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

February 10, 2024[*†]

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

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.
- *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 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)`

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

Notes

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()`

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.

Notes

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)`

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)`.

Notes

Notes

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.

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.

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

Notes

- 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)
```

Matplotlib and Pyplot

Module `matplotlib` provides an extensive set of tools for the graphical display of data. We have focused on the use of submodule `pyplot`, which is quite handy to use and is based on the popular *Matlab* software. `pyplot` can be imported using the following statement:

```
import matplotlib.pyplot as plt
```

In matplotlib, once a new figure is created, multiple graphical elements can be added and/or specified (to have a behavior different from the default). This means that a graphical figure can include, for instance, multiple datasets displayed according to different formats (e.g., a scatter plot, a histogram, a plot, a box plot, etc.).

In the following a list of useful methods from pyplot and matplotlib are described.

Notes

- `plt.figure()` creates a new figure; it is not strictly necessary (but highly recommended) to invoke it, since a new figure will be created implicitly when invoking any plotting method.
- `plt.figure(figsize(xsize, ysize))` allows to specify the dimensions of the figure along x and y.
- `plt.figure()`
  `fig, subplots = plt.subplots(nrows, ncols, figsize=(xsize, ysize))`

  allows to place multiple plots (*subplots*) in the same figure; the `subplots(nrows, ncols, figsize)` method defines a grid (a matrix) of `nrows` $\times$ `ncols` locations where the different plots will be placed. Overall, the figure will have the size defined by `figsize`.

  Graphical elements can be added to subplot (i, j) by using the notation: `subplots[1, 0].hist(my_data)` that for instance adds a histogram plot to the subplot in row 1 and column 0.

  If only one row of subplots is used, then the notation `subplots[i]` should be used.
- `plt.title(title)` uses the string `title` as a title for the plot. The optional parameter `fontsize` can be used to specify the size of the text: `plt.title('My title', fontsize=20)`. The parameter `fontsize` can be also be used in `plt.xlabel()`, `plt.ylabel()`.
- `plt.xlabel(label)`, `plt.ylabel(label)` use the string `label` as the label for the selected x, y axis.
- `plt.xlim(a, b)`, `plt.ylim(a, b)` set the limits for the values displayed on the x and y axis.

- `plt.xticks(ticks_pos, <my_labels>)`, `plt.yticks(ticks_pos, <my_labels>)` control the appearance of the ticks on the x and y axis, where `ticks_pos` is a sequence defining where the tick marks should be placed, the optional `my_labels` sets explicit labels to be placed at the given `ticks_pos`.

- `plt.plot(y_seq)` plots the data in the sequence `y_seq`; since x coordinates are unspecified, the point data are plotted in the same way as in `plt.plot(x_seq, y_seq)`, where `x_seq = range(0,len(y_seq))`: data points are plotted at equally spaced integer x coordinates. The default behavior is to plot the data as a 2D *solid line*, such that `plot()` should be seen as a *line plot*.

- `plt.plot(x_seq, y_seq)` plots the given *pairs* $(x_{seq}, y_{seq})_i,\ i = 0, \ldots, n$, where n is the length of the sequences. Again, the default behavior is to plot the data as a 2D *solid line*.

- `plot()` can be customized by setting many positional and keyword-passed parameters, a few examples are given below, different combinations of the parameters can be used to get different effects:

  - `plt.plot(x, y, marker='.', markersize=4)`
  - ```
    plt.plot(x, y, linestyle='None', marker='v',
             color='b')
    ```
  - ```
    plt.plot(x, y, linestyle='--', linewidth=2,
             color='r')
    ```
  - ```
    plt.plot(x, y, linestyle='-', linewidth=1,
             marker='s', markersize=8,
             color=(0.1, 0.4, 0.3))
    ```
  - `plt.plot(x, y, 'ro', linestyle=':')`

- `plt.scatter(x_seq, y_seq)` makes a *scatter plot* of the pairs $(x_{seq}, y_{seq})_i,\ i = 0, \ldots, n$: no lines are drawn between points. A scatter plot *needs* two input sequences to create the point pairs.

- `plt.scatter(x, y, marker='.', s=12, color='r')` makes a scatter plot using the selected marker, defining its size with the keyword parameter `s`, and setting the color. `color` and `marker` arguments follow the same options of the `plot()` method.

- `plt.hist(values, n_intervals)` plots a *histogram* for the data in `values` by considering their distribution in `n_intervals` bins/intervals of the same size. The color of the histogram can be specified by using the keyword argument `color`, a normalized histogram is obtained by setting the option `density=True`.

