15-110 CMU-Q: A reference card for Python - Part II

April 4, 2019[*]

Contents

Dictionary definitions

Dictionaries are non-scalar, *mutable* types useful for representing collections of data resources that can be accessed through specific keyword identifiers (labels). A dictionary maps *keys* into *values*. Keys are all different / unique and can only defined by immutable types. Values can be anything, any data structure or type. Different keys might be associated to a same value (representing however logically different data records).

A dictionary is unordered: it's not a sequence, items are accessed through the keys and, not by their position in a sequence, as in lists.

- empty dictionary: `{}`
- dictionary with three keys, all strings, each associated to an integer value:
  `d = {'John': 22, 'Jim': 20, 'Mary': 20, 'Paul': 29}`
- dictionary with two integer keys, with float and string values: `d = {22:'John', 75: 35.2}`
- dictionary with two integer keys and list values:
  `d = {2:[1,2,3], 3: [5,7,11]}`
- dictionary defined from a sequence:
  `words = dict([('This', 4), ('is', 2), ('list', 4)])`
- dictionary defined from a sequence:
  `parabola = dict( [(0,0), (0.5, 0.25), (1,1)] )`
- An incorrect definition, with the key being a mutable type: `d = {[2,3]:'Incorrect'}`
- definition of a new dictionary as a clone of an existing one: `d_new = d_exist.copy()`
- definition of a new dictionary as an alias of an existing one: `d_new = d_exist`
- definition of a dictionary using list of keys with default values, `fromkeys()` method:
  ```
  primes = [2, 3, 5, 7]
  primes_dict = dict.fromkeys(primes, 'p')
  ```
- definition of a dictionary using two lists and `zip()` function:
  ```
  list_of_keys = [1, 2, 3, 4, 5, 6];
  list_of_values = ['r', 'p', 'p', 'r', 'p', 'r' ];
  numbers = dict(zip(list_of_keys, list_of_values))
  ```

Notes

[*]Contact G. A. Di Caro for pointing out mistakes, missing information, and for suggestions (`gdicaro@cmu.edu`)

[†]Note: In the following, when the parameter of a function is optional, it is enclosed in `<>` brackets, e.g., `f(x,<y>)`. When a function returns something different than `None`, the notation `var = f()` is adopted, otherwise the function is plainly described as `f()`.

Dictionary operators

- *Inserting*: `[key]` operator, that allows to insert a new pair (key, value) in the dictionary. `d['red'] = 245`
- *Reading*: `[key]` operator, that returns the values associated to key, if key is in the dictionary, error otherwise. `intensity = d['red']`
- *Modifying*: `[key]` operator that allows to to modify the value associated to an existing key. `d['red'] = 245`
- *Comparison:* comparisons of equality between two dictionaries can be done using the relational operator `==` that returns `True` if the dictionaries have exactly the same content (keys and values), `False` otherwise. Opposite for the operator `!=`. Other relational operators do not apply to dictionaries.
- *Membership*: `in`, `not in`, where `key in d`, evaluates to `True` if key `key` is a one of the keys of dictionary `d`.

Notes

Dictionary methods

- `l = d.keys()`
- `l = d.values()`
- `list_of_key_val_pairs = d.items()`
- `v = d.get(key, <value>)`
- `v = d.pop(key, <value>)`
- `(key,val) = d.popitem()`
- `d.clear()`
- `d.update(iterable)`, where `iterable` is an iterable of `(key, value)` such as a dictionary or a sequence `[('r',10), ('b', 90), ('g', 122)]`
- `d.setdefault(key, <value>)`
- `d_new = d.copy()`
- `d.fromkeys(key_list, <value> )`

Notes

Set definitions

Set are non-scalar, *mutable* types useful for representing *unordered collections* of items where every element is *unique* (no duplicates) and must be immutable. The set itself is mutable: elements can be inserted and removed, aliases between sets can be created. All the usual mathematical set operations apply to set object types.

