CIS 210

• closer look: Python objects/assignment
  ✓ dynamic typing
  ✓ strong typing
  ✓ Python objects – value(s), type, id
  ✓ reference semantics
  ✓ garbage collection
  ✓ namespaces
  ✓ variable scope

→ Python collections / mutable data types
→ intro to data analysis

CIS 210

VARIABLE SCOPE

Recall:
When we start Python, two namespaces are created – the built-in namespace and a global (__main__) namespace.

When we create names (e.g., variables, function definitions) in a Python session, they are added to __main__.

When Python executes a function, a local namespace is created to keep track of function variables (activation record / call stack).

CIS 210

SCOPE:  RECALL

Scope refers to the visibility of variables

Python searches namespaces in this order:

Local (if there is one)

Global

Built-in

CIS 210

SCOPE

def twice(x):
  '''
  # parameters are local
  y = 2
  result = y * x
  # so is result
  print(dir())
  # dir() returns current scope:
  print(result, x, y)
  return

>>> y = 5
>>> twice(y)
5
['result', 'x', 'y']
>>> x
10, 5, 2
NameError

CIS 210

SCOPE

def twice(x):
  '''
  result = y * x
  # no local y ->
  # find global y
  # do this sparingly!!
  return

>>> y = 5
>>> twice(y)
??
>>> y >>> x >>> twice # global again
def twice(x):
    """
    result = y * x       # no local y ->
    return result       # find global y
    # do this sparingly!!
    """

In most cases where you are tempted to use a global variable, it is better to use a parameter for getting a value into a function or return a value to get it out.

def thrice_ish(x):
    """
    x += 1
    m = 3
    return m * x
    """

>>> x = 5
>>> x
5
>>> thrice_ish(x)
15
>>> m
NameError

def test1(a):
    """
    a += 5
    return a
    """

>>> a = 6
>>> test1(a)
11
>>> a
11

def test2(b):
    """
    print(b)
    print(a)
    print(a)
    return
    """

>>> test3(14)
>>> test2(14)
>>> a
11
>>> b
11

def test3(b):
    def test4(a):
        test4(b)
        a += 5
        return b + 1
        print(a, b)
        return

    >>> test3(99)
    NameError: name 'b' is not defined

    local namespaces are on the same level -
    static (lexical) scoping
CIS 210

SCOE

def test3(b):
    def test4(a):
        a += 5
        print(a, b)
        return
    test4(b)
    return b + 1

>>> test3(99)
NameError: name 'b' is not defined
>>> b = 1
>>> test3(99)
104 1

printed in test4

100 returned from test3

CIS 210

MORE NAMESPACES AND SCOPE

IMPORT

Recall: at Python startup – two namespaces:

__builtins__
__main__  (global)

>>> dir()
['__annotations__', '__builtins__', '__doc__',
 '__loader__', '__name__', '__package__', '__spec__']

>>> __name__
'__main__'

CIS 210

Recall: at Python startup – two namespaces:

__builtins__
__main__  (global)

>>> dir()
['__annotations__', '__builtins__', '__doc__',
 '__loader__', '__name__', '__package__', '__spec__']

>>> __name__
'__main__'

CIS 210 / Welcome

Recall:
When we start Python, two namespaces are created – the built-in namespace and a global (__main__) namespace.

other objects can be imported from Python modules (libraries)

CIS 210

More Python functions and values are available in modules (.py files) in the Python Standard Library

Accessing the Python Standard Library

    from math import *
    dir()

    from math import pi
    dir()

    import math
    dir()
>>> x = 99
>>> dir()
['__annotations__', '__builtins__', '__doc__',
__loader__, '__name__', '__package__', '__spec__', 'x']

>>> import math  # a new global namespace for the
# imported module is created
# namespace name is module name

>>> dir()
['__annotations__', '__builtins__', '__doc__','__loader__', '__name__', '__package__', '__spec__', 'math', 'x']

A reference to the imported module is added to the
__main__ namespace.

