Matetials: Python Standard Library

Yifei Wang pppppass

February 26, 2018

Contents

1 Brief introduction

The main purpose of this speech is to introduce several commonly used Python library. The standard library to Python is like the Webster Dictionary to English. Though reading the official document is the most recommended way to familiarize oneself with the standard library, using **help()** or **dir()** is a rather convenient option. It can deal with the majority of the problems that we meet.

2 Resources

The official document of Python Standard Library is provided here. The index is provided here. Python 标准库 is the Chinese version. Python 3 Module of the week is a strong complementary material besides the official document. You can also access to a bunch of useful information in 应该怎样系统的学习 Python 标准库.

3 Python Standard Library

3.1 datetime

The `datetime` module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation.

1. Available types
    - **date** An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: *year*, *month*, and *day*.
    - **time** An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds. Attributes: *hour*, *minute*, *second*, *microsecond*, and *tzinfo*.
    - **datetime** A combination of a date and a time, with all attributes that *date* and *time* have.
    - **tzinfo** An abstract base class for time zone information objects. These are used by the *datetime* and *time* classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).
    - **timezone** A class that implements the *tzinfo* abstract base class as a fixed offset from the UTC.

2. *timedelta* objects

   A *timedelta* object represents a duration, the difference between two dates or times.

   class `datetime`.**timedelta**(*days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0*)

   All arguments are optional and default to 0. Arguments may be integers or floats, and may be positive or negative.

   Only *days*, *seconds* and *microseconds* are stored internally. The representation is unique, with:

   - $0 \leq microseconds < 1000000$
   - $0 \leq seconds < 3600 * 24$
   - $-999999999 \leq days \leq 999999999$

   Special supported operation:

   - **abs(t)** equivalent to $+t$ when $t.days \geq 0$, and to -t when $t.days < 0$.
   - **str(t)** returns a `string` in the form [D day[s],][H]H:MM:SS[.UUUUUU], where D is negative for negative t.

3. *date* Objects

   A *date* object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar indefinitely extended in both directions.

   class `datetime`.**date**(*year, month, day*)

   All arguments are required. Arguments may be integers, in the following ranges:

   - MINYEAR $\leq year \leq$ MAXYEAR
   - $1 \leq month \leq 12$
   - $1 \leq day \leq$ number of days in the given month and year

   class methods:

   - **today()** returns the current local date
   - **fromtimestamp**(*timestamp*) returns the local date corresponding to the POSIX timestamp, such as is returned by *time.time()*.
   - **fromordinal**(*ordinal*) returns the date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1.

   instance methods:

   - **replace**(*year=self.year, month=self.month, day=self.day*) returns a date with the same value, except for those parameters given new values by whichever keyword arguments are specified.

- **toordinal()** returns the proleptic Gregorian ordinal of the date, where January 1 of year 1 has ordinal 1.
- **isoweekdat()** returns the day of the week as an integer, where Monday is 1 and Sunday is 7.
- **isocalendar()** returns a 3-tuple, (ISO year, ISO week number, ISO weekday).
- **isoformat()** returns a string representing the date in ISO 8601 format, 'YYYY-MM-DD'.

4. *time* Objects

   A time object represents a (local) time of day, independent of any particular day, and subject to adjustment via a *tzinfo* object.

   class `datetime`.**time**(textithour=0, minute=0, second=0, microsecond=0, tzinfo=None, *, fold=0) All arguments are optional. Instance methods:

   - **replace**(*hour=self.hour, minute=self.minute, second=self.second, microsecond=self.microsecond, tzinfo=self.tzinfo, * fold=0*) returns a *time* with the same value, except for those attributes given new values by whichever keyword arguments are specified.
   - **isoformat**(*timespec='auto'*) returns a string representing the time in ISO 8601 format, HH:MM:SS.mmmmmm The optional argument *timespec* specifies the number of additional components of the time to include (the default is 'auto').

3.2 math

Generally, `math` provides access to the mathematical functions defined by the C standard. It is efficient and with accuracy when calculating with float numbers. If you require support for complex number, use the functions of the same name from the `cmath`. Detailed descriptions about `math`, including inf and *nan* can be found in here.

1. Number-theoretic and representation functions
   - **floor**(x), **ceil**(x) returns the floor(ceiling) of x.
   - **fabs**(x) returns $|x|$.
   - **factorial**(x) returns $x!$, raises *ValueError* if x is not an integral or is negative.
   - **fmod**(x, y) returns $x - n * y$ for some integer n such that the result has the same sign as x and magnitude less than $|y|$.
   - **fsum** ($iterable$) returns an accurate floating point sum of values in the $iterable$ and avoids loss of precision by tracking multiple intermediate partial sums

- **gcd**(a, b) Return the greatest common divisor of the integers a and b.
- **isclose**$(a, b, *, rel\_tol = 1e-09, abs\_tol = 0.0)$ returns *True* if the values a and b are close to each other and *False* otherwise.