- empty set: `s = set()`
- set with three elements, of different immutable types:
  `s = {('John', 'Ann'), 22, 4.56}`
- set from a list:
  `l = [1, 2, 3, 4.5, 5.3, True, (1,2)]`
  `new_set = set(l)`
- set from a dictionary (using the keys):
  `n = {1: 'p', 2: 'p', 3:'p', 4:'r'}`
  `new_set = set(numbers)`
- set from a dictionary (using the values):
  `n = {1: 'p', 2: 'p', 3:'p', 4:'r'}`
  `new_set = set(numbers.values())`
- set from a string (sequence):
  `new_set = set("apple")`, that results in a set of 5 elements, one per each character.
- set from a list of strings (sequence of sequences):
  `new_set = set(["apple", 'peach'])`, that results in a set of 2 string elements, one per each string.
- error, set with a mutable type element:
  `s = {['John', 'Ann'], 22, 4,56}`

– definition of a new set as a clone of an existing one: `s_new = s_exist.copy()`

– definition of a new set as an alias of an existing one: `s_new = s_exist`

Notes

- *Superset:* `A >= B` returns `True` if A is a superset of B, `False` otherwise. `A > B` returns `True` if `A >= B` and `A != B`, `False` otherwise.

- *Membership*: `in`, `not in`, where `val in s`, evaluates to `True` if element `val` is a one of the elements of set `s`.

Notes

Set operators

- *Union*: `|` operator, returns the union of two sets, `C = A | B`. In-place union (modify A): `A |= B`.

- *Intersection*: `&` operator, returns the intersection of two sets, `C = A & B`. In-place intersection (modify A): `A &= B`.

- *Difference*: `-` operator, returns the difference between two sets, `C1 = A - B; C2 = B - A`. In-place difference (modify A): `A -= B`.

- *Symmetric Difference*: `^` operator, returns the symmetric difference between two sets (everything in A and in B but not in their intersection), `C = A ^ B`. In-place symmetric difference (modify A): `A ^= B`.

- *Equality:* comparisons of equality between two sets can be done using the relational operator `==` that returns `True` if the set have exactly the same content, `False` otherwise. Opposite for the operator `!=`.

- *Subset:* `A <= B` returns `True` if A is a subset of B, `False` otherwise. `A < B` returns `True` if `A <= B` and `A != B`, `False` otherwise.

Set methods

- `s.add(item)`
- `s.update(iterable)`, where iterable can be a string, tuple/list, dictionary, set, and add each element into the set, no duplicates are inserted.
- `s.discard(item)`, no error if item is not in set.
- `s.remove(item)`, error if item is not in set.
- `new_set = a.union(b)`
- `a.update(b)`
- `new_set = a.intersection(b)`
- `a.intersection_update(b)`
- `new_set = a.difference(b)`
- `a.difference_update(b)`
- `new_set = a.symmetric_difference(b)`
- `a.symmetric_difference_update(b)`
- `a.issubset(b)`
- `a.issuperset(b)`

Notes

Files

Files are custom, permanent data structures that, in general, reside on a physical support other than the RAM. Files come with a number of class methods for opening, reading, writing file contents. Files can be of binary or text type. Binary files requires a specific encoding and possibly a specific program to read/write them. Text files are organized in multiple records of data, separated by newline characters, where each record is composed of one or more fields. Record composition doesn't need to be uniform across the file. The size of a file is measured in bytes, where a byte is 8 bits. An ASCII character is encoded in a byte. More complex encodings (UTF) require in general more bytes.

File methods

- `f_handler = open(filename, <mode>)`,
  where mode can include combinations of:
  `'r', 'r+', 'w', 'w+', 'a', 'a+', 'x', 'b', 't'`
- `string_with_data = f.read(<number_of_bytes>)`
- `position = f.seek((<position>)`
- `position = f.tell(()`
- `record = f.readline()`
- `remaining_records = f.readlines()`
- `written_bytes = f.write(string_to_write)`
- `read_mode = f.readable()`
- `write_mode = f.writable()`
- `f.flush()`
- `f.close()`

Notes

File system

Module `os` (Operating System) (`import os`) offers a complete set of functionalities to inspect and manipulate the elements of the file system of the computer. Also other modules exist with similar functionalities. In the file system, a file or a folder (directory) is identified by its *name* and its *path* relative to the root of the file system (relative to the physical support). Files and folders in the file system defines a tree data structure.

