The Python Standard Library

Describes the library distributed with Python

Lina Liu Nov. 16 2022

What Python's Standard Library Provide?

1. What python's standard library provide?
    1. Wide range of facilities
    2. built-in modules(written in C) to access to system functionality such as file I/O
    3. Standardized solutions for many problems that occur in everyday programming
    4. Modules designed to abstracting away platforms — neutral APIs
    5. Provides as a collection of packages
    6. Collections of packages: python package index website
        1. https://pypi.org/

The Python Standard Library

Introduction

1. What Python language core defines?
    1. “Core” of language is data types like lists, dictionary, set, tuple modules
    2. Defines the form of literals and places some constraint on their semantics
    3. Built-in functions and exceptions without need of an import statement
2. Bulk of the library introduction
    1. Modules written in C and built in to the Python interpreter.
    2. Modules written in Python and imported in source form
    3. Modules provide interfaces that are highly specific to Python
        1. Printing a stack trace
    4. Modules provide interfaces that are specific to particular operating system
        1. Access to specific hardware
    5. Modules provide interfaces that are specific to a particular application domain
        1. Like for World Wide Web
    6. Some modules are available in all versions and ports of python
    7. Some modules are only available when the underlying system support or require them
    8. Some modules are only available when a particular configuration option was chosen at the time when python was compiled and installed.

The Python Standard Library

Contents

The Python Standard Library

Introduction

1. How the manual organized?
    1. From the inside out organized
    2. First describes the built-in functions
    3. Second describe “core” data types and exceptions
    4. Third describe the modules
2. How to read the Python Standard Library?
    1. Just browse the table of contents
    2. Look for a specific function, module in the index
    3. Better to start with Built-in Functions
    4. It’s kind of dictionary lookup tool, instead of a novel.

Built-in functions

1. bin(number)
2. format(number, ‘#b’) format(number, ‘b’)
3. dir()
    1. If the object is module object, the list contains the names of the module’s attributes
    2. If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.
    3. Otherwise, the list contains the objects’s attributes’ names, the names of its class’s attributes, and recursively of the attributes of its class’s base classes.
4. enumerate(iterable, start=0)
    1. Return an enumerate object
    2. Iterable must be a sequence, an iterator, or some other object which supports iteration
    3. The __next__() method of the iterator returned by enumerate() returns a tuple containing a count and the values obtained from iterating over iterable.
5. eval(expression,globals=None, locals=None)
6. float(string)
7. hex()
8. input()
9. iter(object, sentinel, /): return an iterator object, you can define __iter__() and __next__(self) in you custom object
10. open(file, mode=‘r’, buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)
11. zip(*iterables, strict=False): iterate over several iterables in parallel, producing tuples with an item form each one.
    1. zip() in conjunction with the * operator can be used to unzip a list

```python
# So funny, NotImplementedError: dir_fd unavailable on this platform!!!!!!
import os
dir_fd = os.open('/Users/lina/code/core-python/', os.O_RDONLY)
def openner(path, flags):
    print('not implemented!')
    #return os.open(path, flags, dir_fd=dir_fd)

#with open('test_file.txt', 'a', opener=openner) as f:
with open('test_file.txt', 'a') as f:
    print('This will be written to xxxx/test_file.txt', file=f)

os.close(dir_fd)
```

Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first
`'x'`	open for exclusive creation, failing if the file already exists
`'a'`	open for writing, appending to the end of file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open for updating (reading and writing)

Built-in Types

How many built-in types are built into the interpreter?

1. Numeric Types - int, float, complex
2. Iterator Types
3. Generator Types
4. Sequence Types - list, tuple, range
    1. Common Sequence Operations
    2. Immutable sequence types
    3. Mutable sequence types
5. Text sequence type - str
6. Binary sequence types - bytes, bytearray, memoryview
7. Set types
8. Mapping types
9. Context Manager types
10. Type Annotation Types - Generic Alias, Union
11. Other Built-in Types
12. Special Attributes

Built-in Types

Int

1. int.bit_length()
2. int.to_bytes()

Iterator/Generator types

Please see these in Python Tutorial.key and class.py examples.