2. Power and logarithmic functions
   - **exp**(x) returns e^x
   - **log**$(x[, a])$ With one argument, returns $ln(x)$. With two arguments, returns $log_a x$.
   - **pow**(x, y) returns x^y
3. Trigonometric and Hyperbolic functions
   - **acos**(x), **asin**(x) **atan**(x) return $arccos(x)$, $arcsin(x)$, $arctan(x)$, in radians.
   - **cos**(x), **sin**(x), **tan**(x) return $cos(x)$, $sin(x)$, $tan(x)$
   - **hypot**(x, y) returns the length of the vector from the origin to point (x, y).
   - **acosh**(x), **asinh**(x) **atanh**(x) return $arccosh(x)$, $arcsinh(x)$, $arctanh(x)$, in radians.
   - **cosh**(x), **sinh**(x), **tanh**(x) return $cosh(x)$, $sinh(x)$, $tanh(x)$
4. Constants
   - **pi** returns π.
   - **e** returns e.
   - **inf** returns a floating-point positive infinity.
   - **nan** returns a floating-point "not a number" (NaN) value.

3.3 random

This module implements pseudo-random number generators for various distributions.

Almost all module functions depend on the basic function *random*().

Several commonly used functions in `random`:

- **random()** returns a random float number in [0.0, 1.0).
- **randrange(n), randrange(m,n), randrange(m,n,d)** returns a random integer in the interval (this refers to the definition of `range`).
- **randint**(m, n) is the same as **randrange**$(m, n + 1)$.
- **choice**(s) selects a random character in the string s.
- **seed(n), seed()** uses n or system time to reset the random number generator. It is used to restart a random series.
- **shuffle**(x) shuffles the sequence x in place.
- **sample**$(population, k)$ returns a k length list of unique elements chosen from the

population sequence or set.

`random` also provides functions that generate specific real-valued distributions. You can get more detailed listing in 9.6.4. Real-valued distributions

3.4 serialization: pickle and json

You can get an enlightening introduction about how to use these two modules in this page .

3.4.1 pickle

`pickle` implement binary protocols for serializing and de-serializing a Python object structure. *"Pickling"* is the process whereby a Python object hierarchy is converted into a byte stream, and *"unpickling"* is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy.
The pickle module provides the following functions to make the pickling process more convenient:

- **dump**(obj, file, protocol=None, *, fix_imports=True) writes a pickled representation of *obj* to the open file object file.
- **dumps**(obj, protocol=None, *, fix_imports=True) returns the pickled representation of the object as a bytes object, instead of writing it to a file.
- **load**(file, *, fix_imports=True, encoding="ASCII", errors="strict") reads a pickled object representation from the open file object file and return the reconstituted object hierarchy specified therein.
- **loads**(bytes_object, *, fix_imports=True, encoding="ASCII", errors="strict") reads a pickled object hierarchy from a bytes object and return the reconstituted object hierarchy specified therein.

The pickle module exports two classes, Pickler and Unpickler:

- class `pickle`.**Pickler**(file, protocol=None, *, fix_imports=True)
  This takes a binary file for writing a pickle data stream.
  dump(*obj*) writes a pickled representation of obj to the open file object given in the constructor.
- class `pickle`.**Unpickler**(file, *, fix_imports=True, encoding="ASCII", errors="strict")
  This takes a binary file for reading a pickle data stream.
  load() reads a pickled object representation from the open file object given in the constructor, and returns the reconstituted object hierarchy specified therein. Bytes past

the pickled object’s representation are ignored.

The types that can be pickled can be found in this page.

3.4.2 json

`json` is a lightweight data interchange format inspired by JavaScript object literal syntax.

basic usage:

- **dump**(obj, fp, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw) serializes obj as a JSON formatted stream to fp (a .write()-supporting file-like object) using this conversion table.
  All optional parameters are now *keyword-only*.
- **dumps**(obj, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw) serializes obj to a JSON formatted str.
  The arguments have the same meaning as in *dump()*.
- **load**(fp, *, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw) deserializes fp (a .read()-supporting file-like object containing a JSON document) to a Python object.
  All optional parameters are now *keyword-only*.
- **loads**(s, *, encoding=None, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw) deserializes s (a *str, bytes* or *bytearray* instance containing a JSON document) to a Python object.
  The other arguments have the same meaning as in load(), except *encoding* which is ignored and deprecated.

Besides, we can serialize a class object through utilizing other selective arguments in method *dumps()*. Detailed instruction can be found in here.

