Midterm Summary:

Multiple choice (19 pts possible)
Avg: 13.62 (71.5%)
Min/Max: 5/19
From last time: Pointers to memory
>>> b = 20
>>> id(b)
4297645024

>>> a = b + 1

>>> b = 30

>>> b = 30
>>> b = 20
>>> id(b)
4297645024

>>> a = b + 1
>>> id(a)
4297645056

>>> b = 30
```python
>>> b = 20
>>> id(b)
4297645024
4297645056

>>> a = b + 1
>>> id(a)
4297645056
4297645344

>>> b = 30
>>> id(b)
4297645344
4297645344
```
CIS 210

```python
>>> b = 20
4297645024

>>> id(b)
4297645024

>>> a = b + 1
21

>>> id(a)
4297645056

>>> b = 30
30

>>> id(b)
4297645056

>>> b
??

>>> a
??
```

```python
>>> b
30

>>> id(b)
4297645344

>>> a
21
```

```python
>>> id(a)
4297645056
```
>>> b = 20
>>> a = b
>>> b = 40
>>> a

Evaluate rhs. **If it is a variable name:**

Let the variable on lhs refer to resulting object:

- **use address of rhs variable. no memory is allocated.**
- search current namespace – if name on lhs *is not* there:
  - assign name on lhs to address of memory location
  - add it to the current namespace
- if name on lhs *is* there:
  - replace old reference with new reference (address)
>>> b = 20
>>> id(b)
4297645024

>>> a = b
>>> b = 20
>>> id(b)
4297645024

>>> a = b
>>> id(a)
4297645024
CIS 210

```python
>>> b = 20
>>> id(b)
4297645024

>>> a = b + 1
>>> id(a)
4297645056

>>> b = 40
>>> id(b)
4297645664

>>> b
??
>>> a
??
```
reference semantics:
the name on the lhs is a reference (pointer) to the memory location of the data object

>>> id(msg)
4383118984
>>> id(greeting )
??
>>> msg = 'hello'
>>> msg
'hello'

>>> greeting = msg

>>> id(msg)
4383118984

>>> id(greeting )
greeting “aliases” msg
4383118984
>>> id(msg)
4383118984

>>> id(greeting)
4383118984

greeting "aliases" msg

'hello'
>>> msg = 'goodbye'

>>> msg
'goodbye'

when the variable is given a new value

>>> id(msg)
4383119544

the reference (pointer) is updated

>>> id(greeting)
4383118984

>>> greeting
??
>>> id(msg)
4383119544

>>> id(greeting)
4383118984

>>> greeting
'goodbye'

>>> greeting
'hello'

✘
CIS 210

```python
>>> msg = 'hello'
>>> msg = 'jello'
```

```python
>>> class = 'CIS 210'
>>> class = 'CIS 211'
```

```python
4388854056 4388753168
4332519528 4388847208
```

- `msg` is associated with 'hello' at 4388854056 and 'jello' at 4388753168.
- `class` is associated with 'CIS 210' at 4388854056 and 'CIS 211' at 4388753168.
garbage collection

• done by Python interpreter

• checks whether any variables are pointing at objects for which it has allocated memory

• if none are, the object is deleted and the memory is made available again
• closer look: Python objects/assignment
  ✓ dynamic typing
  ✓ strong typing
  ✓ Python objects – value(s), type, id
  ✓ reference semantics
  ✓ garbage collection
Programming/Computer Science concepts

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

Good programming style: function docstrings (type contract; description including parameters, returned value, and side effects if any; 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
Combining primitive elements: Expressions - expressions evaluate to a value; short circuit evaluation of boolean expressions; overloaded 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
Python toolkit so far

numeric data types (int, float) and operations (e.g., +, **, round, abs)
string data type and operations (e.g., +, len, 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

variable assignment
user-defined functions; function design; docstrings
IDLE interactive development environment; help function; dir, type, id
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 – 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

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.
REVIEW
>>> 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
Reading this week: Lists

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

a)  

b)  

[c)  

d)  

e)  

• 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.
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.

