PYTHON SEQUENCE DATA AND REFERENCES

CSC 109 - Introduction to programming in Python - Summer 2023

Marcus Birkenkrahe

June 9, 2023

Contents

Figure 1: Llyfrgell Genedlaethol Cymru / Longden Bay Go-cart Racers (1953)

1 Overview

- Sequence data types: tuples, lists, strings, range objects
- Mutable and immutable data types
- References and variables for integers
- References and variables for lists
- Memory identity function
- Passing references in function calls
- Copying functions from the copy module

2 Sequence data types

- Python's sequence data include:
  1. `list` objects
  2. string (`str`) objects (sequence of characters)
  3. range objects returned by `range`
  4. `tuple` objects
- These data types support *indexing* and *slicing*, and you can *iterate* over them in a specific order.
- On the other hand, `set` data are not sequence data.
- Here are some examples of indexing, slicing, iterating with other sequence data, e.g. strings:

```
name = 'Alice'
print(name[0])
print(name[:])
print('A' in name)
print('Al' not in name)
for i in name:
    print(f'*** {i} ***')
```

```
A
Alice
True
False
*** A ***
*** l ***
*** i ***
*** c ***
*** e ***
```

3 Mutable and immutable data types

- A `list` and a string differ in an important way:
  1. A `list` value is a *mutable* data type - its values can be added, removed, or changed.
  2. A string (`str`) value is *immutable* (or *literal*): it cannot be changed:

     ```
     name = 'Jack a cat.'
     name[6] = 'the'
     ```

- To mutate a string, you need to slice and concatenate to build a new string from parts of the old string.

- Transform `name = 'Jack a cat'` into `newName = Jack the cat`:

  ```
  name = 'Jack a cat'
  newName = name[0:5] + 'the' + name[-4:]
  print(newName)
  print(name)

  Jack the cat
  Jack a cat
  ```

- What is the output of this code:

  ```
  eggs = [1, 2, 3]
  eggs = [4, 5, 6]
  print(eggs)

  [4, 5, 6]
  ```

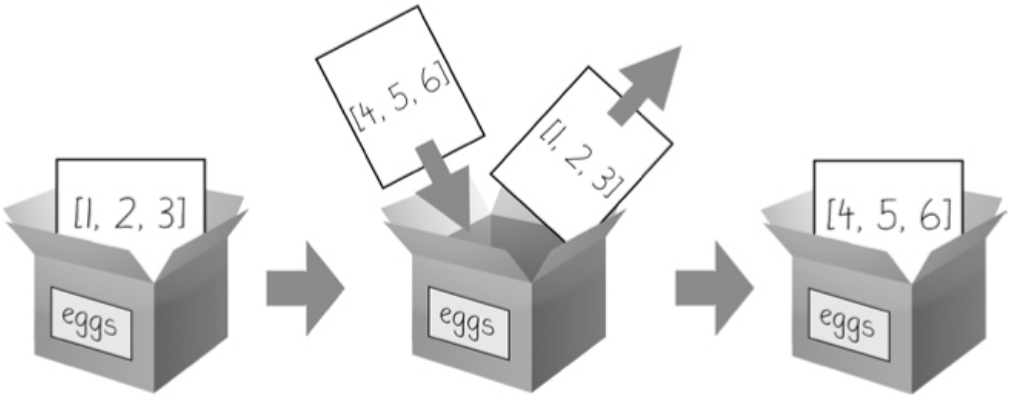


Figure 2: The contents of the list are replaced

- This is what happens:
- To modify the *original* list in `eggs` to contain `[4,5,6]` *in place*, you'd have to go out of your way:

```
eggs = [1, 2, 3]
del eggs[2]      # delete old list items
del eggs[1]
del eggs[0]
eggs.append(4)   # append new list items
eggs.append(5)
eggs.append(6)
print(eggs)

[4, 5, 6]
```

- This is what really happens:
- Changing a value of a mutable data type changes the value *in place*, since the variable's value is not replaced with a new list value.
- Calling functions with mutable vs. immutable arguments exhibits very different behavior.

