Programmieren I (Python)

Christian Osendorfer

2023-10-19

Prime Numbers

Prime Numbers

```python
# Try out #%% in vscode:
# Put it on a single line at the top of your python file.
def smallest_factor(n):
    """Returns the smallest value k>1 that evenly divides N."""
    for factor in range(2, n+1):
        if n % factor == 0:
            return factor

def is_prime(n):
    """Return True iff n is prime."""
    return smallest_factor(n) == n

def print_factors(n):
    """Print the prime factors of n."""
    if n == 1:
      print(1)
      return

    while True:
      factor = smallest_factor(n)
      print(factor)
      n = n // factor
      if n == 1:
        return
```

Tuples

Tuples

- An ordered sequence of elements, can mix element types.
- Cannot change its element values: an **immutable** *object* (more on objects later!)
- Represented with parentheses ( ).
-
```
1   te = () # emtpy tuple
2   t = (1, "HAW LA", 2.3)
3   t[0]
4   t + (4, 1)
5   t[1:3]
6   len(t)
7   t[1] = "LA" # error
```

Tuples

- Used to **swap** variable values in a nice way:
  - `x, y = y, x`
- Used to **return more than one value** from a function.
- Can **iterate** over elements of a tuple:
  - ```
    for elem in t:
      ...
    ```
    ```

Lists

Lists

- A list is a container that holds a sequence of related pieces of information.
- Usually the elements are of the same type (but not required to be so!).
- A list is **mutable**.
- The shortest list is an empty list, just 2 square brackets:
  - ```
    1  members = []
    ```
    ```

Lists

Lists can hold any Python values, separated by commas:

```
members = ["Pamela", "Tinu", "Brenda", "Kaya"]
ages_of_kids = [1, 2, 7]
prices = [79.99, 49.99, 89.99]
digits = [2//2, 2+2+2+2, 2, 2*2*2]
remixed = ["Pamela", 7, 79.99, 2*2*2]
```

Lists – `len`

Use the **global** `len()` function to find the *length of a list*.

```
1 attendees = ["Tammy", "Shonda", "Tina"]
2 print(len(attendees))
3
4 num_of_attendees = len(attendees)
5 print(num_of_attendees)
```

🤔 What could go wrong with storing the length?

Strings are *lists of characters*. `len` also works for Strings.

Accessing list items

Each list item has an index, starting from 0.

```
1 letters = ['A', 'B', 'C']
2 # Index:    0    1    2
```

Access each item by putting the index in brackets:

```
1 letters[0]
2 letters[1]
3 letters[2]
4 letters[3]
5 curr_ind = 1
6 letters[curr_ind]
```

Negative indices are also possible:

```
1 letters[-1]
2 letters[-2]
```

Accessing list items

It’s also possible to use a function from the `operator` module:

```
1 from operator import getitem
2
3 getitem(letters, 0)
```

Assignment is possible! Lists are **mutable**.

```
1 letters = ['A', 'B', 'C']
2 letters[1] = 'P'
```

List concatenation

Add (i.e. connect/concatenate) two lists together using the + operator:

```
1 boba_prices = [5.50, 6.50, 7.50]
2 smoothie_prices = [7.00, 7.50]
3 all_prices = boba_prices + smoothie_prices
```

Or the add function:

```
1 from operator import add
2
3 boba_prices = [5.50, 6.50, 7.50]
4 smoothie_prices = [7.00, 7.50]
5 all_prices = add(boba_prices, smoothie_prices)
```

List repetition

Concatenate the same list multiple times using the `*` operator:

```
1 boba_prices = [5.50, 6.50, 7.50]
2 more_boba = boba_prices * 3
```

Or the `mul` function:

```
1 from operator import mul
2
3 boba_prices = [5.50, 6.50, 7.50]
4 more_boba = mul(boba_prices, 3)
```

Nested Lists

Since Python lists can contain any values, an item can itself be a list.

```
gymnasts = [
  ["Brittany", 9.15, 9.4, 9.3, 9.2],
  ["Lea", 9, 8.8, 9.1, 9.5],
  ["Maya", 9.2, 8.7, 9.2, 8.8]
]
```

- What is the length of `gymnasts`?
- What is the length of `gymnasts[0]`?