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
Python collections

Sequential
Strings, Tuples, Lists

Unordered
Dictionaries, Sets, Frozensets

Immutable
Strings, Tuples, Frozensets

Mutable
Lists, Dictionaries, Sets
# 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>

*Table 4.1 (text)*
Python collections – Sequential

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

```python
g>>> astr = 'abc'
g>>> astr.upper()
'ABC'
g>>> astr
```

??

a) ‘abc’
b) ‘ABC’
c) ‘Abc’
d) None
CIS 210: iClicker Question

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

>>> astr
'abc'

>>> astr = astr.upper()

>>> astr
??
   a) 'abc'
   b) 'ABC'
   c) 'Abc'
   d) None
```
>>> astr = 'abc'
>>> astr.upper()
'ABC'
>>> astr
'abc'

>>> astr = astr.upper()

>>> astr
??

 a) ‘abc’
 b) ‘ABC’
 c) ‘Abc’
 d) None
Recall:

```python
>>> x = 'xyz'

>>> x[0] = 'z'  # ✗ (not allowed)

?? how do we get to x = 'zyz'
```
Recall:

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

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

>>> x = 'zyz'
```

```python
>>> x
zyz
```
>>> x = ‘xyz’
>>> x[0] = ‘z’  ✗
>>> x = ‘z’ + x[1:]
>>> x =
zyz
>>> x
‘zyz’
>>> x
‘zyz’

COMPARE:
>>> y = [‘a’, True, 100]
>>> y[0] = ‘b’
>>> y
[‘b’, True, 100]
Lists are a mutable data type
(and strings and tuples are not)
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]
>>> y.append(101)
[[[1], [2,3]], True, 100, 101]
```
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

```python
>>> y.append(100)
>>> y
[99, 2, 3, 100]  >>> id(y)
4331561040
>>> y.remove(2)
>>> y
[99, 3, 100]  >>> id(y)
??
>>> y = [99, 3, 100]
>>> y
[99, 3, 100]  >>> id(y)
??
```
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]
```
```python
>>> y = [1, 2, 3]
>>> id(y)
4353070720

>>> y = [99] + y[1:]
>>> id(y)
??
```
CIS 210

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

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

different
>>> y
[99, 3, 100]

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

>>> id(y)
4324240840

?? (same or new?)

>>> y = y.append(100)

???
>>> y
[99, 3, 100]

>>> id(y)
4324240840

>>> y.append(100)

>>> y
[99, 3, 100, 100]

>>> id(y)
same

>>> y = y.append(100)

>>> y
??

>>> id(y)
different
Many list methods update a list as a *side effect* — and return `None`
Many list methods update a list as a side effect – and return None

```python
>>> mystr = 'bye'
>>> mystr[0] = 'r'
>>>
>>> mystr
>>> mystr = mystr.upper()
>>> mystr
>>> mystr = 'hello'
>>> yourstr = mystr
>>> mystr = mystr.capitalize()
>>> mystr; yourstr
```
Many list methods update a list as a **side effect** — and return None

```python
>>> x = [100, 101, 102, 103, 101]
>>> w = x.remove(101)
>>> y = x.pop()
>>> z = x.pop(2)
>>> a = x.append(99)
>>> print(x, w, y, z, a)
```
Recall:

```python
>>> b = 20
>>> a = b
>>> b = 30
>>> b, a = 30, 20
30 20
```

And for list:

```python
>>> y = [1, 2, 3]
>>> x = y
>>> y = [4, 5, 6]
>>> y, x = [4, 5, 6], [1, 2, 3]
?? ??
```
Recall:

>>> b = 20
>>> a = b
>>> b = 30

Now for list updated in place:

>>> y = [1, 2, 3]
>>> x = y
>>> y[1] = 99

>>> b
30

>>> a
20

>>> y
??

