COMP122/20 - Data Structures and Algorithms

03 Python Objects

Instructor : Ke Wei 〖柯韋〗

➽ A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/20/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

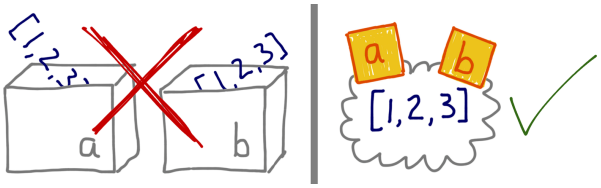
January 10, 2020

Outline

Textbook §2.1 – 2.3.

Objects and References

- Every value in Python is an object, including built-in type values, such as numbers and strings.
- We don't distinguish primitive types and class types. Types are all classes in Python.
- All variables store references to objects, they point to objects.
- The reference to an object is called the *identity* of the object.
- Another analogy is very clever — variables can also be treated as labels sticking on objects.



- An assignment `a = b` copies only the reference from `b` to `a`.
- There is a reference pointing to nothing — the `None` reference.

Comparing Values and Identities

- To compare if two values (objects) are equal or *not* equal, we use (==) or (!=). This is the content equality test.
- To compare if two references are the same identity or not the same, we use (`is`) or (`is not`).

```
>>> x = 11111111111111111111111111111
>>> y = 11111111111111111111111111111
>>> x is y
False
>>> x == y
True
```

- Comparisons using (`is`) are much quicker than those using (==).
- The comparison *a* `==` *b* calls the special method *a*.*__eq__*(*b*) in the background.
- We should always compare with `None` using (`is`).

Constructors and Equality Tests

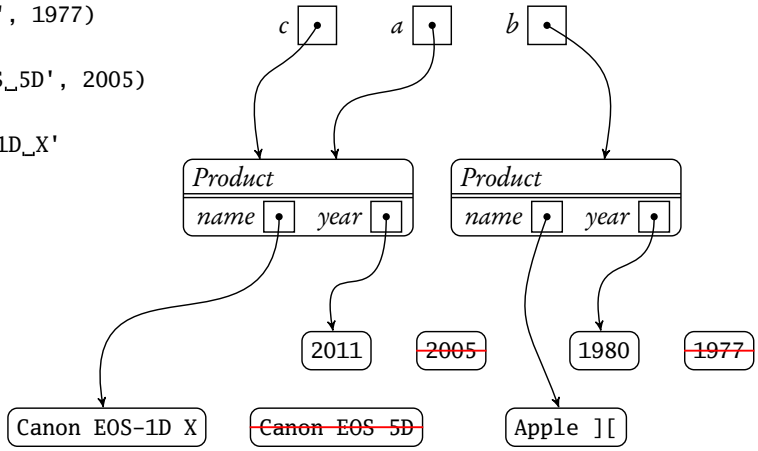
- Suppose we have a class *Product* containing two attributes — *name* and *year*. We define the constructor, equality test and hash function for the class.

```
class Product:
    def __init__(self, name, year):
        self.name, self.year = name, year
    def __eq__(self, other):
        return self is other or (self.name, self.year) == (other.name, other.year)
    def __hash__(self):
        return hash((self.name, self.year))
```

- Now we trace the object creations and assignments to illustrate the Python memory model for objects.

Tracing Object Creations and Assignments

```
a = Product('Apple ][', 1977)
b = a
c = Product('Canon EOS 5D', 2005)
a = c
a.name = 'Canon EOS-1D X'
b.year = 1980
c.year = 2011
```

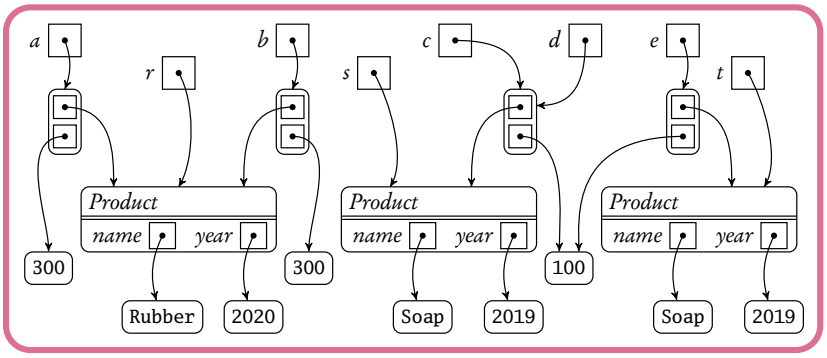


Propagated Deep Equality Tests

If all the classes implements the content equality test properly, deep equality tests can be achieved through propagation.

```
r, s, t = Product('Rubber', 2020), Product('Soap', 2019), Product('Soap', 2019)
a = (r, 300)
b = (r, 300)
c = (s, 100)
d = c
e = (t, 100)
```

`a == b`	`True`
`a is b`	`False`
`a == c`	`False`
`c is d`	`True`
`d == e`	`True`
`d is e`	`False`



Immutable and Mutable Objects

- Immutable objects are those cannot be changed (mutated) in-place.
- Any change to an immutable object creates a new object to reflect the change.
- References to immutable objects can be regarded as values.

```
>>> x = 100
>>> y = x
>>> y is x
True
```

```
>>> x += 1
>>> y is x
False
```

- Mutable objects can be changed in-place.
- Two references pointing to the same object create aliasing. Changing one of them also changes the other.

```
>>> s = [0,1,2,3]
>>> t = s
>>> s[2:2] = [-5,-6]
```

```
>>> t is s
True
```

```
>>> t
[0, 1, -5, -6, 2, 3]
```


Integral Types

- Python provides two built-in integral types, `int` and `bool`.
- Both integers and booleans are immutable.
- When used in boolean expressions, 0 and `False` are `False`, and any other integer and `True` are `True`.

```
>>> a = 0
>>> 'Non-zero' if a else 'Zero'
'Zero'
```

```
>>> 1 + (a < 100)
2
```

- When used in numerical expressions `True` evaluates to 1 and `False` to 0.
- The size of an integer is limited only by the machine's memory, so integers of hundreds of digits long can easily be created and worked with.
- The `a // b` integer division returns the floor $\lfloor \frac{a}{b} \rfloor$.
- We also have `a == (a // b) * b + (a % b)`.

