Ch08-2-Lists-Advanced

August 7, 2020

1 8.2 List Comprehensions & Higher order functions

1.1 Topics

- list shortcuts
- lambda functions applications
- built-in higher order functions

1.2 8.2.1 List comprehension

- list is a very powerful and commonly used container
- list shortcuts can make you an efficient programmer
- E.g., an arithmetic set $S = \{x^2 : x \in \{0...9\}\}$
  - is equivalent to:

  `S = [x**2 for x in range(10)]`
- consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses
  - the expressions can be anything
  - always results a new list from evaluating expression
- syntax:

```
somelist = [expression for item in list if conditional]
```

[2]:
```
# Typical way to create a list of squared values of list 0 to 9?
sq = []
for i in range(10):
    sq.append(i**2)
```

[3]:
```
print(sq)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

[4]:
```
# List comprehension -- handy technique:
S = [x**2 for x in range(10)]
```

[5]:
```
S
```

[5]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In maths: V = (1, 2, 4, 8, ... 2 12)

[6]:
```
# In python ?:
V = [2**x for x in range(13)]
print(V)
```

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]
```

In mathematics: $M = \{x|x \in S\ and\ x\ even\}$

[10]:
```
# Simple approach
M = []
for x in S:
    if x%2 == 0:
        M.append(x)
```

[11]:
```
print(M)
```

```
[0, 4, 16, 36, 64]
```

[12]:
```
# List comprehension
M1 = [x for x in S if x%2==0]
```

[14]:
```
assert M == M1, 'M and M1 are not equal!'
```

[12]:
```
#sentence = "The quick brown fox jumps over the lazy dog"
#words = sentence.split()
# can make a list of tuples or list of lists
wlist = [(w.upper(), w.lower(), len(w)) for w in "The quick brown fox jumps␣
 ↪over the lazy dog".split()]
```

[13]:
```
wlist
```

[13]:
```
[('THE', 'the', 3),
 ('QUICK', 'quick', 5),
 ('BROWN', 'brown', 5),
 ('FOX', 'fox', 3),
 ('JUMPS', 'jumps', 5),
 ('OVER', 'over', 4),
 ('THE', 'the', 3),
 ('LAZY', 'lazy', 4),
 ('DOG', 'dog', 3)]
```

1.3 8.2.2 higher order functions and lambda applications

- map, reduce, filter, sorted functions take function and iterable such as list as arguments
- lambda expression can be used as a parameter for higher order functions

1.3.1 sorted( )

[6]:
```
list1 = ['Apple', 'apple', 'ball', 'Ball', 'cat']
list2 = sorted(list1, key=lambda x: x.lower())
```

[7]:
```
print(list2)
```

```
['Apple', 'apple', 'ball', 'Ball', 'cat']
```

[8]:
```
list3 = [('cat', 10), ('ball', 20), ('apple', 3)]
from operator import itemgetter
list5 = sorted(list3, key=itemgetter(1), reverse=True)
```

[9]:
```
print(list5)
```

```
[('ball', 20), ('cat', 10), ('apple', 3)]
```

[10]:
```
list6 = sorted(list3, key=lambda x: x[1], reverse=True)
```

[11]:
```
print(list6)
```

```
[('ball', 20), ('cat', 10), ('apple', 3)]
```

1.3.2 filter( )

- filter elemets in the list by returning a new list for each element the function returns True

[13]:
```
help(filter)
```

```
Help on class filter in module builtins:

class filter(object)
 |  filter(function or None, iterable) --> filter object
 |
 |  Return an iterator yielding those items of iterable for which function(item)
 |  is true. If function is None, return the items that are true.
 |
 |  Methods defined here:
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __next__(self, /)
 |      Implement next(self).
 |
 |  __reduce__(…)
 |      Return state information for pickling.
```

```
[14]: list7 = [2, 18, 9, 22, 17, 24, 8, 12, 27]
      list8 = list(filter(lambda x: x%3==0, list7))
```

```
[15]: print(list8)
```

```
[18, 9, 24, 12, 27]
```

1.3.3 map( )

```
[16]: help(map)
```

```
Help on class map in module builtins:

class map(object)
 |  map(func, *iterables) --> map object
 |
 |  Make an iterator that computes the function using arguments from
 |  each of the iterables.  Stops when the shortest iterable is exhausted.
 |
 |  Methods defined here:
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __next__(self, /)
 |      Implement next(self).
 |
 |  __reduce__(…)
 |      Return state information for pickling.
```

```
[17]: items = list(range(1, 11))
      squared = list(map(lambda x: x**2, items))
```

```
[18]: print(squared)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

[1]:
```
# map each words with its length
sentence = "The quick brown fox jumps over the lazy dog"
words = [word.lower() for word in sentence.split()]
```

[2]:
```
print(words)
```

```
['the', 'quick', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
```

[21]:
```
w_len = list(map(lambda w: (w, w.upper(), len(w)), words))
```

[22]:
```
print(w_len)
```

```
[('the', 'THE', 3), ('quick', 'QUICK', 5), ('fox', 'FOX', 3), ('jumps', 'JUMPS',
5), ('over', 'OVER', 4), ('the', 'THE', 3), ('lazy', 'LAZY', 4), ('dog', 'DOG',
3)]
```

1.3.4 reduce( )

- reduce() is found in functools module
- used to reduce a list of values to a single output

[23]:
```
import functools
help(functools)
```