>>> sqrt(81)
math.sqrt(81)

A reference to the imported function is added to the
__main__ namespace.

>>> from math import pi

>>> dir()
['__annotations__', '__builtins__', '__doc__','__loader__', '__name__', '__package__', '__spec__', 'math', 'x', 'pi']

A reference to the imported function is added to the
__main__ namespace.

>>> pi
math.pi

>>> from turtle import fd

>>> dir()
['__annotations__', '__builtins__', '__doc__','__loader__', '__name__', '__package__', '__spec__', 'fd',
'math', 'x', 'pi']

>>> turtle.bk(50)  >>> turtle.fd(50)  >>> fd(50)
??  ??  ??

>>> from turtle import *

>>> dir()
['__annotations__', '__builtins__', '__doc__','__loader__', '__name__', '__package__', '__spec__', 'bk',
'fd', [all turtle functions and values], 'math', 'x', 'pi']

>>> turtle.bk(50)  >>> bk(50)  >>> fd(50)
??  ??  ??
CIS 210

• closer look: Python objects/assignment
  ✓ dynamic typing
  ✓ strong typing
  ✓ Python objects – value(s), type, id
  ✓ reference semantics
  ✓ garbage collection
  ✓ namespaces
  ✓ variable scope

→ Python collections / mutable data types
→ intro to data analysis

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)

CIS 210

Python collections
Sequential (ordered – index access)
Strings, Tuples, Lists
Key access (insertion ordered)
Dictionaries
Unordered
  Sets, Frozensets
Immutable
  Strings, Tuples, Frozensets
Mutable
  Lists, Dictionaries, Sets
### Python collections – Sequential – Index access

<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>

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

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

### Recall:
```python
>>> x = ‘xyz’
>>> x[0] = ‘z’
>>> 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)

>>> x = 'xyz'
>>> x[0] = 'z'
✗

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)

Python updates the object IN PLACE

• closer look: Python objects/assignment
  ✓ dynamic typing
  ✓ strong typing
  ✓ Python objects – value(s), type, id
  ✓ reference semantics
  ✓ garbage collection
  ✓ namespaces
  ✓ variable scope

→ Python collections / mutable data types
  lists, tuples, dictionaries
  ✓ intro to data analysis

So the day before the election, something happened with the fivethirtyeight website where it changed to show Trump with a 99% percent chance of winning, which was not actually what their model had. Turns out it was a bug. The issue is that their model weights non-poll data as some fraction, and that fraction goes to zero on the day before the election. Turns out when that happened, it drove some numbers to zero in their model which caused regular old div-zero bugs. The if-statement that was supposed to guard against it was off-by-one, kicking in on election day, when it should have been the day before the election.

The story appears [about 4 minutes 20 seconds in approx. 51:40] of the podcast linked below. Note that there is a certain amount of swearing [bleeped out]!

— Nick Parlante, Stanford University, Computer Science

https://fivethirtyeight.com/features/politics-podcast-final-reflections-on-the-2020-campaign-before-election-day/

multiple, related bugs:
  division by zero error (except logic not runtime error)
  fencepost error (off-by-one for election day checking)
missed equivalence class for testing: non-election day, election day, day before election day
data ethics consideration:
  “... stuff happens. That’s why you have to have live humans looking at models.”
Lists are mutable. They can be updated in place.

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

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

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.append(100)
>>> y
[99, 3, 100, 100]
>>> id(y)
4331561040
```

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

>>> y = [99] + y[1:]
>>> id(y)
4324240840
```

```
>>> y = [99, 3, 100]
>>> y.append(100)
>>> y
[99, 3, 100, 100]
>>> id(y)
4390703048
```

```
>>> y
[99, 2, 3]
>>> id(y)
4324240840

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

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. COMPARE:

```python
>>> mystr = 'bye'
>>> mystr[0] = 'r'
>>> mystr
>>> mystr = mystr.upper()
>>> mystr
>>> yourstr = mystr.capitalize()
>>> yourstr
```

```python
>>> x = [100, 101, 102, 103, 101]
>>> w = x.remove(101)
>>> y = x.pop()
>>> z = x.pop(1)
>>> a = x.append(99)
>>> print(x, w, y, z, a)
```

