CIS 210

Programming/Computer Science concepts

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

Good programming style: function docstrings (type contract; description including parameters, return types, and side effects if any; examples of function use), well-named variables, use of whitespace to indicate logical structure.

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, type, memory location, memory management, immutable data types.

Combining primitive elements: expressions, operators, expressions evaluate to a value, short-circuit evaluation.

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

Automatic approaches to testing and debugging; automated testing

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

Python toolkit so far

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

String data types and operations (e.g., +, len, count, find);

Boolean data types 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;

Assignment statement;

Python repetition - for, while;

Python conditionals - if;

Variable assignment;

User-defined functions: function design; docstrings;

IDLE interactive development environment; help function; dir, type, id; data analysis.

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, algorithm, pseudocode (not Python code);
-- check algorithm using your examples -- review 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: briefly describe;
-- write the function specification: 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.
>>> __name__

a) '__builtins__'
b) '__main__'
c) str
d) TypeError
e) dir()

>>> x = 99.9
>>> id(x)
4298470336

refers to

a) value of x
b) type of x
c) memory location of x
d) attribute of x
e) method of x

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

??

a) [1, 2, 3, 4]
b) ['hi', 1, 2, 3, 4]
c) [1, 2, 3, 4, 'hi']
d) TypeError
e) list

• Python collections / mutable data types

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)

Data Structures
Data structures are how we store and access data in a computer program. A data structure organizes the data and supports basic operations on the data (e.g., add, update, retrieve, delete).

Python collections – strings, tuples, lists, dictionaries
CIS 210

Python collections

Sequential
- Strings, Tuples, Lists

Unordered
- Dictionaries, Sets, Frozensets

Immutable
- Strings, Tuples, Frozensets

Mutable
- Lists, Dictionaries, Sets

---

CIS 210

Python collections - Sequential

<table>
<thead>
<tr>
<th>Strings</th>
<th>Tuples</th>
<th>Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered</td>
<td>ordered</td>
<td>ordered</td>
</tr>
<tr>
<td>sequential ops</td>
<td>sequential ops</td>
<td>sequential ops</td>
</tr>
<tr>
<td>characters</td>
<td>multiple types</td>
<td>multiple types</td>
</tr>
<tr>
<td></td>
<td>(including lists)</td>
<td>(including lists)</td>
</tr>
<tr>
<td>immutable</td>
<td>immutable</td>
<td>mutable</td>
</tr>
</tbody>
</table>

---

CIS 210

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

---

CIS 210

Recall: Updating a string:

```python
given = 'abc'
given.upper()  # 'ABC'
given = given.upper()  # 'ABC'
```

---

CIS 210

Recall: Updating a string:

```python
>>> astr = 'abc'
>>> astr.upper()
'ABC'
>>> astr
'a'
```
Recall:

```python
>>> x = 'xyz'
>>> x[0] = 'z'  # X
    ?? how do we get to x = 'zyz'
```

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

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

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)

```python
>>> y = ['a', True, 100]
>>> y[0] = '[1], [2,3]]
>>> y
[[[1], [2,3]], True, 100]
```

Lists are a mutable data type
(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

Can change the size of the list

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

>>> y = [1, 2, 3]
>>> id(y)
4353070720

>>> y = [99] + y[1:]
>>> id(y)
??

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

>>> y[0] = 99
>>> y
[99, 2, 3]
>>> id(y)
4331561040

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

>>> y = y.append(100)
>>> y
[99, 3, 100]

>>> y = [1, 2, 3]
>>> id(y)
4353070720

>>> y = [99] + y[1:]
>>> id(y)
??
Many list methods update a list as a side effect — and return None

>>> mystr = 'bye'
>>> myl = [1, False, 'hi']
?? >>>
>>> mystr = mystr.upper()
>>> myl.reverse()  # myl is modified

Recall:
And for list:

>>> b = 20
>>> y = [1, 2, 3]
>>> a = b
>>> x = y
>>> b = 30
>>> y = [4, 5, 6]
>>> b  # x is updated
30
>>> a  # y is updated
20

Now for list updated in place:

>>> astr = 'abc'
>>> x = [1, 2, 3]
>>> astr = astr.upper()  # astr is modified
>>> x
??

>>> astr = 'abc'
>>> myl = [1, 2, 3]
>>> myl.reverse()  # myl is modified
>>> myl
??

>>> astr = astr.upper()  # myl is modified
>>> myl
??

>>> astr
??

>>> x
??
CIS 210 – Mutable data types

**Aliasing is also an issue**

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

Lists are a **mutable** data type  # powerful, convenient

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

Content can be changed after object is created

```python
>>> myl.append([1, 2])  # list updated as a side effect
>>> myl
# of append method
[True, 'Oregon', 99, [1, 2]]
```

Content is changed in place

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

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

```python
x = y
>>> x = y.copy()  # or y[:]  # list updated as a side effect
>>> id(x)
4359639552
```

```python
>>> y[0] = 99
>>> y[0] = 99
>>> id(y)
4331561040
```

```python
>>> x is y
True
```

```
CIS 210

```python
def q1(s):
    """(str) -> str
    Return string with information about string length.
    >>> q1('hello, world')
    'hello, world-12'
    ""
    ct = len(s)
    s = s + ',' + ct
    return s
```

Executing function q1 will cause a

a) logic error  b) run time error – IndexError  c) run time error – TypeError

d) run time error – NameError  e) run time error – ZeroDivisionError