```
Help on module functools:

NAME
    functools - functools.py - Tools for working with functions and callable
objects

MODULE REFERENCE
    https://docs.python.org/3.6/library/functools

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

CLASSES
    builtins.object
        partial
        partialmethod

    class partial(builtins.object)
     |  partial(func, *args, **keywords) - new function with partial application
     |  of the given arguments and keywords.
     |
     |  Methods defined here:
     |
     |  __call__(self, /, *args, **kwargs)
     |      Call self as a function.
     |
     |  __delattr__(self, name, /)
     |      Implement delattr(self, name).
     |
     |  __getattribute__(self, name, /)
     |      Return getattr(self, name).
     |
     |  __new__(*args, **kwargs) from builtins.type
     |      Create and return a new object.  See help(type) for accurate
signature.
     |
     |  __reduce__(…)
     |      helper for pickle
     |
     |  __repr__(self, /)
     |      Return repr(self).
     |
     |  __setattr__(self, name, value, /)
     |      Implement setattr(self, name, value).
     |
     |  __setstate__(…)
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |
     |  args
     |      tuple of arguments to future partial calls
     |
     |  func
     |      function object to use in future partial calls
     |
     |  keywords
     |      dictionary of keyword arguments to future partial calls

    class partialmethod(builtins.object)
     |  Method descriptor with partial application of the given arguments
     |  and keywords.
     |
     |  Supports wrapping existing descriptors and handles non-descriptor
     |  callables as instance methods.
     |
     |  Methods defined here:
     |
     |  __get__(self, obj, cls)
     |
     |  __init__(self, func, *args, **keywords)
     |      Initialize self.  See help(type(self)) for accurate signature.
     |
     |  __repr__(self)
     |      Return repr(self).
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __isabstractmethod__
     |
     |  __weakref__
     |      list of weak references to the object (if defined)

FUNCTIONS
    cmp_to_key(…)
        Convert a cmp= function into a key= function.

    lru_cache(maxsize=128, typed=False)
        Least-recently-used cache decorator.

        If *maxsize* is set to None, the LRU features are disabled and the cache
        can grow without bound.

        If *typed* is True, arguments of different types will be cached
separately.
        For example, f(3.0) and f(3) will be treated as distinct calls with
        distinct results.

        Arguments to the cached function must be hashable.

        View the cache statistics named tuple (hits, misses, maxsize, currsize)
        with f.cache_info().  Clear the cache and statistics with
f.cache_clear().
        Access the underlying function with f.__wrapped__.

        See:  http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used

    reduce(…)
        reduce(function, sequence[, initial]) -> value
        Apply a function of two arguments cumulatively to the items of a
sequence,
        from left to right, so as to reduce the sequence to a single value.
        For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates
        ((((1+2)+3)+4)+5).  If initial is present, it is placed before the items
        of the sequence in the calculation, and serves as a default when the
        sequence is empty.

    singledispatch(func)
        Single-dispatch generic function decorator.

        Transforms a function into a generic function, which can have different
        behaviours depending upon the type of its first argument. The decorated
        function acts as the default implementation, and additional
        implementations can be registered using the register() attribute of the
        generic function.

    total_ordering(cls)
        Class decorator that fills in missing ordering methods

    update_wrapper(wrapper, wrapped, assigned=('__module__', '__name__',
'__qualname__', '__doc__', '__annotations__'), updated=('__dict__',))
        Update a wrapper function to look like the wrapped function

        wrapper is the function to be updated
        wrapped is the original function
        assigned is a tuple naming the attributes assigned directly
        from the wrapped function to the wrapper function (defaults to
        functools.WRAPPER_ASSIGNMENTS)
        updated is a tuple naming the attributes of the wrapper that
        are updated with the corresponding attribute from the wrapped
        function (defaults to functools.WRAPPER_UPDATES)

    wraps(wrapped, assigned=('__module__', '__name__', '__qualname__',
'__doc__', '__annotations__'), updated=('__dict__',))
        Decorator factory to apply update_wrapper() to a wrapper function

        Returns a decorator that invokes update_wrapper() with the decorated
        function as the wrapper argument and the arguments to wraps() as the
        remaining arguments. Default arguments are as for update_wrapper().
        This is a convenience function to simplify applying partial() to
        update_wrapper().

DATA
    WRAPPER_ASSIGNMENTS = ('__module__', '__name__', '__qualname__', '__do…
    WRAPPER_UPDATES = ('__dict__',)
    __all__ = ['update_wrapper', 'wraps', 'WRAPPER_ASSIGNMENTS', 'WRAPPER_…
```

```
FILE
    /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/functools.py
```

1.4 8.2.3 higher order function applications

1.4.1 find sum of first n values

[24]:
```
s = functools.reduce(lambda x,y:x+y, range(1, 11))
```

[25]:
```
assert sum(range(1, 11)) == s
```

1.4.2 find factorial (or product of) first n values

[26]:
```
fact = functools.reduce(lambda x,y:x*y, range(1, 11))
```

[27]:
```
fact
```

[27]:
```
3628800
```

[28]:
```
import math
assert math.factorial(10) == fact
```

[ ]:
```

```