>>> x
??
Many list methods update a list as a side effect – and return None

```python
>>> astr = 'abc'
>>> astr = astr.upper()
>>> astr

??
```

```python
>>> astr = 'abc'
>>> astr.upper()
>>> astr

??
```

```python
>>> myl = [1, 2, 3]
>>> myl.reverse()
>>> myl

??
```

```python
>>> myl = [1, 2, 3]
>>> myl.reverse()
>>> myl

??
```
Aliasing is also an issue

```python
>>> yourstr = mystr
>>> mystr = mystr.capitalize()
>>> yourstr
??
>>> mystr
??
```
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
[True, 'Oregon', 99, [1, 2]]  # of append method
# 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)
4331561040

>>> x
[99, 2, 3]
>>> x is y
True

>>> x = y.copy() # or y[:]
>>> x
[1, 2, 3]
>>> id(x)
4359639552

>>> x is y
False
def q1(s):
    '''(str) -> str

    Return string with information about string length.

    >>> q1('hello, world')
    'hello, world-12'
    '''

c = len(s)
s = s + '-' + c
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
def q1(s):
    '''(str) -> str

    >>> q1('hello, world')
    'hello, world-12'
    '''
    ct = len(s)
    s = s + '-' + ct
    return s

The error in function q1 stems from what characteristic of Python?

a) static typing  
b) strong typing  
c) parameter passing  
d) None value  
e) side effect
>>> roman = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'M': 1000}
>>> roman['X']
??

a) X    b) 'X'    c) 10    d) [10]    e) TypeError
def bar(x):
    y = foo(x)
    print('bar-x:', x, 'bar-y:', y)
    return None

def foo(li):
    y = li.pop()
    return y

x = ['CIS 210', 'CIS 211', 'CIS 212']
bar(x)
bar(x)
z = x.copy()
bar(z)
print('z:', z)
print('x:', x)
Good Morning CIS 210
Midterm Summary:

Multiple choice (19 pts possible)
Avg: 13.62 (71.5%)
Min/Max: 5/19

Short answer (6 pts possible)
Avg: 4.5 (75%)
Min/Max: 1/6

Code (10 pts possible)
Avg: 8.2 (82%)
Min/Max: 2/10
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.

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
Python collections

Sequential
Strings, Tuples, Lists

Unordered
Dictionaries, Sets, Frozensets

Immutable
Strings, Tuples, Frozensets

Mutable
Lists, Dictionaries, Sets
## 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><strong>multiple types</strong></td>
<td><strong>multiple types</strong></td>
</tr>
<tr>
<td></td>
<td>(including lists)</td>
<td>(including lists)</td>
</tr>
<tr>
<td>immutable</td>
<td>immutable</td>
<td><strong>mutable</strong></td>
</tr>
</tbody>
</table>

Table 4.1 (text)
Recall: Updating a string:

```python
>>> astr = 'abc'
>>> astr.upper()
'ABC'
>>> astr
a) 'abc'
b) 'ABC'
c) 'Abc'
d) None
```
def bar(x):
    y = foo(x)
    print('bar-x:', x, 'bar-y:', y)
    return None

def foo(li):
    y = li.pop()
    return y

x = ['CIS 210', 'CIS 211', 'CIS 212']
bar(x)
bar(x.copy())
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 - use copy object to avoid aliasing
Nested lists

>>> myl
[True, 'Oregon', 99, [1, 2]]

>>> myl[0]
??

>>> myl[3]
??

>>> myl[3][1]
??
```python
>>> li1 = [1, 2, [3, 4], [5, 6]]
>>> li2 = li1.copy()
>>> li1[0] = 'hi'
>>> li1
['hi', 2, [3, 4], [5, 6]]
>>> li2
[1, 2, [3, 4], [5, 6]]
>>> li1[2][0] = 999
>>> li1
['hi', 2, [999, 4], [5, 6]]
>>> li2
[1, 2, [999, 4], [5, 6]]
```
>>> li1 = [1, 2, [3, 4], [5, 6]]
>>> li2 = li1.copy()
>>> li1[0] = 'hi'
>>> li1
['hi', 2, [3, 4], [5, 6]]
>>> li2
[1, 2, [3, 4], [5, 6]]
>>> li1[2][0] = 999
>>> li1
[7, 2, [999, 4], [5, 6]]
>>> li2
[1, 2, [999, 4], [5, 6]]
>>> import copy
>>> li3 = copy.deepcopy(li1)
>>> li1[2][1] = True
>>> li1
['hi', 2, [999, True], [5, 6]]
>>> li3
['hi', 2, [999, 4], [5, 6]]
Lists – heterogenous (multiple types), 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.
Python tuples

