Chapter 8 - Advanced Programming Techniques

CS 172 - Computer Programming 2
Lanzhou University

These slides use many elements provided in the main bibliographic reference for these lectures:

Programming in Python 3
A Complete Introduction to the Python Language,
2nd Edition,
Mark Summerfield

Outline

Outline

Recursive Functions

- Recursive functions (or methods) are ones that call themselves
- They can be seen as having two cases:
  - the *base case*, used to stop recursion, and
  - the *recursive case*
- Recursive functions can be computationally expensive
- But some algorithms are naturally expressed using recursion

An Example - Factorial

- The classic example of a recursive function is the `factorial` function
- If we want the value of the factorial of 5
  - this is calculated as: 5*4*3*2*1 which is 120
- Factorial can quite naturally be expressed in a recursive function

```
def factorial(x):
    if x <= 1:                  # base case
        return 1
    else:                       # recursive case
        r = factorial(x - 1)
        return x * r
```

- What about termination?

A More Complex Example

- Let's study a function – `indented_list_sort()` – that takes a list of strings that use indentation to create a hierarchy
- It also receives a string that holds one level of indent (e.g., 4 white spaces)
- It returns a list with the same strings
- But where all the strings in the same level are sorted in case-insensitive alphabetical order
- And with indented items sorted under their parent item, recursively
- See the example of the before and after lists in the next slide

Before and After

Before:

```
before = ["Nonmetals",
          "    Hydrogen",
          "    Carbon",
          "    Nitrogen",
          "    Oxygen",
          "Inner Transitionals",
          "    Lanthanides",
          "        Cerium",
          "        Europium",
          "    Actinides",
          "        Uranium",
          "        Curium",
          "        Plutonium",
          "Alkali Metals",
          "    Lithium",
          "    Sodium",
          "    Potassium"]
```

After

```
after = ["Alkali Metals",
         "    Lithium",
         "    Potassium",
         "    Sodium",
         "Inner Transitionals",
         "    Actinides",
         "        Curium",
         "        Plutonium",
         "        Uranium",
         "    Lanthanides",
         "        Cerium",
         "        Europium",
         "Nonmetals",
         "    Carbon",
         "    Hydrogen",
         "    Nitrogen",
         "    Oxygen"]
```

Solution

- The solution to this problem is provided in file `IndentedList.py`
  - which we are providing
- We will revise and study its `indented_list_sort()` function
  - it can be used, e.g., as
    `after = IndentedList.indented_list_sort(before)`
  - it has an optional `indent` argument, which by default is `"    "`

The Function indented_list_sort()

- We start by looking at `indented_list_sort()` as a whole

```
def indented_list_sort(indented_list, indent="    "):

    KEY, ITEM, CHILDREN = range(3)

    def add_entry(level, key, item, children):
        ...

    def update_indented_list(entry):
        ...

    entries = []
    for item in indented_list:
        level = 0
        i = 0
        while item.startswith(indent, i):
            i += len(indent)
            level += 1
        key = item.strip().lower()
        add_entry(level, key, item, entries)

    indented_list = []
    for entry in sorted(entries):
        update_indented_list(entry)
    return indented_list
```

- It begins by creating three constants used to provide names for index positions used by the local functions
- The function defines two *local* functions, that we will analyze later

The First Part of the Algorithm

- The sorting algorithm works in two stages:
  1. we create a list of entries, each a 3-tuple containing
     - a key, that will be used for sorting (the original string, lowercased and stripped of spaces)
     - the original string
     - a list of the string's child entries

     This is supported by function `add_entry()`
     - which is called for each string in the list

     ```
     def add_entry(level, key, item, children):
         if level == 0:
             children.append((key, item, []))
         else:
             add_entry(level - 1, key, item, children[-1][CHILDREN])
     ```

     `level` is the indentation level
     - 0 for top-level items, 1 for children of top-level items, and so on

     `children` is the list to which new entries are added
     - when called from `indented_list_sort()`, this is the entries list

The Recursive Function add_entry

```
def add_entry(level, key, item, children):
    if level == 0:
        children.append((key, item, []))
    else:
        add_entry(level - 1, key, item, children[-1][CHILDREN])
```

- If the level is 0 (top-level), we add a new 3-tuple to `entries` list with:
  - the key (for sorting)
  - the original item (which will go into the resultant sorted list), and
  - an empty children list
- This is the base case since no recursion takes place
- If the level is greater than 0, then we know the item is a child of the last item in the children list
- In this case we recursively call `add_entry()` again
  - We reduce the level by 1 and pass the children list's last item's children list as the list to add to
- If the level is 2 or more, more recursive calls will take place
- Until eventually the level is 0 and the children list is the right one for the entry to be added to