Recall: And for list:

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

Recall: Now for list updated in place:

```python
>>> b = 20
>>> a = b
>>> y = [1, 2, 3]
>>> x = y
>>> y[1] = 99
>>> a
30 20 ?? ??
```

```python
>>> astr = 'abc'
>>> astr.upper()
>>> myl
>>> astr = astr.upper()
>>> myl.reverse()
>>> astr
>>> myl.reverse()
```
CIS 210 – Mutable data types

Aliasing is also an issue

```python
>>> yourstr = mystr
>>> mystr = mystr.capitalize()
>>> mystr
??
>>> yourstr
??
```

```python
def f(li):
    '''
    li.insert(0, 99)
    return
    '''
    >>> a = [1, 2, 3]
    >>> f(a)
    >>> a
    [99, 2, 3]
    >>> x = y.copy() # or y[:] or list(y)
    >>> id(y)
    4331561040
    >>> id(x)
    4359639552 (new)
    >>> y[0] = 99
    >>> id(y)
    4331561040
    >>> x = y
    >>> id(x)
    >>> x
    [99, 2, 3]
    >>> x is y
    True
    >>> a
    [1, 2, 3]
```

CIS 210 – Lists

Lists are a mutable data type  # powerful, convenient
content can be changed after object is created
content is changed in place
```python
>>> myl = [True, 'Oregon', 99]
>>> id(myl)
4359098952

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

>>> id(myl)
4359098952 # call-by-assignment parameter passing
```

CIS 210 – RECALL: PARAMETER PASSING IS CALL BY ASSIGNMENT  ALIASING

def bar(x):
    y = foo(x)
    print('bar-x:', x)
    return y

def foo(li):
    y = li.pop()
    print('foo-y:', y)
    return

x = ['CIS 210', 'CIS 211', 'CIS 212']
x = x[:]
bar(x)
z = x.copy() # z = x[:]
bar(z)
print('global-z:', z)
print('global-x:', x)
```

CIS 210

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
```python
id function (also is) can help us see this
copy object to avoid aliasing
```
Nested lists

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

```python
>>> myl[0]
??
>>> myl[3]
??
>>> myl[3][1]
??
```

### Python collections – Sequential – Index access

Lists – heterogenous, 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,

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

```python
>>> a = nestedseq[2][0]
>>> x = 10,000
>>> b = nestedseq[2][1]
>>> (a, b) = (a * 2, b * 2)
>>> (a, b) = (b, a)
```

```python
>>> intseq[0] >>> len(nestedseq)
>>> genseq[1:4] >>> len(nestedseq[2])
>>> nestedseq[2] >>> intseq + nestedseq
>>> nestedseq[2][1] >>> 'a' in nestedseq
```
CIS 210

- closer look: Python objects/assignment
  - dynamic typing
  - strong typing
  - Python objects – value(s), type, id
  - reference semantics
  - garbage collection
  - namespaces
  - variable scope

  ➔ Python collections / mutable data types
  - lists, tuples, dictionaries
  - intro to data analysis

CIS 210 / Welcome

CIS 210 is a community of learners where

- Everyone is welcome
- Everyone is respected
- We value intellectual challenges and deliberate practice in pursuit of new knowledge and skills
- We support and encourage each other
- We celebrate our own and each other’s accomplishments

CIS 210 / Programming/Computer science concepts (week 4)

- Conceptual framework: imperative vs. functional programming
- Computational problem solving: design, implement, analyze, test, review
- Problem solving: data analysis, algorithms; use of mathematical/formal methods and abstractions of code
- Python is a programming language and Python is an interpreter (program, execution)

CIS 210 Review (week 5)

- Python objects (reference diagrams, type, memory management, garbage collection)
- Functions
- Functions as arguments
- Functions as return values
- Functions as side effects
- Functions as expressions

CIS 210 Learning Outcomes

- 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.