Accessing nested list items

```
gymnasts = [
  ["Brittany", 9.15, 9.4, 9.3, 9.2],
  ["Lea", 9, 8.8, 9.1, 9.5],
  ["Maya", 9.2, 8.7, 9.2, 8.8]
]
```

Access using bracket notation, with more brackets as needed:

```
gymnasts[0]
gymnasts[0][0]
gymnasts[1][0]
gymnasts[1][4]
gymnasts[1][5]
gymnasts[3][0]
```

Containment operator

Use the `in` operator to test if a value is inside a list:

```
1 digits = [2, 8, 3, 1, 8, 5, 3, 0, 7, 1]
2 1 in digits
3 3 in digits
4 4 in digits
5 not (4 in digits)
```

Iterating over lists

- Very common pattern, iterate over list elements:
-
```
1   L = [4, 2, 1, 5]
2   for i in range(len(L)):
3     # do something with L[i]
```
- Iterate *directly* over list elements (also see strings!):
-
```
1   L = [4, 2, 1, 5]
2   for elem in L:
3     # do something with elem
```

Add one element to a list

- We want to keep the list we are working with and **append** one element:
  - ```
    1  L = [2, 1, 3]
    2  L.append(4) # L is now [2, 1, 3, 4]
    ```
- The existing list is changed (*in place*)! A *side effect*!
- What is this *dot* in `L.append(4)`?
  - A list is a **python object** (much more on this later).
  - Objects have data and methods that act on the data.
  - Call a method of an object `obj` by `obj.method(...)`
    ```

Extend a list

- Extend a given list with the elements of another list.
  - ```
    1  A, B = [2, 1, 3], [10, 20, 30]
    2  A.extend(B) # A is now ?
    ```
    ```

Remove elements from a list

- Delete an element at a specific index: `del(L[index])`.
- Remove last element *and* return it: `x = L.pop()`
- Remove a given element: `L.remove(2)`
  - If element not in list, gives an error
  - If element occurs multiple times, removes first occurance.

Lists and Strings

- Convert a *string* `s` *to a list*: `list(s)`
- Use `s.split()` to split string `s` on provided character (or space if none is given)
- Use `''.join(L)` to turn a *list of* ***strings*** *into one string*. Character in quotes is added between every element.

Lists in memory

- A list is an object in memory.
- Variable names can point to objects in memory.
- Any variable pointing to the same object in memory is affected by changes to this object!
-

```
1  L = [1, 3, 2]
2  l = L # variable l is the same list as L. l is an `alias`
3  l.append(3)
4  print(L) # run this in python tutor
```

Lists in memory – slicing

- Use **list slicing** to create a *proper* copy of part or all of a list.
- `lst[<start index>:<end index>:<step size>]` evaluates to a new list containing elements of `lst`:
  - Starting at and including the element at `<start index>`.
  - Up to but not including the element at `<end index>`.
  - With `<step size>` as the difference between indices of elements to include.

```
1  lst = [6, 5, 4, 3, 2, 1, 0]
2  lst[:3]     # Start index defaults to 0
3  lst[3:]     # End index defaults to len(lst)
4  lst[:]      # Creates a copy of the list
5  lst[:] == lst
6  lst[:] is lst
7  lst[::-1]   # Make a reversed copy of the entire list
8  lst[::2]    # Skip every other; step size defaults to 1 otherwise
```

Lists in memory

- Do not change a list while iterating over it.

```
1 def remove_dups(L1, L2):
2   for e in L1:
3     if e in L2:
4       L1.remove(e)
5 L1, L2 = [1, 2, 3, 4], [1, 2, 5, 6]
6 remove_dups(L1, L2) # Before running this, predict result!
```

- What is happening ?
  - With `for ... in ...` Python uses an internal counter to keep track of index.
  - `.remove` changes the length of the list, but does not update internal counter! Element `2` is never handled by loop!

Lists in memory – side effects

- Compare `sort` and `sorted` with respect to their **side effects**. Investigate in PythonTutor.
- Side effects on nested lists can be very tricky!

Expressions on lists – `enumerate` and `zip`

- Many nice functions available that do cool things with lists
- `enumerate` provides the element index of the iterated element
  - ```
    my_list = [1, 2, 3]
    for (i, elem) in enumerate(my_list):
        print("{}-th element is {}".format(i, elem))
    ```
- `zip` merges the elements of various lists at the same index position
  - ```
    alist = ['a', 'b', 'c']
    blist = [1, 2, 3]
    for (aelem, belem) in zip(alist, blist):
        print(aelem, belem)
    ```

List comprehension

- A way to create a new list by “mapping” an existing list in a very compact way.
- `new_list = [<map exp> for <name> in <iter exp>]`

```
1 odds = [1, 3, 5, 7, 9]
2 evens = [(num + 1) for num in odds]
3 # or
4 letters = ['a', 'b', 'c', 'd', 'e', 'f', 'm', 'n', 'o', 'p']
5 word = [letters[i] for i in [3, 4, 6, 8]]
```