Example

- When the `"Inner Transitionals"` string is reached, the outer function calls `add_entry()` with a level of 0, a key of `"inner transitionals"`, an item of `"Inner Transitionals"`, and the `entries` list as the children list
- The `entries` list already contains

```
[('nonmetals',
  'Nonmetals',
  [('hydrogen', '    Hydrogen', []),
   ('carbon'  , '    Carbon'  , []),
   ('nitrogen', '    Nitrogen', []),
   ('oxygen'  , '    Oxygen'  , [])])]
```

- Since the level is 0, a new item will be appended to the children list (`entries`), with the key, item, and an empty children list

```
[('nonmetals',
  'Nonmetals',
  [('hydrogen', '    Hydrogen', []),
   ('carbon'  , '    Carbon'  , []),
   ('nitrogen', '    Nitrogen', []),
   ('oxygen'  , '    Oxygen'  , [])]),
 ('inner transitionals',
  'Inner Transitionals',
  [])]
```

Example (cont.)

- The next string is `"    Lanthanides"`
  - This is indented, so it is a child of the `"Inner Transitionals"` string
- The `add_entry()` call this time has a level of 1, a key of `"lanthanides"`, an item of `"    Lanthanides"`, and the entries list as the children list
- Since the level is 1, the `add_entry()` function calls itself recursively, this time with level 0 (1 - 1), the same key and item, but with the children list being the children list of the last item, that is, the `"Inner Transitionals"` item's children list (which is the empty list)
- When `add_entry()` executes this recursive call, since the level will be 0, it will add the tuple `(("lanthanides", "    Lanthanides", [])` to the list of children of `"Inner Transitionals"` as expected

Example of the Result of the Recursive Function add_entry

- For the before list, the resulting entries list, before sorting, is:

```
[('nonmetals',
  'Nonmetals',
  [('hydrogen', '    Hydrogen', []),
   ('carbon'  , '    Carbon'  , []),
   ('nitrogen', '    Nitrogen', []),
   ('oxygen'  , '    Oxygen'  , [])]),
 ('inner transitionals',
  'Inner Transitionals',
  [('lanthanides',
    '    Lanthanides',
    [('cerium'  , '        Cerium'  , []),
     ('europium', '        Europium', [])]),
   ('actinides',
    '    Actinides',
    [('uranium'  , '        Uranium'  , []),
     ('curium'   , '        Curium'   , []),
     ('plutonium', '        Plutonium', [])])]),
 ('alkali metals',
  'Alkali Metals',
  [('lithium'  , '    Lithium'  , []),
   ('sodium'   , '    Sodium'   , []),
   ('potassium', '    Potassium', [])])]
```

- The output was post-processed to make it easier to read (printed using the pprint() function of the module pprint)
- The list has 3 items, all of which are 3-tuples
  - and each 3-tuple's last element is a list of child 3-tuples, or []

More About add_entry

- Regarding `add_entry()`
  - It is a recursive function
    - it has a base case, when the level is 0
    - and a recursive case
  - It is also a **local** function
    - it provides helping functionality inside another function

The Second Part of the Algorithm

- The sorting algorithm works in two stages:
  2. In the second stage, we begin with a new empty indented list
     We iterate over the **sorted** entries
     - This is done by calling update_indented_list() for each one

```
def update_indented_list(entry):
    indented_list.append(entry[ITEM])
    for subentry in sorted(entry[CHILDREN]):
        update_indented_list(subentry)
```

     - This is a(nother) recursive function
     - For each top-level entry it adds an item to indented_list
     - It then calls itself for each of the item's child entries
     - Each child is added, and the function calls itself again
     - Until an item, or child, or child of a child, and so on, has no children of its own; this is the base case

     Note indented_list is not a variable of update_indented_list()
     When Python looks for indented_list in the local (inner function) scope, doesn't find it
     So, it then looks in the enclosing scope (function update_indented_list()) and finds it there

Outline

Functional-Style Programming

- Computations are built from combining functions, that
  - do not modify their arguments
  - do not refer to or change the program's state
  - provide their results as return values
- This style is convenient in many ways:
  - it is easier to develop functions in isolation
  - it is easier to debug functional programs
- Three concepts are strongly associated with functional programming
  - *mapping*
  - *filtering*
  - *reducing*

