COMP 1510 Winter 2022

Programming Methods

Week 05: Identity vs equality, function objects, debugging

1

Agenda for week 5

1. Lists and identity vs equality
2. Copying (deep vs shallow copies)
3. Memory management and garbage collection
4. Tuples
5. Passing functions to functions as objects
6. filter and map
7. Intro to debugging
8. Debugging with PyCharm

2 February 2022 COMP 1510 202210

2

LISTS AND IDENTITY VS EQUALITY

3

Recall the membership operators

- A common task is to determine if a container contains a specific value
- Python uses the membership operators **in** and **not in**
- These operators return true if the left operand matches the value of some element in a container

```
>>> colours = ['red', 'white', 'cerulean']
>>> print('cerulean' in colours)
>>> print('magenta' not in colours)
```


4

Identity

- We can use the identity operators ***is*** and ***is not*** to determine identity
- Two variables that refer to the same object are aliases (they have the same identity aka they store the same reference)

```
>>> x = 5
>>> y = 5
>>> x is 5
True
>>> x is not y
False
```


5

COPYING (DEEP VS SHALLOW)

6

Copying a list aka making a new list object

```
>>> numbers = list(range(1, 6))
>>> numbers
[1, 2, 3, 4, 5]
>>> same_numbers = numbers[:]
>>> same_numbers
[1, 2, 3, 4, 5]
>>> numbers.append(6)
>>> numbers
[1, 2, 3, 4, 5, 6]
>>> same_numbers
[1, 2, 3, 4, 5]
```


7

Compare to these aliases for the same object

```
>>> numbers = list(range(1, 6))
>>> numbers
[1, 2, 3, 4, 5]
>>> exact_same_numbers = numbers
>>> exact_same_numbers
[1, 2, 3, 4, 5]
>>> numbers.append(6)
>>> numbers
[1, 2, 3, 4, 5, 6]
>>> exact_same_numbers
[1, 2, 3, 4, 5, 6]
```


8

What if I copy a list of mutable things?

```
>>> even = list(range(0, 10, 2))
>>> odd = list(range(1, 9, 2))
>>> numbers = [even, odd]
>>> numbers
[[0, 2, 4, 6, 8], [1, 3, 5, 7]]
>>> split_copy = numbers[:]
>>> split_copy
[[0, 2, 4, 6, 8], [1, 3, 5, 7]]
>>> negative_evens =
        list(range(0, -10, -2))
>>> numbers.append(negative_evens)
>>> numbers
[[0, 2, 4, 6, 8], [1, 3, 5, 7], [0, -2, -
4, -6, -8]]
>>> split_copy
[[0, 2, 4, 6, 8], [1, 3, 5, 7]]
>>> split_copy[1].append(9)
>>> split_copy
[[0, 2, 4, 6, 8], [1, 3, 5, 7, 9]]
>>> numbers
[[0, 2, 4, 6, 8], [1, 3, 5, 7, 9], [0, -2,
-4, -6, -8]]
>>>
```


Copies in programming are deep or shallow

- A shallow copy just copies the data structure and the addresses it contains, but it doesn't copy the structures at those addresses
- A deep copy makes a copy of the list, and then visits each address in the old list and makes a copy of the object it finds, and so on…
- Assignment statements in Python do not copy objects, they copy addresses
- Of course (of course!) there is a module in Python to help us with this: the copy module

The copy module

- https://docs.python.org/3.9/library/copy.html
- There are two functions we can use with collections that are mutable or that contain elements that are mutable:

1. copy.copy(x) will return a shallow copy of x – it constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
2. copy.deepcopy(x) will return a deep copy of x – it constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

11

Should we ever loop through a copy of a list?

- **Rule of thumb in programming: avoid adding elements to a list while you're looping through it!**
- ***It's better to make a copy of the list, and loop through the copy***
- We can add and remove elements to/from the original list
- Remember we can copy a list using a slice of the whole list: list_name[:]

12

MEMORY MANAGEMENT AND GARBAGE COLLECTION

Do you remember doing this?

```
nums = list(range(-300, 300))
for i in nums:
    print(i, id(i), id(i + 1) - id(i))
```

What does the output tell us?

The interpreter caches some immutable literals

cache

/kaSH/

Noun

- A selection of items of the same type stored in a hidden or inaccessible place

Verb

- **Store away** in hiding or **for future use**.

About all those objects in memory…

- A **variable** is just an identifier
- An identifier is just an easily remembered way to reach an object
- We know the ***variable actually stores the address of the object***
- Programmers will often call this a **reference**
- An object in memory can have lots of references (names)
- Every object tracks how many references (names) it has
- That is, every object keeps a reference count.