Development Tools

The modules help you write software

1. The module lists help you write software
    1. typing —support for type hint
    2. Pydoc — Documentation generator and online help system
    3. Python Development Mode — additional runtime checks that are too expensive to be enabled by default. Enable by -X dev cmd line
    4. doctest — Test interactive Python examples
    5. unittest — Unit testing framework
    6. unittest.mock — mock object library
    7. test — Regression tests package for python
    8. test.support — Utilities for the Python test

Automated test

10 Python modules for full stack automation and testing

1. Why we need automated test?
    1. It provides optimization, efficiency, and best practices in any business process.
    2. It will trigger faster development cycles
    3. It will trigger quicker resolution of any app issues

Built-in Type

Common Sequence Operations

1. What's the rules for comparisons between common sequence?
    1. Compared lexicographically, means compared equally.
    2. The two sequences must be the same type
    3. The two sequences must have the same length
    4. Forward and reversed iterators access values using an index.
    5. The index will continue to march forward even if the underlying sequence is mutated.
    6. The iterator terminates only when an IndexError or a StopIteration is encountered.

Built-in Types

The Notes for the Common Sequences operations

1. ‘in’ and ‘not in’ operations
    1. used for simple containment testing
    2. used for specialized sequences(such as str, bytes, and bytearray) for subsequence testing
2. Values of n less than 0 are treated as 0
3. Items in the sequence s are not copied; they are referenced multiple times
    1. See the amazing example in Built-in_types.py
4.

Built-in Types

Common Sequence Operations

Operation	Result	Notes
`x in s`	`True` if an item of *s* is equal to *x*, else `False`	(1)
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`	(1)
`s + t`	the concatenation of *s* and *t*	(6)(7)
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times	(2)(7)
`s[i]`	*i*th item of *s*, origin 0	(3)
`s[i:j]`	slice of *s* from *i* to *j*	(3)(4)
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*	(3)(5)
`len(s)`	length of *s*	
`min(s)`	smallest item of *s*	
`max(s)`	largest item of *s*	
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)	(8)
`s.count(x)`	total number of occurrences of *x* in *s*	

Bulit-in Types

Iterator Types

1. How to implement a iterator?
    1. User-defined classes to support iteration
    2. Sequences
        1. List
        2. Tuple
        3. Range
        4. Str
        5. Bytes, bytearrary, memoryview,
        6. The above always support the iteration methods.

Built-in Types

```
rev = Reverse('spam')
print(iter(rev))
```

```
def __next__(self):
    if self.index == 0:
        raise StopIteration
    self.index = self.index - 1
    return self.data[self.index]
```

Implement a container to provide iterable support

```
#if the class defines __next__(), then __iter__() can just return self
def __iter__(self):
    return self
```

1. How to create an iterator object?
    1. The objected required to support two method to form the iterator protocol
        1. Define __iter__() method
            1. Require to return the iterator itself
        2. Define __next__() method
            1. Return the next item from the iterator
            2. If no further items, raise the StopIteration exception

Built-in Types

Generator types

1. What's is Generator used for?
    1. Generator provide a convenient way to implement the iterator protocol
    2. Use the yield expression

```python
#An example shows that generators can be trivially easy to create:
def reverse(data):
    for index in range(len(data) - 1, -1, -1):
        yield data[index]
```

Built-in Types

Sequence Types — list, tuple, range

1. Immutable sequence types
    1. Tuple is immutable sequences
    2. hash() built-in method is only operation for immutable sequence types.
        1. Tuple can be used as dict keys
        2. Tuple can stored in set and frozenset instances
2. Mutable sequence types

Built-in Types

Mutable Sequence types

Operation	Result	Notes
`s[i] = x`	item *i* of *s* is replaced by *x*	
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*	
`del s[i:j]`	same as `s[i:j] = []`	
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*	(1)
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list	
`s.append(x)`	appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`)	
`s.clear()`	removes all items from *s* (same as `del s[:]`)	(5)
`s.copy()`	creates a shallow copy of *s* (same as `s[:]`)	(5)
`s.extend(t)` or `s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)	
`s *= n`	updates *s* with its contents repeated *n* times	(6)
`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)	
`s.pop()` or `s.pop(i)`	retrieves the item at *i* and also removes it from *s*	(2)
`s.remove(x)`	remove the first item from *s* where `s[i]` is equal to *x*	(3)
`s.reverse()`	reverses the items of *s* in place	(4)