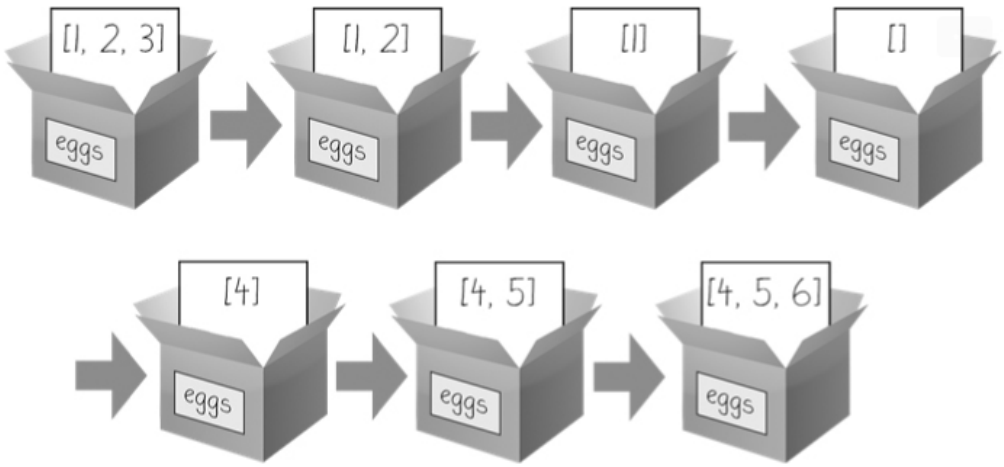


Figure 3: The original list is being altered

4 The `tuple` data type

- The `tuple` data type is very similar to `list` except:
  1. tuples are created with `()` while lists are created with `[]`
  2. tuples, like strings, are immutable, lists are mutable.
- Creating, indexing, slicing, measuring a tuple:

```
eggs = ('hello', 42, 0.5)
print(eggs[0])
print(eggs[1:3])
print(len(eggs))

hello
(42, 0.5)
3
```

- So far so good - but you cannot assign values to tuple items:

```
eggs = ('hello', 42, 0.5)
eggs[1] = 99
```

- So replacing an item works similarly as with strings: you have to rebuild the tuple.

```
eggs = ('hello', 42, 0.5)
eggs = (eggs[0], 99, eggs[2])
print(eggs)

('hello', 99, 0.5)
```

- Item deletion with `del` does not work, neither is there an `append` method for tuples: try that yourself!

```
eggs = ('hello', 42, 0.5)
del eggs[0]
eggs.append(1)
```

- It is OK to have trailing commas in list or tuple assignments:

```
eggs = [1,2,]
print(eggs)

bacon = (1,2,)
print(bacon)

[1, 2]
(1, 2)
```

- If you only have one value in your tuple, the trailing comma tells Python that this is a tuple and not a scalar or string:

```
print(type(1))
print(type((1,)))

print(type('hello'))
print(type(('hello',)))

<class 'int'>
<class 'tuple'>
<class 'str'>
<class 'tuple'>
```

- Why use tuples?
  1. to create ordered sequences that you don't want to change
  2. to gain speed when using ordered sequences for loops
- The functions `list` and `tuple` will convert their arguments to lists and tuples just like `str`.
- Converting a `list` into a `tuple`:

```
t = tuple(['cat','dog',5])
print(t)
print(type(t))

('cat', 'dog', 5)
<class 'tuple'>
```

- Converting a string into a `list` or a `tuple`:

```
l = list('hello')
print(l)
print(type(l))

t = tuple('hello')
print(t)
print(type(t))

['h', 'e', 'l', 'l', 'o']
<class 'list'>
('h', 'e', 'l', 'l', 'o')
<class 'tuple'>
```

- Converting a `tuple` into a `list` (handy if you want it to be mutable):

```
l = list(('cat','dog', 5))
print(l)
l.append([1,2,])
print(l)

['cat', 'dog', 5]
['cat', 'dog', 5, [1, 2]]
```