A reference count

- The Python interpreter maintains a reference count for each object
- We can get an object's reference count using getrefcount( )
- When we invoke getrefcount( ), we create a temporary reference as the argument, so getrefcount( ) will always return one more than we expect:

```
>>> import sys
>>> hashtag = "#freebritney"
>>> sys.getrefcount(hashtag) # == 2
```


So back to reference count, Chris…

- We increase the reference count by assigning the address of an object in memory to another variable
- We call this creating an alias of course!
- Each **variable** that points to the object increases the reference count by one
- **Local variables** contribute to reference counts
- When we reach the end of a function and a local variable goes out of **scope**, the reference count decrements by 1

Non-local variable reference count

- What about a non-local variable?
- Objects in the **global namespace** never go out of scope
- The reference count never goes down to zero
- Programmers tend to **avoid putting large objects or complex objects in the global namespace**
- We are stingy with memory and sometimes actively mindful of reference counts when programming
- We are always on the lookout for ways to make code efficient

19

Garbage collection

- A way for the interpreter to automatically release (free up) memory so it can be reused
- Identifies and erases objects that are no longer in use
- The interpreter uses reference counts as one of the tools for determining whether an object can be deleted from memory
- It also uses complicated things that we won't examine with names like:
  - Tracing
  - Generational garbage collection
  - Cyclical references.

20

TUPLES

Tuples are immutable lists!

- A ***tuple***, usually pronounced "tuh-ple" or "too-ple", behaves like a list but is immutable – once created the tuple's elements can not be changed
- A tuple is also a sequence type, supporting len(), indexing, and the other sequence type operations and methods
- A new tuple is generated by creating a list of comma-separated values, such as 5, 15, 20
- Typically, tuples are surrounded with **parentheses**, as in (5, 15, 20) but if we forget Python will put them there for us. This can be perplexing for new developers!
- Note that printing a tuple always displays surrounding parentheses.

Tuples

- Tuples are not as common as lists in practical usage
- ***Useful when a programmer wants to ensure that values do not change (constant values!)***
- Tuples are typically used when *element position*, and not just the relative ordering of elements, is important too
- For example, a tuple might store the latitude and longitude of a landmark because a programmer knows that the first element should be the latitude, the second element should be the longitude, and the landmark will never move from those coordinates:

23

Example

```
parliament_hill_coords = (45.4236, 75.7009)
print('Coordinates:', parliament_hill_coords)
print('Tuple length:', len(parliament_hill_coords))

# Access tuples via index
print('\nLatitude:', parliament_hill_coords[0], 'north')
print('Longitude:', parliament_hill_coords[1], 'west\n')

# Error. Tuples are immutable
parliament_hill_coords[1] = 50
```


24

PASSING FUNCTIONS TO FUNCTIONS AS OBJECTS

25

FILTER AND MAP

26

The map( ) function

- The map function accepts two parameters:
  1. A function
  2. A list
- It applies the function to everything in the list
- We like to do this because it's faster than a loop
- The map function returns an iterator object (coming soon!)
- We can pass the iterator to the list( ) function to get the result

A map example (it's so easy!)

```
def double(number):
    return number * 2

some_values = [1, 2, 3]
mapped_values = map(double, some_values)
new_list = list(mapped_values)

print(new_list)
>>> [2, 4, 6]
```

The filter( ) function

- The filter function also accepts two parameters:
  1. A *predicate* function that acts as a filter
  2. A list
- It applies the filter to the things in the list
- We like to do this because it's faster than a loop
- It returns an iterator object too!
- We can pass the iterator to the list( ) function to get the result

A filter example (it's so easy!)

```
def odd(number):
    return number % 2 != 0

data = [1, 2, 3, 4, 5]

list(filter(odd, data))
>>> [1, 3, 5]
```

STEPS FOR MAKING GREAT FUNCTIONS

31

How do we create functions?

- Writing a good essay requires planning:
  - Deciding on a topic
  - Researching background material
  - Writing an outline
  - Filling in the outline and revising drafts until we are done…
- Writing a good function also requires planning

32

Answer these questions before coding

1. What do you want to ***name*** the function?
2. What are the ***parameters*** (input) and what types of information do they refer to?
3. What ***calculations*** do you need to do with that information?
4. What information does the function ***return***?
5. Are there any ***side-effects***?

