Advanced Python

Advanced Python

Eric Roberts
CSCI 121
December 3, 2018

Cool Features in Python

1. List and set comprehensions
2. Iterators
3. Generators
4. Decorators
   - `@staticmethod`
   - `@property`
   - `@memoize`

Set and List Comprehensions

- Mathematicians often define sets—just as Python does—by listing the elements inside curly braces, as in

  $E = \{ 0, 2, 4, 6, 8 \}$

  It is also common practice to define sets using a rule, as in

  $E = \{ x : x \text{ is an even integer and } 0 \le x \le 9 \}$

- Python supports a similar syntax called a ***set comprehension*** that uses the keywords `for` and `if` like this:

```
evenDigits = { x for x in range(10)
               if x % 2 == 0 }
```

- Python also defines ***list comprehensions*** in much the same way in which the braces are replaced by square brackets.

The Euler Totient Function

- Modern cryptography uses mathematics that spans more than two millennia.
- One important result was established by the Swiss mathematician Euler (1707–1783). The Euler totient function $\phi(n)$ is defined to be the number of positive integers less than n that are relatively prime to n. For example, $\phi(18)$ is 6:

- Comprehensions make the totient function easy to implement:

```
def totient(n):
    return len({ i for i in range(1, n)
                 if gcd(i, n) == 1 })
```

Exercise: Set and List Comprehensions

- What are the values of the following Python expressions:
  (a) `{ d for d in range(10) }`
  (b) `{ chr(ord("a") + i) for i in range(26) }`
  (c) `[ x ** 2 for x in range(10) ]`
  (d) `[ [ 0 ] * 3 for i in range(3) ]`
- How would you express these values using comprehension:
  (a) `{ "0", "1", "2", "3", "4", "5", "6", "7" }`
  (b) `[ 1, 10, 100, 1000, 10000, 100000, 1000000}`
  (c) `[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]`

Defining Iterators

- The `for` loop in Python allows the object following the `in` keyword to be any ***iterable object***, which must implement the following methods:
  - The `__iter__` method initializes any data structures required for the iterator and then returns the object.
  - The `__next__` method returns the next element in the iterator sequence. When the iterator is finished, the `__next__` method raises the `StopIteration` exception.
- The code on the next slide implements a `CharacterIterator` class that returns characters one at a time from the specified inclusive range. For example,

```
for ch in CharacterIterator("A", "Z"):
```

  iterates through the uppercase letters.

The `CharacterIterator` Class

```
# File: CharacterIterator.py

"""Implements an iterator for an inclusive character range."""

class CharacterIterator:

    def __init__(self, min, max):
        self._min = ord(min)
        self._max = ord(max)

    def __iter__(self):
        self._index = self._min
        return self

    def __next__(self):
        if self._index > self._max:
            raise StopIteration
        value = chr(self._index)
        self._index += 1
        return value
```

Generators

- Python also supports a programming model called a ***generator*** in which successive values can be returned in the course of a computation.
- Generators in Python look similar to functions except that they use the keyword `yield` in place of `return`. The difference between these statements is that `yield` stores the current computation state and starts over from there on the next call.
- This idea is illustrated by the following function, which generates successive Fibonacci numbers:

```
def FibonacciGenerator():
    t0, t1 = 0, 1
    while True:
        yield t0
        t2 = t0 + t1
        t0, t1 = t1, t2
```

Decorators

- One of Python's most interesting features is the ***decorator***, which is a function that is applied to another function that modifies it in some way.
- Decorators are applied by writing an `@` and the name of the decorator before a function definition.
- You have already seen the built-in decorator `staticmethod`, which is used to label a method that belongs to a class rather than to an instance of a class.

```
@staticmethod
def createChain(names):
    towers = None
    for name in reversed(names):
        towers = SignalTower(name, towers)
    return towers
```

The `@property` Decorator

- Another built-in decorator is `@property`, which provides what is now recognized as the "Pythonic" approach to traditional getters and setters.
- The code on the next slide reimplements to `GPoint` class so that the field names `x` and `y` provide immutable access to these components of the class, as shown in the following example:

```
Python 3.7.0 Shell
>>> from GPoint import GPoint
>>> pt = GPoint(3, 4)
>>> pt.x
3
>>> pt.x = 17
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    pt.x = 17
AttributeError: can't set attribute
>>>
```

The `@property` Decorator

```
# File: GPoint.py

class GPoint:

    def __init__(self, x=0.0, y=0.0):
        self._x = x
        self._y = y

    @property
    def x(self):
        return self._x

    @property
    def y(self):
        return self._y
```

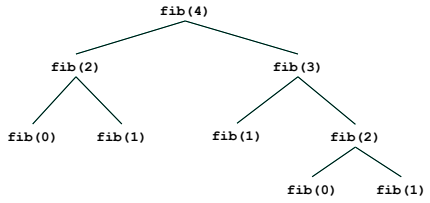
The Fibonacci Function

- In 1202, the Italian mathematician Leonardo Fibonacci proposed the following problem: How many pairs of rabbits exist after *n* months if one pair is introduced in month 0 and the rabbits reproduce according to the following rules:
  - Each pair of rabbits produces a new pair each month.
  - Rabbits become fertile in the second month of life.
  - Old rabbits never die.
- The Fibonacci function is easy to implement recursively:

```
def fib(n):
    if n < 2:
        return n
    else
        return fib(n - 2) + fib(n - 1)
```

Efficiency of the Recursive Solution

- Unfortunately, the recursive solution on the preceding slide runs in exponential time because the computation recalculates the same call many times:



- The calculation runs much faster if you keep track of previous results in a ***cache***. This strategy is called ***memoization***.

The `@memoize` Decorator

```
# File: MemoizedFib.py

def memoize(fn):
    cache = { }
    def memoizedFunction(x):
        if x not in cache:
            cache[x] = fn(x)
        return cache[x]
    return memoizedFunction

@memoize
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```