```python
def q1(s):
    """(str) -> str
    >>> q1('hello, world')
    'hello, world-12'
    ""
    ct = len(s)
    s = s + ',' + ct
    return s
```
RECALL: PARAMETER PASSING IS CALL BY ASSIGNMENT  → ALIASING

```python
def bar(x):
    def foo(li):
        y = foo(x)
        print('bar - x:', x, 'bar - y:', y)
        return y
    return None

x = ['CIS 210', 'CIS 211', 'CIS 212']
bar(x)
```

Lists (and dictionaries and sets) are mutable data types

- content can be changed after it is created
- content is changed in place
- content of any alias is also changed
- parameter passing creates an alias
- id function can help us see this
- copy object to avoid aliasing

```
>>> li1 = [1, 2, [3, 4], [5, 6]]
>>> li2 = li1.copy()
>>> li1[0] = 'hi'
>>> li1
['hi', 2, [3, 4], [5, 6]]
```

```python
>>> li1[2][0] = 999
>>> li1
['hi', 2, [999, 4], [5, 6]]
```

```python
>>> li2
[1, 2, [999, 4], [5, 6]]
```

```python
>>> li2 = li1.copy()
>>> li1[2][1] = True
>>> li1
['hi', 2, [999, True], [5, 6]]
```

```python
>>> li2
[1, 2, [999, True], [5, 6]]
```

```python
>>> li3 = copy.deepcopy(li)
>>> li3
['hi', 2, [999, 4], [5, 6]]
```

```python
>>> import copy
>>> li3 = copy.deepcopy(li)
>>> li3
['hi', 2, [999, 4], [5, 6]]
```
Python collections – Sequential

Lists – heterogeneous, mutable – are a very flexible and powerful data type: use wisely!

Is a list the best choice for representing data?

Does the data need to be changed?

No → tuple – safer and faster
Yes → list

Python tuples

Tuples are immutable, heterogeneous sequences of references to any object.

For example,

intseq = (10, 20, 30, 40, 50)
genseq = (10, 20.0, ‘a’, True)
nestedseq = (10, 20, (‘a’, ‘b’), True)

This would be handy

Storing items with labels, e.g., binary numbers with their decimal equivalents, or the name of a month with the corresponding number of days in the month.
Python collections - Dictionary

<table>
<thead>
<tr>
<th>Dictionary</th>
<th>Lists</th>
<th>Tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered*</td>
<td>ordered</td>
<td>ordered</td>
</tr>
<tr>
<td>key access</td>
<td>sequential ops</td>
<td>sequential ops</td>
</tr>
<tr>
<td>multiple types</td>
<td>multiple types</td>
<td>multiple types</td>
</tr>
<tr>
<td>(keys immutable)</td>
<td>(including lists)</td>
<td>(including lists)</td>
</tr>
<tr>
<td>mutable</td>
<td>mutable</td>
<td>immutable</td>
</tr>
</tbody>
</table>

```python
>>> binaryD = {}
>>> binaryD[0] = '00000000'
>>> binaryD[1] = '00000001'
>>> binaryD[2] = '00000010'
>>> binaryD[3] = '00000011'
>>> binaryD
{0: '00000000', 2: '00000010', 3: '00000011', 1: '00000001'}
```

```python
>>> binaryD[2]
'00000010'
```

```python
>>> import pprint
>>> pprint.pprint(binaryD)
```

```python
>>> MONTHS_DAYS = {'January': 31, 'February': 28,
    'March': 31, 'April': 30,
    'May': 31, 'June': 30, 'July': 31,
    'August': 31, 'September': 30,
    'October': 31,
    'November': 30,'December': 31}
```

```python
countd = {'CIS': 4, 'EXPL': 3}
```

```python
for k in countd:
    print(k); print(countd[k])
```

??

```python
>>> pprint.pprint(MONTHS_DAYS)
```

```python
>>> pprint.pprint(MONTHS_DAYS)
```
```python
>>> countd = {'CIS': 4, 'EXPL': 3}
>>> for k in countd:
    print(k); print(countd[k])
CIS
4
EXPL
3

>>> countd = {'CIS': 4, 'EXPL': 3}
>>> for k in countd:
    print(k); print(countd[k])
>>> for k in d:
    print(k, end=''); print(d[k])
>>> for k in d:
    print(f'{k}: {d[k]}')

CIS 210 List Comprehensions
(List declarative style programming)

Given:
S = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Generate:
T is a list of x such that x is a member of S and x is even

T = []
# procedural
for x in S:
    if even(x):
        T.append(x)

T = [x for x in S if even(x)]
# declarative (filter pattern)
```

```python
# map pattern
doubleli = []
for k in countd:
    doubleli.append(countd[k] * 2)

# map pattern
doubleli = [countd[k] * 2 for k in countd]
...
doubleli = []
for k in countd:
    doubleli.append(countd[k] * 2)
...
```

```python
>>> doubleli
??
```
List comprehensions are a concise way to create lists. The general syntax is:

\[
[\text{<expression>} \quad \text{# map} \\
\text{for <item> in <sequence> ...} \quad \text{# other collection(s)} \\
\text{if <condition>} \quad \text{# filtering (if needed)}
\]

Each item in the new list is the result of applying a given operation (\text{<expression>}) to a value (\text{<item>}) from a sequence (\text{<sequence>}).

T = \{x - 1 \text{ for } x \text{ in } S \text{ if even}(x)\}

\[
[2 * i \text{ for } i \text{ in } [1, 2, 3]]
\]