11/5/19

CIS 210

Programming/Computer Science concepts

Computational Problem Solving: designing, implementing, checking, revising algorithms/approaches.

Good programming style: function docstrings (type contract; description including parameters, return value, and side effects; examples of function use), well-named variables, use of whitespace between operators and sections of code, judicious use of inline comments (why not what).

Python is a programming language and Python is an interpreter (program)

Python Shell is a REPL (read-evaluate-print loop)

Python primitive elements: Objects - value(s), type, memory location; memory management; garbage collection; immutable data type; mutable data types

Combining primitive elements: Expressions - expressions evaluate to a value; short circuit evaluation of boolean expressions, pre- and post- operators

Naming values: Variables/assignment - assignment statements are not expressions and do not return a value; namespaces - builtins and global (__main__); scope; dynamic typing, strong typing

Functions are an executable data type, what happens when a function - method - is called: Activation record/stack frame added to call stack for local namespace; return address

Call-by-assignment parameter passing

Functions always return a value (sometimes None)

Functions sometimes have side effects

Functions as arguments

Iterative algorithms; accumulator pattern; Monte Carlo algorithms; data analysis

CIS 210 Learning Outcomes

• understand, develop, implement algorithms for computational problem solving;
• use structured design and testing methods to develop and implement programs;
• read, write, revise, document, test, and debug code;
• demonstrate robust mental models of data representation and code execution;
• demonstrate good understanding of a high-level programming language;
• introduce and/or implement a sampling of classic computer science problem domains and algorithms.

CIS 210

Programming/Computer Science concepts, 2.

Systematic approaches to testing and debugging; automated testing

What happens when an assignment statement is executed: memory allocation; reference semantics; Faculty Home

CIS 210

Python toolkit so far

Numeric data types (int, float) and operations (e.g., +, *, round, abs)

String data type and operations (e.g., +, , count, find)

Formatted strings (and string format method)

Boolean data type and relational/Boolean operations  (e.g., <, and)

Python collections data types and operations – tuples, lists, dictionaries

Data type coercion functions, e.g., str, int

NoneType (None)

print, input

expressions

Python Standard Library – math, turtle, random, doctest, datetime
modules; import

assignment statement

Python repetition – for, while

Python conditionals – if, else

user-defined functions; function design; docstrings

IDLE interactive development environment; help function; dir, type, id

CIS 210

A Structured Approach to Computational Problem-Solving

– review the project specification thoroughly
– write examples of expected results for specified inputs – re-review spec, if needed
– develop, review, and/or revise a problem-solving approach, using natural language, algorithms, pseudocode (not Python code)
– check algorithm using your examples – revise algorithm, re-review spec, if needed

Starting with the lowest level function –

– write the function header
– write the function docstring – type contract
– write the function docstring – brief description
– write the function docstring – examples of use (use ones developed earlier)
– write the return statement
– using tools from the Python toolkit, start writing the body of the function
– test often, revise as needed
– test using examples in the docstring, and then project spec, and then others

CIS 210

REVIEW
```python
>>> __name__
a) '__builtins__'
b) '__main__'
c) str
d) TypeError
e) dir()
```

```python
>>> x = 99.9
>>> id(x)
```

```python
7
```

```python
>>> x = [1, 2, 3, 4]
>>> x.append('hi')
>>> x
```

```python
9
```

```python
>>> x = [1, 2, 3, 4]
>>> x.append('hi')
>>> x
```

```python
10
```

```python
CIS 210

Data

An assortment of items, often numerical, that have been observed, measured, or collected by some means, that represent the starting point for analysis that can be done in an attempt to understand the data and understand underlying characteristics that may be present. (text)
```

```python
11
```

```python
CIS 210

Python collections / mutable data types

- Python collections – strings, tuples, lists, dictionaries
```

```python
12
```
Python collections

Sequential
- Strings, Tuples, Lists

Unordered
- Dictionaries, Sets, Frozensets

Immutable
- Strings, Tuples, Frozensets

Mutable
- Lists, Dictionaries, Sets

13

Python collections – Sequential

<table>
<thead>
<tr>
<th>Strings</th>
<th>Tuples</th>
<th>Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = ‘abc’</td>
<td>y = (‘a’, ‘b’, ‘c’)</td>
<td>z = [‘a’, ‘b’, ‘c’]</td>
</tr>
<tr>
<td>x[0]</td>
<td>y[1]</td>
<td>z[2]</td>
</tr>
<tr>
<td>‘a’ in x</td>
<td>‘b’ in y</td>
<td>‘0’ in z</td>
</tr>
<tr>
<td>x = ‘xyz’</td>
<td>y = (‘a’, 1, True)</td>
<td>z = [‘a’, 1, (2,3)]</td>
</tr>
</tbody>
</table>