Built-in Types

Mutable sequence types

1. Mutable sequence types Notes:
    1. reverse() used in reversing large sequence operates by side effect, it does not return the reversed sequence.
    2. Some Mutable containers don’t support slicing operations
        1. Set
        2. Dict
        3. But set and dict support clear() and copy() method

Built-in Types

Lists

1. Lists are mutable sequences, used to store collections of homogeneous items.
2. How to construct a list?
    1. []
    2. [],[],[]
    3. List comprehension: [x for x in iterable]
    4. Type constructor: list() or list(utterable)
    5. sorted() function
3. sort(*,key=None,reverse=False)
    1. Sorts the list in place
    2. Exceptions are not suppressed.
    3. Difference with built-in function sorted():
        1. sorted() to explicitly request a new sorted list instance.
    4.

Built-in Types

Ranges

1. Represents an immutable sequence of numbers
2. Used for looping a specific number of times in for loop.
3. The argument to the range constructor must be integers.
4. range(start, stop, step)
5. The advantage of the range over a regular list or tuple is that a range object will always take the same amount of memory.

Built-in Types

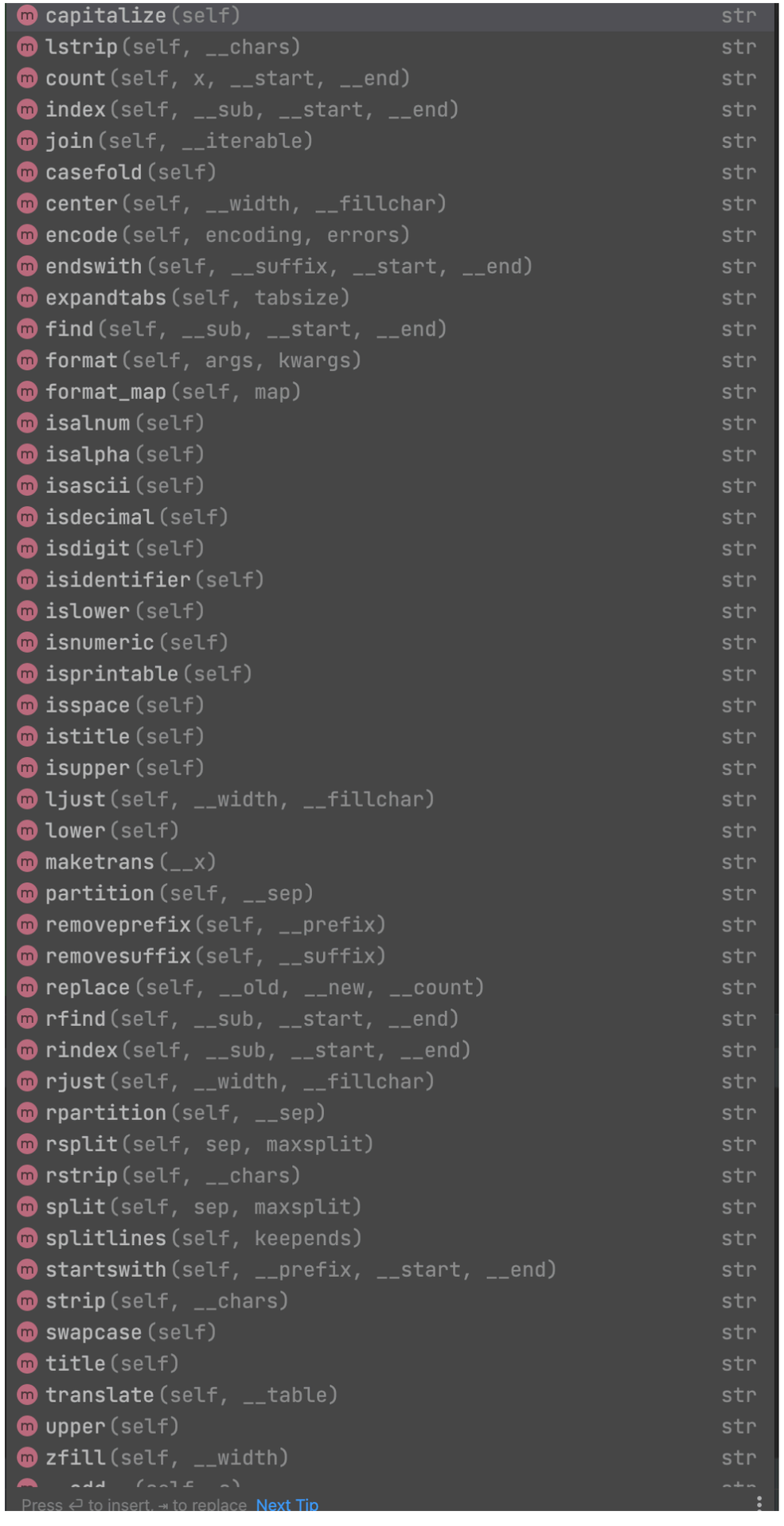
Text sequence type — str

1. String literals are written in variety of ways:
    1. Single quotes
    2. Double quotes
    3. Triple quoted:
        1. May span multiple lines
    4. String literals are part of single expression and have only whitespace between them will be converted to a single string literal.
2. Created using str constructor

Built-in Types

String methods

1. Split
    1. Return a list of the words in the string



Bulit-in Types

Set types - set, frozenset

1. A set object is an unordered collection of distinct hashable object.
2. Common uses:
    1. Membership testing
    2. Remove duplicates from a sequence
    3. Computing mathematical operations
        1. Intersection
        2. Union
        3. Difference
        4. Symmetric difference
4. Support x in set, len(set), for x in set.
5. Set do not record element position or order of insertion
6. Set do not support indexing, slicing, or other sequence like behavior.
7. Two built-in types
    1. Set
    2. frozenset

Built-in Types

Two built-in set types

1. Two built-in set types, what's the difference between the two?
    1. Set
    2. Frozenset
    3. Set type is mutable, the contents can be changed using methods like add() and remove()
    4. Set type is mutable, it has no hash value and cannot be used as either dictionary key or as an element of another set.
    5. Frozenset is immutable and hashable it's contents cannot be altered after it is created.
