, or variable holding that type:
>>> dir( 'blah' )
...
```

- Note, members surrounded by underscores are generally helper methods, used for defining operators

Try it yourself

Write a python program for each of these:

- **Personal Message:** Use a variable to represent a person's name, and print a message to that person. Your message should be simple, such as,

  "Hello Marvin, would you like to learn some Python today?"

- **Famous Quote:** Find a quote from a famous person you admire. Print the quote and the name of its author. Your output should look something like the following, including the quotation marks:

  Albert Einstein once said, "A person who never made a mistake never tried anything new."

Numbers

Integers - `int`

- You can add (+), subtract (-), multiply (*), divide (/) integers

```
>>> 2 + 3
5
>>> 3 - 2
1
>>> 2 * 3
6
>>> 3 / 2
1.5
```

- With the typical operator precedence

```
>>> 2 + 3*4
14
>>> (2 + 3) * 4
20
```

Integers - `int`

- `int` is the only integer type, and it handles arbitrarily large integers

```
>>> type( 10 )
<class 'int'>
>>> type( 10**20 )
<class 'int'>
```

Floats - `float`

- Python calls any number with a decimal point a float
- For the most part we use floats without worrying how they behave

```
>>> 0.1 + 0.1
0.2
>>> 0.2 + 0.2
0.4
>>> 2 * 0.1
0.2
>>> 2 * 0.2
0.4
```

- But sometimes we get an arbitrary number of decimal places

```
>>> 0.2 + 0.1
0.30000000000000004
>>> 3 * 0.1
0.30000000000000004
```

Integers & Floats

- If you mix an integer and a float, you get a float
- Python has 2 division operators

  / is float division

  // is integer division

```
>>> 12.0 / 5
2.4
>>> 12 / 5
2.4
>>> 12 // 5
2
```

- Python has an exponentiation operator, **

```
>>> 3 ** 4
81
>>> 3.0 ** 4
81.0
>>> 2**0.5
1.4142135623730951
```

Complex Numbers

- Python actually has built-in complex types.
- Components can be `int` or `float`

```
>>> type( 3+4j )
<type 'complex'>
>>> type( 3.0+4j )
<type 'complex'>
>>> ( 3+4j ) * ( 27-12j )
(129+72j)
>>> ( 12+17j ) / 5
(2.4+3.4j)
>>> (3+4j)**2
(-7+24j)
```

Underscores in Numbers

- When you're writing long numbers, you can group digits using underscores to make large numbers more readable

```
>>> universe_age = 14_000_000_000
>>> print(universe_age)
14000000000
```

Multiple Variable Assignment

- You can assign values to more than one variable using just a single line.
- Used when initializing a set of numbers
- Can help shorten programs and make them easier to read

```
>>> x, y, z = 0, 0, 0
```

Constants

- Python does not have built-in constant types
- Python programmers use all capital letters to indicate a variable should be treated as a constant and never be changed

```
MAX_CONNECTIONS = 5000
```

Lists (arrays)

Lists

- A *list* is a collection of items in a particular order
- Indicated with []
- You can put anything you want into a list, and the items in your list don't have to be related in any particular way.

```
cars=['bmw','audi','ford','ferrari']
print(cars)
print(cars[1])
print(cars[0].title())
```

- Index positions start at 0, not 1
- The last element in a list can be accessed with index -1

```
print(cars[1])
print(cars[-1])
message = f"My first car was a {cars[0].title()}."
print(message)
```

Lists

```
>>> l = list() # explicitly call constructor
>>> l = [] # Use language syntax
>>> m = [ 5.7, 'Dead Collector', 13 ]
>>> m2 = [ 'A', m, 42 ]
```

- Use len

```
>>> len( m )
3
>>> m2
['A', [5.7, 'Dead Collector', 13], 42]
>>> len( m2 )
3
```

List Operations – `append`, `insert`

- Modify elements

```
cars[0]='audi'
```

- Use `append` to add an item to the end of the list:

```
cars = []
cars.append('bmw')
cars.append('audi')
cars.append('ford')
print(cars)
```

- Use `insert(index,item)` to insert `item` at `index` position

```
cars.insert(0,'ferrari')
```

List Operations – `del`, `pop`, `remove`

- Use the `del` statement to remove indexed elements

```
del cars[1]
```

- Use the `pop(index)` method to remove the element at the index position (last by default)

```
popped = cars.pop()
popped = cars.pop(2)
```

- Use `remove()` to remove an item by value

```
cars.remove('ferrari')
```

A parenthesis: A statement, a method or a function?

- A statement is a command. It does something. In most languages, statements do not return values.

```
del cars[1]
```

- A function is a subroutine that can be called elsewhere in the program. Functions often (but not necessarily) return values. Example:

```
print(cars)

print(sorted(cars))
```

- A method is a function that "belongs to" an object.