File system methods

- `current_working_directory = os.getcwd()`
- `path = os.path.join(comma_sep_list_of_sub_folders)`
- `file_exist = os.path.isfile(file_name)`
- `folder_exist = os.path.isdir(folder_full_path_and_name)`
- `folder__or_file_exist = os.path.exists(name)`
- `file_list = os.listdir(<path>)`
- `os.chdir(path)`
- `info_stat = os.stat(file_name)`;
  `info_stat` is a data structure with multiple fields,
  e.g., `info_stat.st_size` or `info_stat.st_mtime`.
- `os.mkdir(new_file_or_folder_path)`
- `os.remove(path)`

Notes

CSV files

CSV files (comma-separated files), are commonly used formats for data storing and sharing. CSV files are text data files where the records shares the same field structure across the file, and are separated by newlines. In spite of the name, in CSV files it is possible to select the field separator, quoting and escape characters, as well as conveniently treat the records as a sequence of ordered dictionaries.

The csv module (`import csv`) offers a number of methods for the effective manipulation of CSV files both in reading and writing modes. A CSV file handler is constructed that allows a flexible access to the file. The file must be first opened in the appropriate mode using the usual open call: `f_csv = open(file_path)`.

CSV file methods

- `csv_data = csv.reader(f_csv, <delimiter=',', <quotechar='"'>, <escapechar='\'>)`
- `csv_data_dict_reader = csv.DictReader(f_csv, <>)`
- `csv_writer = csv.writer(f_csv, <delimiter=',', quotechar='"', quoting=csv.QUOTE_MINIMAL>)`
- `csv_writer.writerow(list_of_data_fields)`
- `csv_writer.writerows(list_of_lists_of_fields)`
- Header is not needed (but still useful) if the file will not be read in dictionary mode:
  `header = ['name','dept','since','score']; csv_writer.writerow(header)`
- Header is needed if the file will be read in dictionary mode:

```
names = ['name','dept','since','score']
csv_writer = csv.DictWriter(f_csv, fieldnames=names)
csv_writer.writeheader()
```

- `next(csv_data)`, this is a function, not a method, that makes an iterator to execute one step, for a file means that the reading/writing position is moved one record down.

Notes

String formatting

The basic string methods offer a variety of possibilities to manipulate strings. However, in order to flexibly combine numeric and textual data into string data (e.g., to write to files or display as output) the *Formatter method* `format()` results particularly useful.

The general format includes a string with format specifiers and the specification of the variable / literals to include in the string.

- *Positional substitution, no format specifiers*:
  `'{} {} {} {} {}'.format(name, title, age_years, height, weight)` produces
  `John Mr. 30 178.57142857142858 80.83999999999999`
  (based on the specific values of the variables / literals in the `format()` part of the call).
- *Positional substitution, format specifiers*:
  `'{:12s} {:4s} {:3d} {:8.3f} {:8.3f}'.format(name, title, age_years, height, weight)` produces
  `John Mr. 30 178.571 80.840`.
- *Keyword (name) substitution, format specifiers*:
  `'{Name:12s} {Title:4s} {Age:3d} {Height:8.3f} {Weight:8.3f}'.format(Name=name, Title=title, Age=age_y, Height=height, Weight=weight)` produces `John Mr. 30 178.571 80.840`.

- *Mixed, keyword-positional substitution*: `'{} {} {:3d} {Height:g} {Weight:e}'.format(name, title, age, Height= height, Weight=weight)` produces `John Mr. 30 178.571 8.084e+01`.
- *Alignment:* by default string fields are left-aligned, numeric fields are right-aligned. `<` and `>` align respectively left and right.