3.5 collections

This module implements specialized container datatypes providing alternatives to Python’s general purpose built-in containers, *dict, list, set,* and *tuple*.

1. ChainMap objects
   A *ChainMap* class is provided for quickly linking a number of mappings so they can be treated as a single unit. It is often much faster than creating a new dictionary and

running multiple update() calls. The class can be used to simulate nested scopes and is useful in templating.

All of the usual dictionary methods are supported. In addition, there is a *maps* attribute, a method for creating new subcontexts, and a property for accessing all but the first mapping:

- **maps** A user updateable list of mappings. The list is ordered from first-searched to lastsearched. It is the only stored state and can be modified to change which mappings are searched. The list should always contain at least one mapping.
- **new_child**($m = None$) returns a new *ChainMap* containing a new map followed by all of the maps in the current instance.
- **parents** Property returning a new *ChainMap* containing all of the maps in the current instance except the first one.

2. Counter objects

   A *counter* tool is provided to support convenient and rapid tallies.

   Counter objects support three methods beyond those available for all dictionaries:

   - **elements()** returns an iterator over elements repeating each as many times as its count. Elements are returned in arbitrary order.
   - **most_common**([n]) returns a list of the n most common elements and their counts from the most common to the least.
   - **subtract**([$iterable - or - mapping$]) Elements are subtracted from an *iterable* or from another *mapping* (or *counter*).

3. deque objects

   Deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction.

   Deque objects support the following methods:

   - **append/appendleft**(x) adds x to the right/left side of the deque.
   - **clear()** removes all elements from the deque leaving it with length 0.
   - **copy()** creates a shallow copy of the deque.
   - **count**(x) counts the number of deque elements equal to x.
   - **extend/extendleft**($iterable$) extends the right/left side of the deque by appending elements from the $iterable$ argument.

- **index**(x[, $start$[, $stop$]]) returns the position of x in the deque (at or after index $start$ and before index $stop$).
- **insert**(i, x) inserts x into the deque at position i.
- **pop()/popleft()** removes and returns an element from the right/left side of the deque.
- **remove**($value$) removes the first occurrence of $value$.
- **reverse()** reverse the elements of the deque in-place and then return None.
- **rotate**(n) rotates the deque n steps to the right. If n is negative, rotate to the left.

4. defaultdict objects

   class `collections`.**defaultdict**([$default_factory$[, ...]])

   returns a new dictionary-like object. defaultdict is a subclass of the built-in `dict` class. It overrides one method and adds one writable instance variable. The remaining functionality is the same as for the `dict` class

5. nametuple()

   Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code. They can be used wherever regular tuples are used, and they add the ability to access fields by name instead of position index.

   `collections`.**namedtuple**(typename, field_names, *, verbose=False, rename=False, module=None)

6. OrderedDict()

   Ordered dictionaries are just like regular dictionaries but they remember the order that items were inserted. When iterating over an ordered dictionary, the items are returned in the order their keys were first added.

   class `collections`.**OrderedDict**([$items$])

   An OrderedDict is a dict that remembers the order that keys were first inserted.

7. UserDict/UserList/UserString

   They are wrapper around dictionary/list/string objects for easier dict subclassing

3.6 re

This module provides regular expression matching operations similar to those found in Perl. Generally, we use Python’ s raw string notation for regular expression patterns. Backslashes are not handled in any special way in a string literal prefixed with ’r’. For instance, r"\n" is a two-character string containing ’\’ and ’n’, while "\n" is a one-character string containing a

newline.

A regular expression (or RE) specifies a set of strings that matches it; the functions in this module let you check if a particular string matches a given regular expression (or if a given regular expression matches a particular string, which comes down to the same thing).

Certain special characters:

- `.` In the default mode, this matches any character except a newline.
- `+` causes the resulting RE to match 1 or more repetitions of the preceding RE.
- `?` causes the resulting RE to match 0 or 1 repetitions of the preceding RE.
- `{m}` specifies that exactly m copies of the previous RE should be matched; fewer matches cause the entire RE not to match.
- `{m,n}` causes the resulting RE to match from m to n repetitions of the preceding RE, attempting to match as many repetitions as possible.
- `\` Either escapes special characters, or signals a special sequence.
  - `\d` matches any Unicode decimal digit.
  - `\s` matches Unicode whitespace characters.
  - `\S` matches any character which is not a whitespace character.
  - `\w` matches Unicode word characters; this includes most characters that can be part of a word in any language, as well as numbers and the underscore.
  - `\W` matches any character which is not a word character.
- `[ ]` is used to indicate a set of characters.