```
cars.remove('ferrari')
```

List Operations – `sort, sorted, reverse`

- Sort the list permanently with the `sort` method

```
cars.sort()
cars.sort(reverse=True)
```

- Sort the list temporarily with the `sorted` function

```
print(sorted(cars))
```

- Reverse the order permanently using the `reverse` method

```
cars.reverse()
```

- Find the length of a list using the `len` function

```
len(cars)
```

Working with lists – `for` loop

- Use a `for` loop to iterate over each element in a list

```
cars=['bmw','audi','ford','ferrari’]

for car in cars:
    print(f”car.title()”)
print("that's all!")
```

Python relies on proper indentation

- Indentation is used to block statements together (in a loop, if-then-else statement, etc.)

The pros

- Makes code easy to read
- Forces the programmer to write well formatted code showing the program's organization

The cons

- Since indentation is important, forgetting to indent or indenting unnecessarily may be an issue
- Be careful not to mix spaces and tabs

Making numeric lists - `range`

- The `range` function can be called a couple ways:
  - `range( end )`
  - `range( start, end[, step] )`
- Returns a list
- `end` is one-past-the-end

```
for value in range(1, 5):
    print(value)
```

- Returns

```
1
2
3
4
```

- Note that 1 is included but 5 is not

Making numeric lists - `range`

- Using the `range` function, Python stops one item before the second index you specify

```
numbers = list(range(1, 6))
print(numbers)
```

- Returns

```
[1, 2, 3, 4, 5]
```

Making numeric lists - `range`

- Using the `range` function to populate a list.
- You don't have to add 1 each time, you can add a specified amount (third argument)
- For example, here's how to list the even numbers between 1 and 10 (start with 2, add 2 each time, until you get to under 11)

```
even_numbers = list(range(2, 11, 2))
print(even_numbers)
```

- Returns

```
[2, 4, 6, 8, 10]
```

List statistics – `min, max, etc`

- Find the minimum, maximum, and sum of a list of numbers:

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

List Comprehensions

- An efficient, easy way to map a function over a list
- Instead of this

```
squares = []
for value in range(1,11):
    squares.append(value**2)
print(squares)
```

- You can use a list comprehension to do this

```
squares = [value**2 for value in range(1, 11)]
print(squares)
```

- Outputs

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

List Comprehensions

- We can do joins

```
>>> [ (i, j) for i in range(1,5) for j in range(1,5) ]
[(1, 1), (1, 2), (1, 3), (1, 4), (2, 1), (2, 2), (2, 3), (2, 4),
(3, 1), (3, 2), (3, 3), (3, 4), (4, 1), (4, 2), (4, 3), (4, 4)]
```

Slices

- Just as with strings, we can take slices of a list
  - returns new list
  - does not modify original list

```
>>> l = [ 1, 2, 3, 4 ]
>>> l[1:3]
[2, 3]
>>> l[:-1]
[1, 2, 3]
```

- Take a slice of the entire array to make a copy:

```
>>> m = [ 'a', l[:], 'b' ]
>>> m
['a', [1, 2, 3, 4], 'b']
>>> l[2] = 42
>>> l
[1, 2, 42, 4]
>>> m
['a', [1, 2, 3, 4], 'b']
```

Slices

- To make a slice, you specify the index of the first and last elements you want to work with.
- As with the range() function, Python stops one item before the second index you specify.
- The index of the first and last element maybe missing
-

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[0:3])
print(players[1:4])
print(players[:4])
print(players[2:])
print(players[-3:])
```

```
['charles', 'martina', 'michael']
['martina', 'michael', 'florence']
['charles', 'martina', 'michael', 'florence']
['michael', 'florence', 'eli']
['michael', 'florence', 'eli']
```

Slices – more examples

```
my_foods = ['pizza', 'falafel', 'carrot cake’]
friend_foods = my_foods[:]

my_foods.append('cannoli’)
friend_foods.append('ice cream')


print("My favorite foods are:")
print(my_foods)
print("\nMy friend's favorite foods are:")
print(friend_foods)
```

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli']


My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'ice cream']
```

Tuples

Tuples – immutable lists

- Lists of items that cannot change
- Uses parenthesis, instead of square brackets
- Handy as a key in a dictionary (soon)

```
dimensions = (200, 50)
```

- Tuples are technically defined by the presence of a comma; the parentheses make them look neater and more readable. If you want to define a tuple with one element, you need to include a trailing comma:

```
my_t = (3,)
```

Tuples

- Can be made from any iterable

```
>>> l = [ 1, 2, 3, 4, 5 ]
>>> t = tuple( l )
>>> t
(1, 2, 3, 4, 5)
>>> d=tuple('Dimitra')
>>> d
('D', 'i', 'm', 'i', 't', 'r', 'a')
```

- Can be indexed

```
>>> t[1:4]
(2, 3, 4)
```

- Cannot be modified (immutable)

```
>>> t[3]='Gollum'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Tests

Relational Operators

- Python has two boolean literals: `False / True`
- We have the usual relational operators:

`< <= == != >= >`

```
>>> 3 < 22
True
>>> 'bulb' < 'flower'
True
>>> 'bulb' < 'Flower'
False
>>> 42 < '17'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
```

Logical Membership

- We have friendly logical operators:

  `not and or in`

  - Listed in order of decreasing precedence
  - Use parentheses
- Python has a membership operator, `in`:

```
>>> 3 in [1, 2, 3]
True
>>> 2 in (1, 2, 3)
True
>>> 'i' in 'team'
False
>>> 'am' in 'team'
True
>>> 'i' not in 'team'
True
```

Identity Operator

- Remember, everything in Python is a reference
- `is` operator tests references

```
>>> l=['a','b','c']
>>> m = l
>>> m == l
True
>>> m is l
True
>>> c = l[:]
>>> c == l
True
>>> c is l
False
>>> c is not l
True
```

Implementing Operators

Behavior for operators supplied by method in left-most operand

+	__add__
-	__sub__
*	__mult__
/	__div__
//	__floordiv__
%	__mod__
str()	__str__
> == etc.	__cmp__
~	__invert__
&	__and__
&	__and__

Control

Branches – `if`

```
if cond :
    body
```

- If consequent is a single statement, you can do this:

```
if cond : body
```

- Note, the lack of parentheses / brackets
- Body of consequent is uniformly indented
- Indentation necessary

if-else

More generally:

```
if cond :
    body
elif cond :
    body
    ...
else :
    body
```

Loops

- Python has `for` and `while` loops
- No `until` nor `do` loops
- We have the usual `break` and `continue` statements

While Loops

- Again, statements in body denoted by new (consistent) indent level
- Body of loop is indented (consistently)

Dictionaries

Dictionaries

- A *dictionary* in Python is a collection of *key-value pairs*.
- Key can be any immutable type
- Value can be anything

```
>>> alien_0 = {'color': 'green', 'points': 5}
>>> print(alien_0['color'])
green
>>> alien_0['color'] = 'yellow'
>>> print(f"The alien is now {alien_0['color']}.")
The alien is now yellow.
```

Testing for membership

- Use the keyword `in`

```
>>> d={17:'seventeen','Gandalf':'White',('one','two'):'buckled shoe',3.1416:'Pi'}
>>> 3.1416 in d
True
>>> 'Gandalf' in d
True
>>> "Dimitra" in d
False
```

Deleting from a dictionary – `del`

- Use the `del` statement to remove a key/value pair
- It is permanent

```
>>> alien_0 = {'color': 'green', 'points': 5}
>>> print(alien_0)
{'color': 'green', 'points': 5}
>>> del alien_0['points']
>>> print(alien_0)
{'color': 'green'}
```

Accessing values – `get`

- Using keys in square brackets to retrieve a value returns an error, if the key doesn't exist
- Use the `get()` method to return a default value if the requested key doesn't exist
- Use the `get()` method to set the default value

```
>>> alien_0 = {'color': 'green', 'speed': 'slow'}
>>> print(alien_0['points'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'points'
>>> point_value = alien_0.get('points', 'No point value
assigned.')
>>> print(point_value)
No point value assigned.
```

Looping through the items

- Loop using a `for` loop and the `items()` method

```
alien_0 = {'color': 'green', 'speed': 'slow'}
print(alien_0)

for key, value in alien_0.items():
    print(f"\nKey: {key}")
    print(f"Value: {value}")
```

```
{'color': 'green', 'speed': 'slow'}
Key: color
Value: green
```

Looping through the keys

- An iterator over a dictionary is an iterator over its keys
- Or, use the `keys` method
- Use the `sorted()` function to sort the keys first

```
favorite_languages = {
       'jen': 'python',
       'sarah': 'c',
       'phil': 'python',
       }

for name in favorite_languages.keys():
       print(name.title())

for name in sorted(favorite_languages.keys()):
       print(name.title())
```

```
Jen
Sarah
Phil
Jen
Phil
Edward
```

Looping through a dictionary

- An iterator over a dictionary is an iterator over its keys
- Or, use the `keys` method

Nesting

Nesting is a powerful feature. You can nest

- a dictionary inside a list
- a list inside a dictionary
- a dictionary in a dictionary

Files

Reading from a file

- Here's a program that opens this file, reads it, and prints the contents of the file to the screen

```
with open('pi_digits.txt') as file_object:
    contents = file_object.read()
print(contents)
```

- Remove the extra blank lines

```
with open('pi_digits.txt') as file_object:
    contents = file_object.read()
    print(contents.rstrip())
```

- Read line by line

```
filename = 'pi_digits.txt'

with open(filename) as file_object:
    for line in file_object:
          print(line)
```

Reading from a file

- Making a list of lines from a file

```
filename = 'pi_digits.txt'

  with open(filename) as file_object:
    lines = file_object.readlines()

  for line in lines:
    print(line.rstrip())
```

Writing to a file

- call open() with “w” (write) as a second argument

```
filename = 'programming.txt’

with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
```

- call open() with “a” (append) as a second argument

```
filename = 'programming.txt'

with open(filename, 'a') as file_object:
    file_object.write("I also love finding meaning in datasets.\n")
```

What's Next

- Assignment 6 due February 25 at 11:59pm