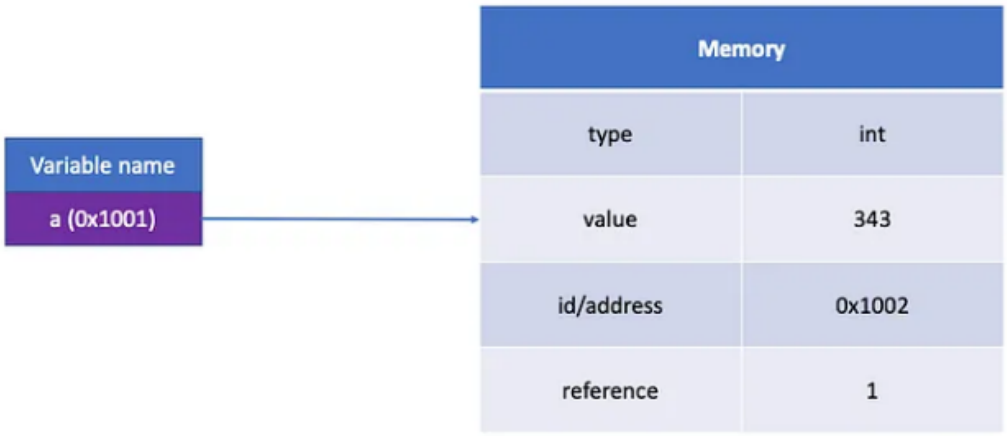


Figure 4: Python objects consist of identity, type, and value

5 References for integer variables

- Variables don't actually "store" strings or integer values. Technically, they store *references* to computer memory locations where the values are stored.
- In other languages like C/C++, you can manipulate values and these references (called *pointers*) separately, in Python you cannot.
- In this example, copying `spam` to `cheese` by assigning it to `cheese` does not copy the *value* over but only the *reference* to the value.

```
spam = 42
cheese = spam  # both point to memory with '42' in it
spam = 100     # spam points to memory with '100' in it
print(spam)
print(cheese)

100
42
```

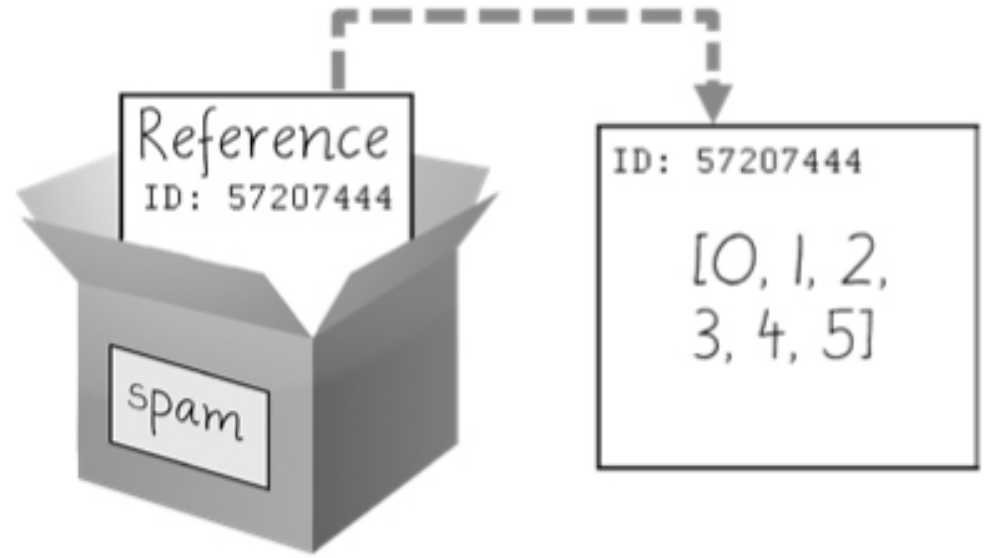
- Integers are *immutable* values that don't change: changing a variable only makes it refer to another value in memory.

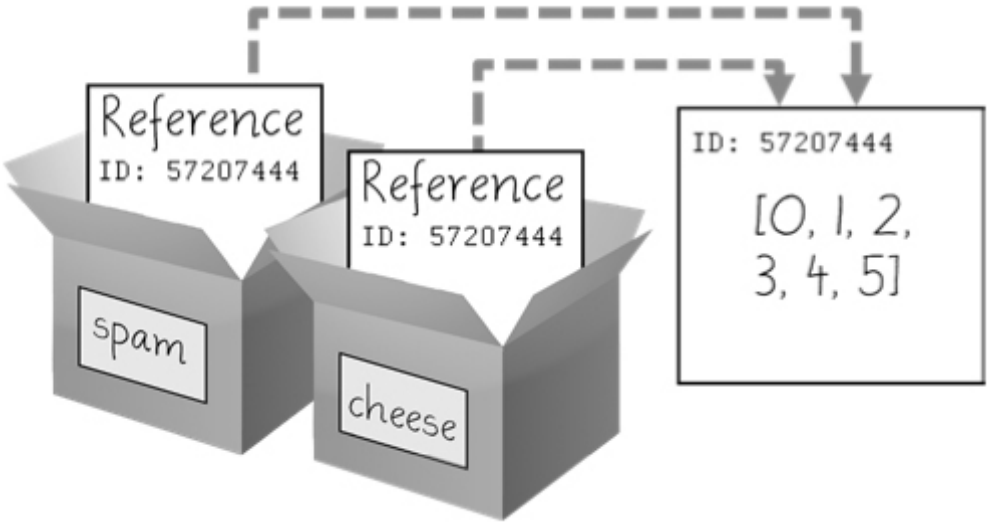
6 References for lists

- Lists don't work this way: they're *mutable* - their values can change.
- In this example, you see both referencing and value changing:

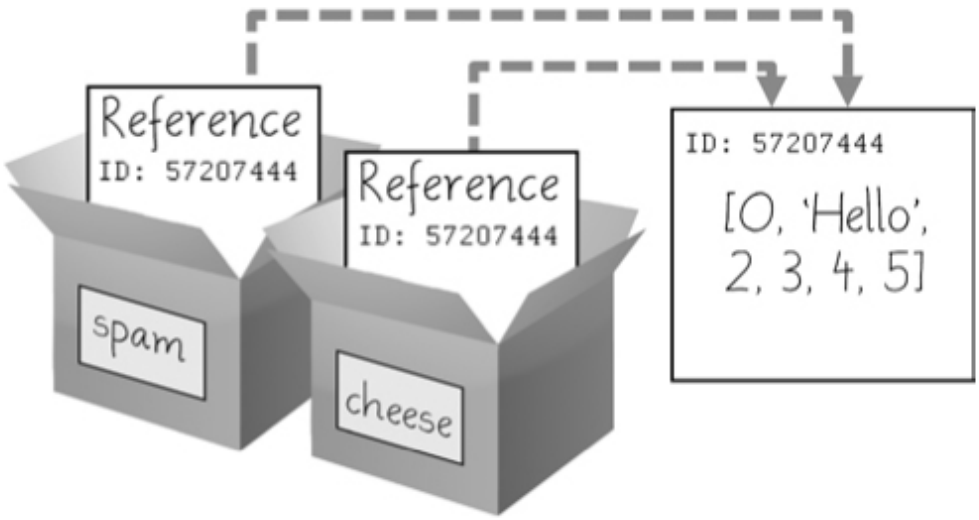
```
spam = [0,1,2,3,4,5]  #1

# Copy the reference to the list spam to a new list value
cheese = spam         #2
print(cheese)

# Change the list value in place
cheese[1] = 'hello'   #3

# Both variables refer to the same list
print(spam)
print(cheese)

[0, 1, 2, 3, 4, 5]
[0, 'hello', 2, 3, 4, 5]
[0, 'hello', 2, 3, 4, 5]
```

- What's happening here?
  1. A reference to the list is assigned to `spam`
  2. The list reference is copied to `cheese` (not the list itself)
  3. When the first element of `cheese` is modified, the same list that `spam` refers to is modified.
- The figures illustrate creation of `spam` and copying of the reference to `cheese`:





- When `cheese` is altered *in place*, the list that `spam` refers to is altered:



7 Identity and the `id` function

- All Python values have a unique ID that can be obtained with the `id` function:

```
print(id('Howdy'))
```

```
2149879939120
```

- If you keep running this chunk, the ID will change: Python picks this address based on where memory happens to be free at the time.
- `'Howdy'` is an immutable string and cannot be changed. To change the string in a variable, you have to make a new string object in a different place in memory to which the variable will refer/point.
- You can follow this process with `id`:

```
bacon = 'Hello'
print(id(bacon))

# Make a new string and refer to it as `bacon`
bacon += ' world!'
print(id(bacon))
```

```
2490280328880
2490280329136
```

- Lists are mutable objects and can be modified: `list.append` changes an existing list object (*in place*):

```
eggs = ['cat', 'dog']
print(id(eggs))

# append 'moose' to `eggs` modifies the list itself
eggs.append('moose')
print(id(eggs))

# create a new list with the same variable (reference) name
eggs = ['bat', 'rat', 'cows']
print(id(eggs))

# what is this then?
mem = eggs.insert(1,'cat')
print(id(mem))
print(id(eggs))
```

```
3064138990208
3064138990208
3064140986368
140733411716296
3064140986368
```

- In the last example, what is `mem = eggs.insert(1,'cat')`?

```
eggs = ['bat', 'rat', 'cows']
print(id(eggs))
mem = eggs.insert(1,'cat')
print(id(mem))
print(mem)
```

```
2416769073792
140733411716296
None
```

`mem` is nothing to us: it's a location that the computer uses to update `eggs` in place, but in actual fact it is a non-value (`None`).