Module Contents:

- **compile**($pattern, flags = 0$) compiles a regular expression pattern into a regular expression object, which can be used for matching using its *match(), search()* and other methods, described below.
- **search**($pattern, string, flags = 0$) scans through string looking for the first location where the regular expression pattern produces a match, and returns a corresponding match object. Return None if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.
- **match**($pattern, string, flags = 0$) If zero or more characters at the beginning of string match the regular expression pattern, return a corresponding match object. Return None if the string does not match the pattern; note that this is different from a zero-length match.
- **split**($pattern, string, maxsplit = 0, flags = 0$) splits string by the occurrences of

pattern. If capturing parentheses are used in pattern, then the text of all groups in the pattern are also returned as part of the resulting list. If maxsplit is nonzero, at most maxsplit splits occur, and the remainder of the string is returned as the final element of the list.

3.7 itertools

The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an "iterator algebra" making it possible to construct specialized tools succinctly and efficiently in pure Python.

Infinite iterators:

- **count**($start$, [$step$]) returns $start$, $start + step$, $start + 2 * step$,
- **cycle**(p) returns p_0, p_1, p_{last}, p_0, p_1,
- **repeat**($elem$[, n]) returns $elem$, $elem$, $elem$, endlessly or up to n times

Itertool functions:

- **accumulate**($iterable$[, $func$]) makes an iterator that returns accumulated sums, or accumulated results of other binary functions (specified via the optional func argument).
- **chain**($*iterables$) makes an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted.
- **groupby**($iterable$, $key = None$) makes an iterator that returns consecutive keys and groups from the iterable. The key is a function computing a key value for each element. If not specified or is None, key defaults to an identity function and returns the element unchanged. Generally, the iterable needs to already be sorted on the same key function.

3.8 `abc`

The module `abc` is introduced in `abc` in Python Standard Library Reference. See this article and this article for comprehensive explanation. Note that the usage in latter article is obsolete and may lead to errors.

3.9 `decimal` and `fractions`

The module `decimal` and `fractions` are introduced in `decimal` and in Python Standard Library Reference. Note that these two articles may be too detailed for first reading.

3.10 `argparse`

The module `argparse` is introduced in `argparse` in Python Standard Library Reference. There are tons of options for this module, but simple skeleton can be found in the first example.

3.11 `logging`

The module `logging` is introduced in `logging` in Python Standard Library Reference. Note that this article provides an introduction.

4 Assignment

Question 1 (Required) *String to Timestamp: If you get the date and time input by the costumer, like 2015-1-21 9:01:30 and the time zone information, like UTC+5:00, all of which are* `string` *objects. Please devise a program to transform these given information to timestamp. See* `Assignment/StandardLibrary-StringToTimestamp` *for details.*

Question 2 *Timing Decorator: Devise a decorator, which can affect any function and print the executing time of the function. See* `Assignment/StandardLibrary-TimingDecorator` *for details.*

Question 3 (Required) *Stack: Use the deque to realize a stack class. See* `Assignment/StandardLibrary-Stack` *for details.*

Question 4 *Stack and JSON: Serialize an object from the stack class above to a {} in JSON. See* `Assignment/StandardLibrary-StackJSON` *for details.*

Question 5 (Required) *FIFO Dict: Use the OrderedDict to realize a FIFO (first in first out) dict. When the dict is full, delete the Key added the ealiest. See* `Assignment/StandardLibrary-FIFODict` *for details.*

Question 6 (Required) *EMail Address: Write a regular expression to examine Email address. It should examine* `someone@gmail.com` *and* `bill.gates@microsoft.com`*. See* `Assignment/StandardLibrary-EMailAddress` *for details.*

Question 7 *EMail Name: Write a regular expression to examine Email address and get the name in it. For example,* `<Tom Paris> tom@voyager.org` $\Longrightarrow$ `Tom Paris`*,*

`bob@example.com` $\Longrightarrow$ `bob`. *See* `Assignment/StandardLibrary-EMailName` *for details.*

Question 8 *π: Use the* `itertools.count` *to calculate the value of π, using the formula*

$$\frac{\pi}{4} = \sum_{i=0}^{\infty} \frac{(-1)^n}{2n+1}. \tag{1}$$

Question 9 *Abstract Base Class: Implement an abstract base class* `AbstractBaseClass`*, from which* `DerivedClass` *is derived. See* `Assignment/StandardLibrary-AbstractBaseClass` *for details.*

Question 10 *Data Loader: Use* `random.shuffle` *to construct a class* `DataLoader` *for SGD. See* `Assignment/StandardLibrary-DataLoader` *for details.*

Question 11 *Numbers: Use* `decimal` *and* `fractions` *module to accomplish two high precision tasks. See* `Assignment/StandardLibrary-Numbers` *for details.*

Question 12 *Implement* `prog.py` *in the first code block in argparse in Python Standard Library Reference.*

Question 13 *Implement the first example about* `logging` *module in this article.*