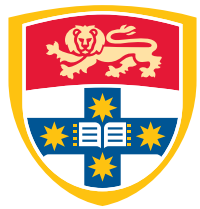
Introduction to Programming (Adv)

School of Computer Science, University of Sydney



Copyright Warning

Lecture 8: Reference semantics

Understanding references and data manipulation in memory

Assignment operator: primitive types

Consider the code with primitive types

```
1 x = 1 # int
2 y = x # int copy
3 x = 2 # int assignment
4 print(x)
5 print(y)
```

```
2
1
```

How can we represent this behaviour

Assignment operator: reference types

Consider the code with reference types

```
1 x = [ 8 ] # list of size 1, with value 8
2 y = x # list copy
3 x = [ 2 ] # list assignment
4 print(x[0])
5 print(y[0])
```

```
2
8
```

How can we represent this behaviour

Assignment operator: reference types (cont.)

Consider the code with reference types, modifying data inside a reference type

```
1 x = [ 8 ] # list of size 1, with value 8
2 y = x # list copy
3 x[0] = 2 # list assignment
4 print(x[0])
5 print(y[0])
```

```
2
2
```

What has happened? Again, how can we represent this behaviour

Reference semantics

Reference variable: a variable which refers to a region of memory which may be shared or common with another variable

Reference variables hold one value: the starting address of the region of memory

Think of primitives as a one to one mapping. A single binding of label (variable name) to one piece of data.
e.g. `x = 1`. Only `x` can change the data.

Think of reference types as a many to one mapping. There can be many bindings, labels, to one piece of data.
e.g. `x = y = [ 2 ]`. Both `x` and `y` can change the data.

Reference semantics (cont.)

The behaviour of the assignment operator is now dictated by the type

Wherever the `=` is seen, the LHS and RHS need to be considered

Equally important, wherever function calls occur, we need to consider the datatype between caller and callee of the function

The value held by the object is its memory address. This can be retrieved for any object in Python using the `id()` function.

Immutable vs mutable

We talk about primitive types and reference types. These are concepts built in to general programming.

In Python, everything is an object, so there are no real primitive types (even a float is wrapped in an object).

In other words, every variable is a reference type!

Question: So why does `x = 1; y = x; x = 2` not change the data?
Question: So why can't I change a string object `s = "abc"; s[0] = "k"` change the data?
Answer: Python language reference rules for data type: whether it is mutable or immutable will change the behaviour!

Immutable vs mutable (cont.)

"The value of some objects can change. Objects whose value can change are said to be mutable; objects whose value is unchangeable once they are created are called immutable. "[1]

Immutable: `bool`, `int`, `float`, `str`, `tuple`, ...

Mutable: `list`, `set`, `dictionary`, ...

Attempting to modify the data of an immutable object has implementation defined behaviour. In most cases it will result in a new object being returned.

[1] Section 3. Data model. Python3 Documentation

Python immutable and mutable

Datatype	operation	behaviour
`bool`	`=`	copy reference value (persistent refs)
`bool`	`+ - * / ...`	reference to int object footnote [2]
`int`	`=`	copy reference value
`int`	`+ - * / ...`	evaluate to new value [2]
`float`	`=`	copy reference value
`float`	`+ - * / ...`	ref. to new float object
`str`	`=`	copy reference value [3]
`str`	`+`	creates new str object
`str`	`format()`	return a new str object
`str`	`[ ]`	read only array access, otherwise exception

[2] 1) persistent ref. to int object [-5,256] or 2) new int object

[3] 1) persistent ref. to *compile time* strings if also compile time string, or 2) new str object

Python immutable and mutable

Datatype	operation	behaviour
`tuple`	`=`	copy reference value
`tuple`	`+`	creates new tuple object
`tuple`	`[ ]`	read only array access, otherwise exception
`list`	`=`	copy reference value
`list`	`+`	returns a new list object
`list`	`append()`	modifies list object
`list`	`[ ]`	read/write array access modifies list object

Python immutable and mutable (cont.)

The assignment operator on a mutable/immutable object will create a copy of the reference value (the memory address).

In Python, attempting to change an immutable datatype, will either:

- create an exception, or
- create a new object, and return the new reference value, or
- map to an existing object in memory (whatever compiler, runtime optimisations applied)

Question: Suppose `box` is a an immutable container object (tuple), containing multiple mutable containers (lists), each of which contain multiple immutable objects (int). What memory can be modified with the variable `box` without resulting in a new object being constructed?

Dereferencing

Reference types refer to a region of memory.

Dereferencing is an operation for visiting a specific region of memory to fetch data.

The dot notation is used to describe a dereference operation
e.g.
`object.data` or,
`object.method()`

The dot will instruct the program runtime library (Python interpreter) to visit the memory region and locate the data or method associated with that object

```
str.format()
sys.argv
sys.argv.append("more args?")
```

Dereferencing

Dereferencing fails when the address is invalid

Reference types that are not assigned to memory regions can take on the value `None`. It is symbolic to mean "no object".

Attempting to dereference `None` as another kind of object results in an exception

```
1 mystring = None
2 print(mystring)
3 print(type(mystring))
```

```
None
<class 'NoneType'>
```

Dereferencing (cont.)

```
1 mystring = None
2 print(mystring.isspace())
3 print(mystring.__len__())
4 print(len(mystring))
```

```
AttributeError: 'NoneType' object has no attribute 'isspace'
```

```
1 mystring = None
2 s = mystring + " is the name of the game"
```

```
TypeError: unsupported operand type(s) for +: 'NoneType' and 'str'
```

Other programming languages are more specific when referring to no object with null, or NULL. These are not types, but keywords which *cannot* be dereferenced, unlike Python `None` which is a `NoneType`.

Calling functions

Recall: parameters and arguments

parameters are assigned argument

`parameter = argument`

Reference semantics apply!

```
1  def foo1(x):
2      x = 1
3
4  x = 2
5  foo1(x)
6  print(x)
```

Calling functions (cont.)

```
1 def foo2(x):
2     x = [ 1 ]
3
4 x = [ 2 ]
5 foo2(x)
6 print(x)
```

```
1 def foo3(x):
2     x[0] = 1
3
4 x = [ 2 ]
5 foo3(x)
6 print(x)
```

Reference operators

is

operator.is_(a, b) - Return a is b. Tests object identity.[4]

==

The default behavior for equality comparison (== and !=) is based on the identity of the objects. [5]

Danger: `is` vs `==`

Well defined datatypes override behaviour of == to something meaningful to the datatype

Behaviour of `==`

Data type	effect
`int`	values are equal
`str`	entire string is equal
`list`	shallow copy is also `==`

[4] Section operator Expressions. Python3 Documentation

[5] Section 6. Expressions. Python3 Documentation

Summary

Assignment operations always copy reference values (Python: only objects)

Calling functions with arguments is equivalent to assignment operations

Depending on the datatype, operators will result in different behaviour for the object

Always know the datatype being used at any time in the code, and all associated behaviour.

Statically typed languages allow programmers to quickly identify what the effect of the operator will be.