Step 1: define the function header

1. Define the function header
   a) Before we write anything, we need to decide:
      a) What information we have (the argument values, if any)
      b) What information we want the function to produce (return value)
   b) Select identifiers for the function and its parameters
      a) Meaningful names
      b) Avoid abbreviations
   c) Include type annotations (coming later today!)

Step 2: design some simple examples

2. Create some examples
   a) Include a few example calls and return values
   b) Ignore HOW the function will work
   c) Consider only WHAT the function ACCEPTS and RETURNS
   d) You can use these for your doctests
   e) This will help you identify exactly what the function will do
   f) (This is what you did for the midterm!)

Step 3: develop a description

3. Description
   a) Write a short informal paragraph that describes the function. The first word of the first sentence must be a verb that describes the main function of the (ahem) function!
   b) Parameters
   c) Precondition(s)
   d) Postcondition(s)
   e) Return value
   f) Use this information to create your docstring
   g) ALERT: does your function require a long description? Can your function be decomposed into smaller functions?

Steps 4 and 5: body and testing

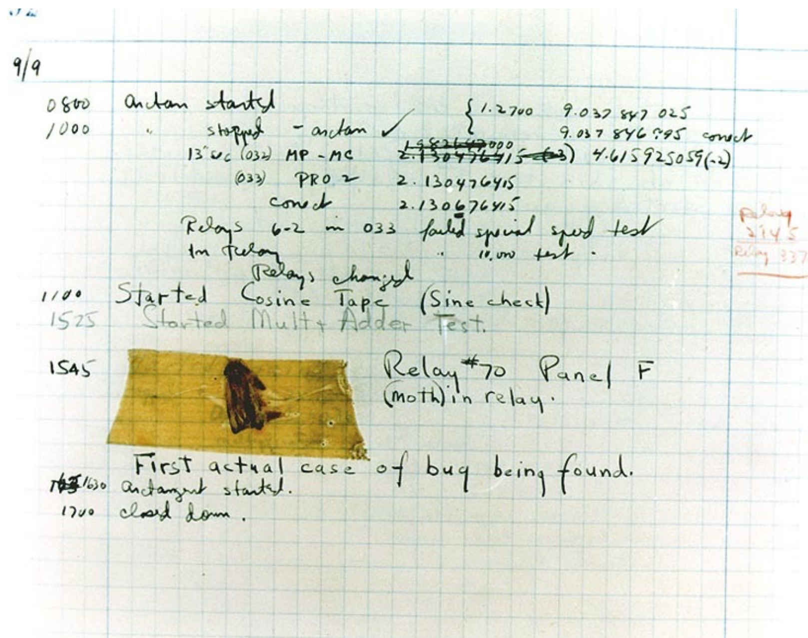
4. Body
   a) Implement the function
5. Testing
   a) Run your doctests to ensure they work
   b) Create unit tests
      a) Use the precondition(s) to identify valid starting conditions
      b) Divide the valid input/parameters into disjointed equivalency partitions
      c) Decide how the function should process each partition
      d) Write a single unit test with a single assertion to test each "quality" we want to ensure is correct. Test boundaries too!

37

DEBUGGING

38

Aside: the first actual bug



Computer log entry from the Mark II with a moth taped to the page Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988. - U.S. Naval Historical Center Online Library Photograph

How to debug code

We have many tools we can use to debug our code, including:

1. Pair Programming
2. Walkthroughs
3. Print statements
4. Debuggers

Pair Programming

- Two programmers work together at one workstation
- The driver types the code
- The navigator watches, reviews each line of code as it is entered
- The two programmers switch roles frequently
- Benefits:
  - **Design quality**
  - Satisfaction and learning
  - Team-building and communication

https://en.wikipedia.org/wiki/Pair_programming

Walkthroughs (often manual!)

- Tabulate the values of key variables on paper
- Document changes during and after each function call
- **Verbal Walkthrough** is when you describe your code to a peer
  - They might spot the error
  - The process of explaining might help you to spot it for yourself
- Formal group-based processes exist for conducting formal walkthroughs /inspections

Print Statements

- *The most popular technique for beginners*
- Ensure you use them in the correct places in the correct methods
- Output may be very long, so be precise and informative!
- Turning off and on requires forethought

Debuggers

- Debuggers are both language- and environment-specific
- Debuggers let us:
  - Set breakpoints
  - Execute line by line using Step and Step-into controlled execution
  - Track the call sequence (stack)
  - Monitor variable value and (later this term) object state.

DEMO TIME!

45

That's it for week 05!

Any questions?

46