14

Important for understanding mutable data types:
• what happens during variable assignment
  - variables are names and references (pointers) to memory locations where a value (object) is stored
  - two variable (names) may reference the same object (value) – aliasing
• what happens when a function is executed
  - activation record on function call stack; local namespace
  - parameter passing by assignment – more aliasing
  - function execution may result in side effects - persist after the function is done executing (e.g., print, update mutable object)
  - at return keyword (or when end of the code is reached):
    • activation record is deleted
    • function returns a value (possibly None)
    • Python resumes processing where the function was called

15

Recall: Updating a string:

```python
>>> astr = ‘abc’
>>> astr.upper()
‘ABC’
>>> astr
‘abc’
>>> astr = astr.upper()
>>> astr
‘ABC’
```

16

17

18
Recall:

```python
>>> x = 'xyz'
>>> x[0] = 'z'  # X
```

?? how do we get to x = ‘zyz’

```python
>>> x = 'z' + x[1:]
>>> x = 'zyz'
>>> x
'zyz'
```

Lists are a mutable data type
(and strings and tuples are not)

```python
>>> y = ['a', True, 100]
>>> y[0] = 'b'
>>> y
['b', True, 100]
```

Lists are a mutable data type
(and strings and tuples are not)
can change the value of a complex object (including size) during program execution

flexible; powerful; convenient

also
potentially expensive (memory management)
Lists are a mutable data type

can change the value of a complex object (including size) during program execution

flexible; powerful; convenient; clear
also
expensive (memory management)

→ Python updates the object IN PLACE

Lists are mutable.
They can be updated in place.

Can change the size of the list

```python
>>> y.append(100)
>>> y
[99, 2, 3, 100]
>>> y.remove(2)
>>> y
[99, 3, 100]
>>> y = [99, 3, 100]
>>> y
[99, 3, 100]
>>> y.append(100)
>>> y
[99, 3, 100, 100]
>>> y = y.append(100)
>>> y
[99, 3, 100, 100, 100]
```

Can change the size of the list

```python
>>> y.append(100)
>>> y
[99, 2, 3, 100]
>>> y.remove(2)
>>> y
[99, 3, 100]
>>> y = [99, 3, 100]
>>> y
[99, 3, 100]
>>> y.append(100)
>>> y
[99, 3, 100, 100]
>>> y = y.append(100)
>>> y
[99, 3, 100, 100, 100]
```
Many list methods update a list as a **side effect** – and return None

```
>>> mystr = 'bye'
>>> myl = [1, False, 'hi']
>>> mystr[0] = 'r'
>>> myl[0] = 99
>>> mystr = mystr.upper()
>>> myl = myl.reverse()
>>> mystr = 'hello'
>>> myl = [10, 12, 2]
>>> yourstr = mystr
>>> yourl = myl
>>> mystr = mystr.capitalize()
>>> myl = myl.reverse()
```

Recall:

And for list:

```
>>> b = 20
>>> y = [1, 2, 3]
>>> a = b
>>> x = y
>>> b = 30
>>> y = [4, 5, 6]
>>> b
>>> a
>>> y
>>> x
30 20 ?? ??
```

---

Many list methods update a list as a **side effect** – and return None

```
>>> astr = 'abc'
>>> astr = astr.upper()
?? ??
>>> astr = 'abc'
>>> astr.upper()
>>> myl = [1, 2, 3]
>>> myl.reverse()
>>> astr
>>> myl
?? ??
>>> astr = 'abc'
>>> astr.upper()
>>> myl
```
Mutable data types

Aliasing is also an issue

>>> yourstr = mystr
>>> yourl = myl
>>> mystr = mystr.capitalize()
>>> myl.reverse()
?? ??
>>> yourstr ?? ??

Lists are a mutable data type # powerful, convenient

>>> myl = [True, 'Oregon', 99]
>>> id(myl)
4359098952

content can be changed after object is created

>>> myl.append([1, 2]) # list updated as a side effect
>>> myl
# of append method
[True, 'Oregon', 99, [1, 2]]
# append returns None

content is changed in place

>>> id(myl) # any aliases reflect the change
4359098952

>>> y = [1, 2, 3]
>>> id(y)
4331561040

>>> x = y
>>> id(x)
4331561040

>>> x = y.copy # or y[:]
>>> id(x)
4359639552

>>> y[0] = 99
>>> y[0] = 99

>>> id(y)
>>> id(y)
4331561040
4331561040

>>> x
>>> x
>>> id(x)
[99, 2, 3]
[1, 2, 3]
4359639552

>>> x is y
>>> x is y
True
False