- `plt.bar(x_pos, heights)` makes a *bar plot* where `x_pos` are the x coordinates of the bars, and `heights` are the heights of the bars. The optional argument `width` can be used to control the width of the bars (default is 0.8). Filled color can be set by the optional argument `color`. Tick labels on the x axis can be set with the argument `tick_label`, for instance `tick_label = ['1998', '2010', '2015']`, or `tick_label= [100, 200, 1000]`.

- `plt.pie(values)` makes a *pie chart* of data `values` where the fractional area of each wedge i is given by `values[i]/sum(values)`. The `labels` optional parameter allows to pass a list of strings for the labels of each wedge. The optional parameter `colors` allows to specify a sequence of colors for the wedges (matplotlib will cycle the sequence if it is less than the number of wedges).

- `plt.legend(handles=list_of_label_references)` allows to place a *legend* for each element in the plot:

  ```
  plt.figure()
  y, = plt.plot(x, y, label='y')
  xx, = plt.plot(x, x**2, label='$x^2$')
  xs = plt.scatter(x, x, label='x')
  plt.legend(handles=[y, xx, xs])
  plt.show()
  ```

  Note the commas after `y` and `xx`, as well as the use of `$ $` to enclose and expression using the LaTeX syntax to generate a nice math-like formatting of the output.

- `plt.show()` completes the figure and shows it on the screen. It is not strictly necessary, but it's better to include it.

- `plt.savefig(filename)` saves the figure in the given image file name (the extension sets the image type). The optional integer argument `dpi` sets the resolution. If the `transparent` options is set to `True`, the image backgorund is transparent. Othe options are available.
  ```

Notes

Numpy module

The `numpy` module is the main module providing mathematical and numerical tools. We haven't really explored the potentialities of the module apart from using the basic method `arange(from, to, step)` that generalizes the built-in function range to float numbers. Below a few methods and examples from the library are listed:

- `import numpy as np` is the typical way to import the module.
- `seq = np.arange(from, to, step)`
  where `from, to, step` are floats, for the rest it follows the same rules as `range()`.
- If `x` and `y` are created as numpy *arrays*, then arithmetic operators can be applied to them (as long as the arrays have the same length):

```
x = np.arange(0, 1, 0.1)
y = np.arange(2, 3, 0.1)
z = x + y
zz = x * y
```

- `numpy` provides most of the mathematical functions provided by `math`, with the same names, such `np.sqrt(x), np.sin(x)`, etc.
- `np.average(z), np.median(z)`

Notes

Summary of string methods

- `capitalize()`: Converts the first character to upper case
- `casefold()`: Converts string into lower case
- `center()`: Returns a centered string
- `count()`: Returns the number of times a specified value occurs in a string
- `encode()`: Returns an encoded version of the string
- `endswith()`: Returns true if the string ends with the specified value
- `expandtabs()`: Sets the tab size of the string
- `find()`: Searches the string for a specified value and returns the position of where it was found
- `index()`: Searches the string for a specified value and returns the position of where it was found
- `isalnum()`: Returns True if all characters in the string are alphanumeric
- `isalpha()`: Returns True if all characters in the string are in the alphabet
- `isdecimal()`: Returns True if all characters in the string are decimals
- `isdigit()`: Returns True if all characters in the string are digits
- `isidentifier()`: Returns True if the string is an identifier

- `islower()`: Returns True if all characters in the string are lower case
- `isnumeric()`: Returns True if all characters in the string are numeric
- `isprintable()`: Returns True if all characters in the string are printable
- `isspace()`: Returns True if all characters in the string are whitespaces
- `istitle()`: Returns True if the string follows the rules of a title
- `isupper()`: Returns True if all characters in the string are upper case
- `join()`: Joins the elements of an iterable to the end of the string
- `ljust()`: Returns a left justified version of the string
- `lower()`: Converts a string into lower case
- `lstrip()`: Returns a left trim version of the string
- `partition()`: Returns a tuple where the string is parted into three parts
- `replace()`: Returns a string where a specified value is replaced with a specified value
- `rfind()`: Searches the string for a specified value and returns the last position of where it was found
- `rindex()`: Searches the string for a specified value and returns the last position of where it was found
- `rpartition()`: Returns a tuple where the string is parted into three parts
- `rsplit()`: Splits the string at the specified separator, and returns a list
- `rstrip()`: Returns a right trim version of the string
- `split()`: Splits the string at the specified separator, and returns a list
- `splitlines()`: Splits the string at line breaks and returns a list
- `startswith()`: Returns true if the string starts with the specified value
- `swapcase()`: Swaps cases, lower case becomes upper case and vice versa
- `title()`: Converts the first character of each word to upper case
- `upper()`: Converts a string into upper case
- `zfill()`: Fills the string with specified number of 0 values at the beginning