Notes

Handling exceptions

When an error occurs during the program, Python generates an exception: it generates an error type that identifies the exception and then stops the execution. Exceptions can be handled using the `try` statement to avoid that the program stops when an error occurs during the execution.

`try-except-else-finally` blocks:

- The `try` block let executing a block of code that can potentially generate an exception
- The `except` block let handling the error, if generated by the try block (i.e., what to do when an error occurs)
- The `else` block let specifying a block of code that is executed if the try block didn't generate any exception
- The `finally` block let executing the code, regardless of the result of the try- and except blocks.
- Example of use:

```
y =1 try:
x /= 10
y += x except:
    print("x doesn't exist")
else:
    print('x:', x)
    del x
finally:
    print('y:', y)
```

Notes

Classes

Classes and object-oriented programming are fundamental in modern software development. A class is a like a blueprint: a design guide, an operational pattern that can be used to create multiple, independent, instances of class objects. Different instances have the same design, are equipped with the same methods and attributes, but attributes might have different values. In any case, two instances are physically different class objects.

Classes provide a high level of abstraction and information hiding/encapsulation, and comes with useful properties such as polymorphism (overloading) and inheritance.

General syntax for class definition:

```
class ClassName:
   def __init__(self, args):
       # code for object initialization
   def one_class_method(self, args):
      # code for this method
   def another_class_method(self, args):
      # code for this method
```

where the method `_init__{}` is the *initializer* of the class and should always be there. The rest of the class definition can feature as many methods as needed. The `self` variable is the reference address to the specific class object created from the abstract class definition (the blueprint).

Example of a full class:

```
class Robot:
   def __init__(self, robot_type='wheeled',
                brand=None, model= None, price=0,
                speak=False):
     self.type  = robot_type
     self.brand = brand
     self.model = model
     self.price = price
     self.speak = speak
```

```
  def can_speak(self):
    return self.speak

  def set_speak(self, speak):
    self.speak = speak

  def get_info(self):
    info = 'Model {} made by {} is a {} robot,
           price is {}, speaks [{}]'.format(
           self.model, self.brand, self.type,
           self.price, self.speak)
    return info
```

Example of use of the class, by creating two instances, one for a flying robot, and one for a humanoid robot:

```
fr = Robot('flying', 'Boeing', '799A')
hr = Robot('humanoid', 'Darwins', 'iHuman',
           speak=True, price=1000000)
fr.set_price(2050000)
print(hr.can_speak())
```

A class can be *derived* from a *base* class, such that the derived class *inherits* all the methods and attributes of the base class. In turn, the methods can be overloaded, and new methods and attributes can be defined.

An example of a simple class `Cyborg` that inherits from Robot (but it extends it since a cyborg has both robotic and human components):

```
class Cyborg(Robot):
   def __init__(self, human_parts=[], brand=None,
                model= None, price=0):
       Robot.__init__('cyborg', brand, model, price, True)
     self.human_parts = human_parts

   def get_info(self):
      info = Robot.info(self)
      info += 'Human parts are: {}'.format(self.human_parts)
      return info
```

Notes

Recursion

A function that makes a call to itself is a recursive function. A recursive function definition includes base case(s), that don't require recursion and where the recursive calls stop, and a recursive case, where the function calls itself, possibly with a simpler instance / smaller input value. A typical example is that of the factorial function:

```
def factorial_recursive(n):
    if n == 1 or n == 0:
        return 1
    else:
    return n * factorial_recursive(n-1)
```

Recursion might incur into a large occupancy of the runtime memory stack (each recursive call is allocated on the stack) and/or a large number of recursive calls, resulting on a slow performance. There are many ways to optimize both aspects. A good design of the recursive case is essential to minimize the number of recursive calls that are strictly necessary for the task. The use of *memoization*, where already performed computations are saved (e.g., in a dictionary) and reused if/when necessary again, can dramatically save computation and made recursion performing very well.

Notes