    6. Frozenset can be used as a dictionary key or as an element of another set.

Built-in Types

Frozenset

1. What is frozenset?
    1. It frozen, it means immutable…
    2. The elements of frozenset are taken from iterable.
    3. The elements of frozenset must be hashable.
    4. To represent sets of sets, the inner sets must be frozenset object.

Built-in Types

How to create a set or frozenset

1. How to create a set or frozenset?
    1. use a comma-separted list of elements within braces
        1. {‘rainny’, ‘day’}
    2. use a set comprehension
        1. {c for c in ‘magic word abra-ca-da-bra’ if c not in ‘abc’}
3. type constructor
    1. set()
    2. set(‘your name’)
    3. set([‘a’, ‘b’])

Built-in Type

set examples

See in built-in type.py

Built-in Types —Mapping Types

mutable objects maps hashable values to arbitrary object

1. Mapping object
    1. maps hashable values to arbitrary objects
    2. are mutable objects
    3. only one standard mapping type
        1. dictionary
        2. dictionary key are arbitrary values
        3. dictionary value are not hashable.
            1. list
            2. dictionaries
            3. set

Built-in Types

Dictionary

1. How to create a dictionary?
    1. comma-separated list of key: value pairs within braces
    2. dict comprehension
    3. constructor dict()

Dictionary view objects

dict.keys() dict.values() and dict.items()

1. what are dictionary view objects?
    1. the objects returned by:
        1. dict.keys()
            1. key views are set-like since their entries are unique and hashable
        2. dict.values()
            1. if all values are hashable, (key, value) pairs are unique and hashable
        3. dict.items()
            1. set-like
        4. the returned value above are dictionary view objects
2. object view provide a dynamic view on the dictionary's entries
    1. when dictionary changes, the view reflects these changes
3. dictionary views can be iterated over to yield their respective data
    1. len(dict.keys())
    2. iter(dict.keys())
    3. x in dict.keys()
    4. reversed(dict.keys())

Other built-in Types

Modules

1. Modules
    1. the only operation on a module is attribute access: m.name
        1. import foo statement requires an definition for a module named foo somewhere
    2. a special attribute of every module is __dict__
        1. __dict__ conatins module’s symbol table
3. Modules built into the interpreter are written like this:
    1. <module ‘sys’(built-in)>
    2. if loaded from a file, they are written:
        1. <module ‘os’ from ‘/usr/local/lib/pythonX.Y/os.pyc’>