Boolean Operations

- There are two built-in boolean objects: `True` and `False`.
- A boolean expression consists of three operations — `and`, `or` and `not`.
- Just like integers, all objects can be regarded as a boolean value in a boolean expression.
- By common sense, empty and nothing are regarded as `False`, others are `True`.

```
>>> bool([])        >>> bool('')        >>> bool(None)        >>> bool('ABCD')
False               False               False                True
```

- The `not` operation returns a `True` or `False`. However, the types of the results of `and` and `or` depends on the operands, and they use short-circuit evaluation.

```
>>> [] and 123          >>> [] or 'ABCD'          >>> '' or []
[]                      'ABCD'                    []
```

- This can be convenient and tricky — `def` *cat*(*a*,*b*)`:` `return` *a* `and` *b* `and` *a*+*b* `or` *a* `or` *b*.

Floating-Point Types

- Python provides three kinds of floating-point values: the built-in `float` and `complex` types, and the *decimal*.*Decimal* type from the standard library. All three are immutable.
- Type `float` holds double-precision floating-point numbers, they have limited precision and cannot reliably be compared for equality.
- Numbers of type `float` are written with a decimal point, or using exponential notation, for example, `0.0`, `4.`, `5.7`, `-2.5`, `-2e9`, `8.9e-4`.
- Floating-point numbers can be converted to integers using the `int()` function which returns the whole part and throws away the fractional part,
- or using `round()` which accounts for the fractional part, or using *math*.*floor*`()` or *math*.*ceil*`()` which convert down to or up to the nearest integer.
- Integers can be converted to floating point numbers using `float()`.

Defining Classes — Attributes and Methods

- Let's start with a very simple class, *Vec*, that holds a 2D vector.

```
class Vec:
    def __init__(self, x = 0, y = 0):
        self.x, self.y = x, y
    def dot(self, other):
        return self.x*other.x+self.y*other.y
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
    def __repr__(self):
        return "Vec({0.x!r}, {0.y!r})".format(self)
    def __str__(self):
        return "({0.x!s}, {0.y!s})".format(self)
```

- Attributes are declared in the methods, qualified by *self*, and *self* must be the first parameter of a method.

Reimplementing Special Methods

- Python calls special methods on an object to perform common actions, such as to initialize a new instance of a class.
- Reimplementing special methods in a user-defined class makes the class behaving like a built-in class.
- When an object is created, first the special method *__new__()* is called to create the object, and then the special method *__init__()* is called to initialize it. Only the *__init__()* method needs to be reimplemented to initialize the attributes.
- To support (==) on user-defined objects, We can reimplement the *__eq__()* special method and, better, the *__hash__()* special method as well, just like we override *equals* in Java.
- The built-in `repr()` function calls the *__repr__()* special method for the object it is given and returns the result. This should the string representation of the internal structure of the object.
- The built-in `str()` function works like the `repr()` function, except that it calls the object's *__str__()* special method. This should return a prettier string for human beings to read.

Overriding Operations and Operators

- Standard conversions and operations on objects also call special methods, such as `bool()` and math plus (+).
- We can reimplement these special methods to define the corresponding operations.

```
class Vec: ...
    def __abs__(self):
        return self.dot(self)**0.5
    def __bool__(self):
        return bool(abs(self))
    def __add__(self, other):
        return Vec(self.x+other.x, self.y+other.y)
    def __sub__(self, other):
        return Vec(self.x-other.x, self.y-other.y)
    def __mul__(self, scalar):
        return Vec(self.x*scalar, self.y*scalar)
```

- We can now use *Vec* as if it is a built-in class, supporting some operators.

Using Vectors

- Given three points $P=(1,2)$, $A=(2,5)$ and $B=(-1,7)$, compute the area of $\triangle APB$. Let $\vec{a}=\overrightarrow{PA}$ and $\vec{b}=\overrightarrow{PB}$. We compute the area by $\frac{\sqrt{(\vec{a}\cdot\vec{a})(\vec{b}\cdot\vec{b})-(\vec{a}\cdot\vec{b})^2}}{2}$.

```
>>> P = Vec(1,2)
>>> A = Vec(2,5)
>>> B = Vec(-1,7)
```

```
>>> a, b = A-P, B-P
>>> (a.dot(a)*b.dot(b)
      -a.dot(b)**2)**0.5/2
5.5
```

```
>>> a
Vec(1, 3)
>>> print(b)
(-2, 5)
```

- Given two points $Q=(3,5)$ and $K=(10,7)$, compute the distance from Q to K. We compute the length of vector $\overrightarrow{KQ}$ by $||Q-K||$.

```
>>> Q = Vec(3,5)
>>> K = Vec(10,7)
```

```
>>> abs(Q-K)
7.280109889280518
```