Mapping

- Takes a function and an iterable
- Produces a new iterable (or a list)
  - where each item is the result of calling the function on the corresponding item in the original iterable
- This is supported by the built-in `map()` function:

```
>>> list(map(lambda x: x**2, [1, 2, 3, 4]))
[1, 4, 9, 16]
```

- The `map` function takes a function and an iterable as arguments
  - and, for efficiency, returns an iterator rather than a list
- We can force a list as a result, using a list comprehension:

```
>>> [x ** 2 for x in [1, 2, 3, 4]]
[1, 4, 9, 16]
```

Filtering

- Takes a function and an iterable
- Produces a new iterable
  - where each item is from the original iterable
  - providing the function returns `True` when called on it
- This is supported by the built-in `filter()` function:

```
>>> list(filter (lambda x: x > 0, [1, -2, 3, -4]))
[1, 3]
```

- We can also use list comprehensions for filtering:

```
>>> [x for x in [1, -2, 3, -4] if x > 0]
[1, 3]
```

Reducing

- Takes a function and an iterable
- Produces a single result
  - the function is called on the iterable's first two values
  - then on the computed result and the third value
  - then on the computed result and the fourth value
  - and so on, until the values have all been used
- This is supported by the `functools.reduce()` function:

```
>>> functools.reduce(lambda x, y: x*y, [1, 2, 3, 4])
24
>>> functools.reduce(operator.mul, [1, 2, 3, 4])
24
```

- `functools.reduce(function, iterable[, initializer])` can receive an optional initializer
  - if the iterable is empty and there is an initializer, then it returns the initializer
  - if the iterable is empty and there is no initializer, then it throws an error
  - if the iterable has one element, that element is returned

Built-in Reducing Functions

- Python provides some built-in reducing functions:

```
>>> all([1, 'a', True]) # returns True if all items return True for bool()
True

>>> all([1, '', True])
False

>>> any([1, '', True]) # returns True if any item returns True for bool()
True

>>> max([1, 17, 4]) # returns the largest item
17

>>> min([1, 17, 4]) # returns the smallest item
1

>>> sum([1, 17, 4]) # returns the sum of all items
22
```

- `itertools.accumulate` returns yields all intermediate values

```
>>> list(itertools.accumulate([1, 2, 3, 4], lambda x, y: x * y))
[1, 2, 6, 24]
```

Further Examples

- To determine the combined size of the files in a list:

```
>>> functools.reduce(operator.add,
                     map(os.path.getsize, ["lecture.pdf", "lecture.tex"]))
206767
```

  - `operator.add` is equivalent to `lambda x, y: x + y`

- We can produce the same result using list comprehension (usually more verbose)

```
>>> functools.reduce(operator.add,
                     (os.path.getsize(x) for x in ["lecture.pdf",
                                                    "lecture.tex"]))
```

- The same as before, but filtering out non-pdf files:

```
>>> functools.reduce(operator.add,
                     map(os.path.getsize,
                         filter (lambda x: x.endswith(".pdf"),
                                 ["lecture.pdf", "lecture.tex"])))
184568
```

- Using `map()`, `filter()` and `functools.reduce()` often leads to the elimination of loops

Further Examples

- Using `map()`, `filter()` and `functools.reduce()` often leads to the elimination of loops
- Example: find the total size of PDF files in a list

```
total_size = 0
for file_ in ["lecture.pdf", "lecture.tex"]:
    if file_.endswith(".pdf"):
        total_size += os.path.getsize(file_)
```

- Can be written in many, equivalent, ways:

```
functools.reduce(operator.add,
                 (os.path.getsize(x)
                    for x in filter(lambda x: x.endswith(".pdf"),
                                    ["lecture.pdf", "lecture.tex"])))

functools.reduce(operator.add,
                 (os.path.getsize(x)
                    for x in (x for x in ["lecture.pdf",
                                          "lecture.tex"] if x.endswith(".pdf")
```

Further Examples

- Can be written in many, equivalent, ways (cont.):

```
functools.reduce(operator.add,
                 map(os.path.getsize,
                     (x for x in ["lecture.pdf", "lecture.tex"]
                                  if x.endswith(".pdf"))))

functools.reduce(operator.add,
                 map(os.path.getsize,
                     filter(lambda x: x.endswith(".pdf"),
                            ["lecture.pdf", "lecture.tex"])))

sum(map(os.path.getsize,
    filter(lambda x: x.endswith(".pdf"), ["lecture.pdf", "lecture.tex"])))

sum(os.path.getsize(x)
    for x in ["lecture.pdf", "lecture.tex"] if x.endswith(".pdf"))
```