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

For example,

```python
intseq = (10, 20, 30, 40, 50)
genseq = (10, 20.0, 'a', True)
nestedseq = (10, 20, ('a', 'b'), True)
```
Python tuples

For example,

```python
>>> intseq = (10, 20, 30, 40, 50)
>>> genseq = (10, 20.0, 'a', True)
>>> nestedseq = (10, 20, ('a', 'b'), True)
>>> shortseq = (99,)

>>> intseq[0]
>>> genseq[1:4]
>>> nestedseq[2]
>>> nestedseq[2][1]

>>> len(nestedseq)
>>> len(nestedseq[2])

>>> intseq + nestedseq
>>> 'a' in nestedseq
```
Python tuples

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

For example,
nestedseq = (10, 20, ('a', 'b'), True)

```python
>>> a = nestedseq[2][0]
>>> b = nestedseq[2][1]
>>> (a, b) = ('c', 'd')
```

```python
>>> a, b = b, a
```

```python
>>> a = nestedseq[2][0]
>>> b = nestedseq[2][1]
```
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.
<table>
<thead>
<tr>
<th>Python collections - Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dictionary</strong></td>
</tr>
<tr>
<td>unordered*</td>
</tr>
<tr>
<td>key access</td>
</tr>
<tr>
<td>multiple types (keys immutable)</td>
</tr>
<tr>
<td>mutable</td>
</tr>
</tbody>
</table>
Python collections - Dictionary

**Dictionary** | **Lists** | **Tuples**
---|---|---
unordered* | ordered | ordered
key access | sequential ops | sequential ops
multiple types | multiple types (including lists) | multiple types (including lists)
(immutable, keys immutable) | | |
mutable | mutable | immutable
Python collections – Dictionary

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

dict_values(['00000000', '00000001', '00000010', '00000011'])

>>> for item in binaryD.values():
    print(item)

00000000
00000011
00000010
```python
>>> binaryD.keys()
dict_keys([0, 1, 2, 3])

>>> binaryD.items()
dict_items([(0, '00000000'), (1, '00000001'), (2, '00000010'), (3, '00000011')])
```
>>> 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}

>>> import pprint

>>> pprint.pprint(MONTHS_DAYS)
countd = {'CIS': 4, 'EXPL': 3}

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

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

>>> for k in countd:
    print(k); print(countd[k])

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

>>> for k in countd:
    print(k); print(countd[k])

>>> for k in countd:
    print(k, end=''); print(countd[k])

>>> for k in countd:
    print(f'{k}: {countd[k]}')
```
List Comprehensions
(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
S = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

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

\{x \mid x \in S \text{ and } x \text{ is even}\} \quad \#\text{mathematical}

T = [] \quad \#\text{procedural}
for x in S:
    if even(x):
        T.append(x)

T = [x for x in S if even(x)] \quad \#\text{declarative (filter pattern)}
List comprehensions

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

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

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

# map pattern
doubleli = [countd[k] * 2 for k in countd]

'''
doubleli = []
for k in countd:
    doubleli.append(countd[k] * 2)
'''

>>> doubleli
??
List comprehensions

List comprehensions are a concise way to create lists. The general syntax is:

```
[<expression>]
for <item> in <sequence> ...
if <condition>]
```

# map
# other collection(s)
# filtering (if needed)

T = [x - 1 for x in S if even(x)]
List comprehensions

List comprehensions are a concise way to create lists. The general syntax is:

```
[<expression> # map
for <item> in <sequence> ... # other collection(s)
if <condition>] # filtering (if needed)
```

Each item in the new list is the result of applying a given operation (<expression>) to a value (<item>) from a sequence (<sequence>).
List comprehensions are a concise way to create lists. The general syntax is:

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

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

```python
>>> [2 * i for i in [1, 2, 3]]
??
```