- Apply a *filter* (boolean expression) to every element:
  - `new_list = [<map exp> for <name> in <iter exp> if <filter exp>]`

```
1 temps = [60, 65, 71, 67, 77, 89]
2 hot = [tmp for tmp in temps if tmp > 70]
3 # observe in PythonTutor!
```

Dictionaries

- A `dict` (*hash map*) is a mapping of **key-value pairs**.
- It *somewhat* resembles a list: We use the **key** to **look-up** (*indexing* in a list) the associated **value** stored in the dictionary.

```
states = {
    "CA": "California",
    "DE": "Delaware",
    "NY": "New York",
    "TX": "Texas",
    "WY": "Wyoming"
}
```

- Dictionaries support similar operations as lists/strings:

```
len(states)
"CA" in states
"ZZ" in states # Error if key not available
```

Dictionary access

```
words = dict(
    "más"="more",
    "otro"="other",
    "agua"="water"
)
```

Ways to access a value by key:

```
words["otro"]
first_word = "agua"
words[first_word]
words["pavo"]
words.get("pavo", "🤔")
```

Dictionary rules

- All keys in a dictionary are distinct
- A key cannot be a list or dictionary (or any other **mutable type**)
- There is only *one value per key*!
  - The values can be any type, however!
  - For example, a list or a dictionary!

```
spiders = {
  "smeringopus": {
      "name": "Pale Daddy Long-leg",
      "length": 7
  },
  "holocnemus pluchei": {
      "name": "Marbled cellar spider",
      "length": (5, 7)
  }
}
```

Dictionary iteration

```
1 insects = {"spiders": 8, "centipedes": 100, "bees": 6}
2 for name in insects:
3     print(insects[name])
```

- What will be the order of items?
  - In general hard to determine!

Dictionary methods

- Use `.keys()`, `.values()` and `.items()` to iterate over the keys, values or items respectively.

```
1 insects = {"spiders": 8, "centipedes": 100, "bees": 6}
2 for name in insects.keys(): # also try with .values and .items!
3     print(name)
```

- Sometimes we know that a key should have a default value!

```
1 from collections import defaultdict
2 insects = defaultdict(int)
3 print(insects['unknown'])
```

Dictionary comprehension

Similar to list comprehension, build a dictionary in a very compact way.

```
1 new_dict = {some_key_expr : some_value_expr for <name> in <iter exp
2 # note the { } brackets!
```

`list` vs `dict`

- Ordered sequence of elements.
- Look up elements by an *integer index*.
- Indices have an order.

- **Matches** *keys* to *values*.
- Look up an element by some (almost) arbitrary key.
- **No order** on keys is guaranteed.
- *key* can be any **immutable** type.

File I/O

File I/O

- A **file** is a set of (logically) connected data which are recorded on a storage device (*hard disk*).
- Hence, a file can exist **beyond the runtime of a program**.
  - A file is a potentially shared resource!
- Opening a file in python: `f = open(filename, option)`
  - `filename`: A file is identified by its *filename* (potentially a complete *path*).
  - `option`: `r` to read a file, `w` to write to a file, `a` to append to an existing file.
    - We are assuming *text files* for now!
- Need to close a file after done working with it: `f.close()`.

Reading files

- `read()`: Read entire content (or first n characters, if supplied)
  - Remember: We are dealing with text files!
- `readline()`: Reads a single line per call
- `readlines()`: Returns a list with lines (splits at newline)
- When dealing with files, we often use a **context manager** (*maybe* more later?!):

```
1 with open('myfile.txt', 'r') as f:
2   for line in f: # no 'read' necessary!!
3   print(line)
```

- It is a good practice do use a context manager (that is, the `with ...` statement) when dealing with files.
  - At the end of the code block it is ensured that the file is closed.

Writing to a file

- Use `write()` to write to a (text) file

```
1 name = "Sissi"
2 with open(filename, 'w') as f:
3   f.write("Hello, {}!\n".format(name)) # Wait, what is this?
```

- Or **format** the written text in a different way:

```
1 name = "Sissi"
2 with open(filename, 'w') as f:
3   f.write(f'Hello, {name}!\n') # Wait, what is this now?
```

- Look up f-strings

More file writing

Write elements of a list to a file

```
1 filename = 'mylist.txt'
2 xs = [1, 2, 3]
3 with open(filename, 'w') as f:
4     for x in xs:
5         f.write(f'{x}\n')
```

Write elements of a dictionary to a file

```
1 filename = 'mydict.txt'
2 d = {'a': 3, 'b': 4}
3 with open(filename, 'w') as f:
4     for k, v in d.iteritems():
5         f.write(f'{k}: {v}\n')
```