8 Passing references in function calls

- When a function is called, the values of the arguments are passed to the parameter variables.
- If the function parameter is a list, this means that a copy of the reference is used for the parameter:

```
# function appends string to argument list (print list ref ID)
def eggs(someParameter):
    print('Parameter ID: ', id(someParameter))
    someParameter.append('Hello')

# initialize list and print ID
spam = [1,2,3]
print('List ID:      ', id(spam))

# call function on list and print return value
eggs(spam)

# the list is modified directly in place
print(spam)
```

```
List ID:       2605592052352
Parameter ID:  2605592052352
[1, 2, 3, 'Hello']
```

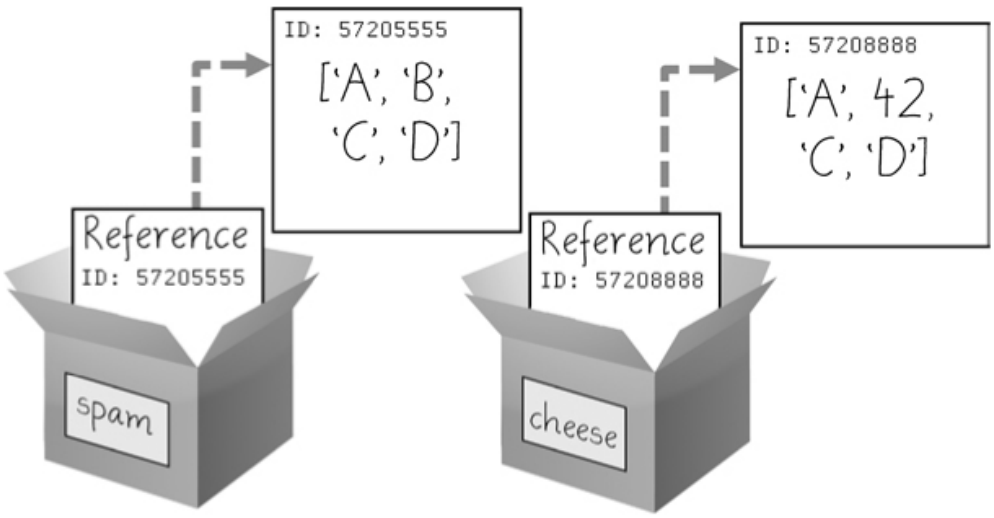
9 `copy` and `deepcopy` from the copy module

- If you do NOT want changes to apply to your original list, you can use the `copy.copy` and `copy.deepcopy` functions:

```
# import module from the standard library (no need to install)
import copy

# create a list and check its reference
spam = ['A', 'B', 'C', 'D']
print(id(spam))

# copy ‘spam‘ to ‘cheese‘ creating a new list
```

```
cheese = copy.copy(spam)
print(id(cheese))

# change list value
cheese[1] = 42

print(spam)
print(cheese)

1906525614592
1906525039296
['A', 'B', 'C', 'D']
['A', 42, 'C', 'D']
```

- `spam` and `cheese` variables now refer to separate lists: #caption: Different ID for different lists after copy.copy()



- If the list you need to copy contains lists, then use the `copy.deepcopy` function instead. Try this with `blt` below:

  1. create list `food = ['soup', ['bacon','lettuce','tomato']]`
  2. copy `food` to `sandwich`

3. change `'soup'` in `sandwich` to `'bread'`
4. print the IDs of `food` and `sandwich`
5. print the `food` and `sandwich` lists

- Solution:

```
import copy

# create list with list as item
food = ['soup', ['bacon','lettuce','tomato']]

# copy `food` to `sandwich`
sandwich = copy.deepcopy(food)

# change 'soup' in `sandwich` to 'bread'
sandwich[0] = 'bread'

# print the IDs of both lists
print('food: ', id(food))
print('copy: ', id(sandwich))

# check that list values are identical
print(food == sandwich)
print(food)
print(sandwich)

food:  1878210508224
copy:  1878210508032
False
['soup', ['bacon', 'lettuce', 'tomato']]
['bread', ['bacon', 'lettuce', 'tomato']]
```

10 Summary

- Tuples and strings, though also sequence data types, are immutable and cannot be changed.
- A variable that contains a tuple or string value can be overwritten with a new tuple or string value

- This is not the same thing as modifying the existing value in place — like, say, the `append()` or `remove()` methods do on lists.
- Variables do not store list values directly; they store references to lists. Any change you make to a list may impact other variables.
- You can use `copy()` or `deepcopy()` if you want to make changes to a list in one variable without modifying the original list.

11 Glossary

TERM/COMMAND	DEFINITION
`()`	Create tuple
`tuple`	Convert to tuple
`id`	Memory ID (address)
`copy.copy`	Copy list to new list
`copy.deepcopy`	Copy list that contains list to new list

12 References

- Sweigart, A. (2019). Automate the Boring Stuff with Python. NoStarch. URL: automatetheboringstuff.com