Bulit-in Types

Special Attributes

1. special attributes may not reported by the dir() built-in functions
    1. object.__dict__
    2. instance.__class__
    3. class.__bases__
    4. definition.__name__
    5. definition.__qualname__
    6. class.__mro__
    7. class.mro()
    8. class.__subclasses__()

Built-In Types

Integer string conversion length limitation

1. why need length limitations?
    1. CPython has a global limit for converting between int and str to mitigate denial of service attacks.
2. who has length limitations?
    1. decimal
    2. non-power-of-two numbers
3. who doesn’t has length limitations?
    1. Hexadecimal, octal, binary conversion are unlimited
4. How to set limitations?
    1. see examples ….
    2. there is no sys.set_int_max_str_digits(4300) #no this function anymore
    3. couldn’t set by user anymore? HHHHHHHHHHHHH

Built-in Exceptions

BaseException

1. BaseException
    1. all exceptions must be instances of a class that derived from BaseExeption
    2. try statement with an except clause
        1. can handles any exception classes derived the mentioned class
3. user code can raise built-in exceptions
    1. using keyword raise

Exception Hierarchy

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ExceptionGroup [BaseExceptionGroup]
      ├── ImportError
      │    └── ModuleNotFoundError
      ├── LookupError
      │    ├── IndexError
      │    └── KeyError
      ├── MemoryError
      ├── NameError
      │    └── UnboundLocalError
      ├── OSError
      │    ├── BlockingIOError
      │    ├── ChildProcessError
      │    ├── ConnectionError
      │    │    ├── BrokenPipeError
      │    │    ├── ConnectionAbortedError
      │    │    ├── ConnectionRefusedError
      │    │    └── ConnectionResetError
      │    ├── FileExistsError
      │    ├── FileNotFoundError
      │    ├── InterruptedError
      │    ├── IsADirectoryError
      │    ├── NotADirectoryError
      │    ├── PermissionError
      │    ├── ProcessLookupError
      │    └── TimeoutError
      ├── ReferenceError
      ├── RuntimeError
      │    ├── NotImplementedError
      │    └── RecursionError
      ├── StopAsyncIteration
      ├── StopIteration
      ├── SyntaxError
      │    └── IndentationError
      │         └── TabError
      ├── SystemError
      ├── TypeError
      ├── ValueError
      │    └── UnicodeError
      │         ├── UnicodeDecodeError
      │         ├── UnicodeEncodeError
      │         └── UnicodeTranslateError
      └── Warning
           ├── BytesWarning
           ├── DeprecationWarning
           ├── EncodingWarning
           ├── FutureWarning
           ├── ImportWarning
           ├── PendingDeprecationWarning
           ├── ResourceWarning
           ├── RuntimeWarning
           ├── SyntaxWarning
           ├── UnicodeWarning
           └── UserWarning
```

Text Processing Services

how many Text Processing Services provided?

1. how many Text Processing Services provided?
    1. string - common string operations
    2. re - regular expression operations
    3. difflib — helpers for computing deltas
    4. textwrap — text wrapping and filling
    5. unicodedata — unicode database
    6. stringprep — internet string preparation
    7. readline — GNU readline interface
    8. rlcompleter —Completion function for GNU readline

String - Common string operations

str.format and Template string

1. Common String operations
    1. str.format
    2. Template string
        1. the string module provided a Template class
            1. from string import Template
            2. you can derive subclasses of Template to customize
                1. the placeholder syntax
                2. delimiter character
                3. entire regular expression used to parse template strings.

String - Common string operations

Template class

1. Why template class was added to string module?
    1. an alternative to the built-in substitution options(mainly to %) for creating complex string-based templates and handling them in a user-friendly way.
    2. use re-regular expression to match a general pattern of valid template strings.

Template class

template class implementation

1. template class implementation
    1. use regular expressions to match:
        1. a pattern of valid template strings
        2. placeholder
2. consists of two parts:
    1. $ symbol
    2. a valid python identifier
    3. like $name, $age, $ID

Template class

How to use template class?

1. How to use template class?
    1. import template from string
    2. create a valid template string
    3. instantiate template using the template string as an argument
    4. Perform the substitution using a substitution method

re - regular expression operations

what are regular expressions?

1. what are regular expressions?
    1. provide matching operations similar to those found in perl
    2. re use backslash(‘\’) to indicate special forms to be used without invoking their special meanings
    3. re module
    4. matching patterns
    5. matching methods

re - regular expression operations

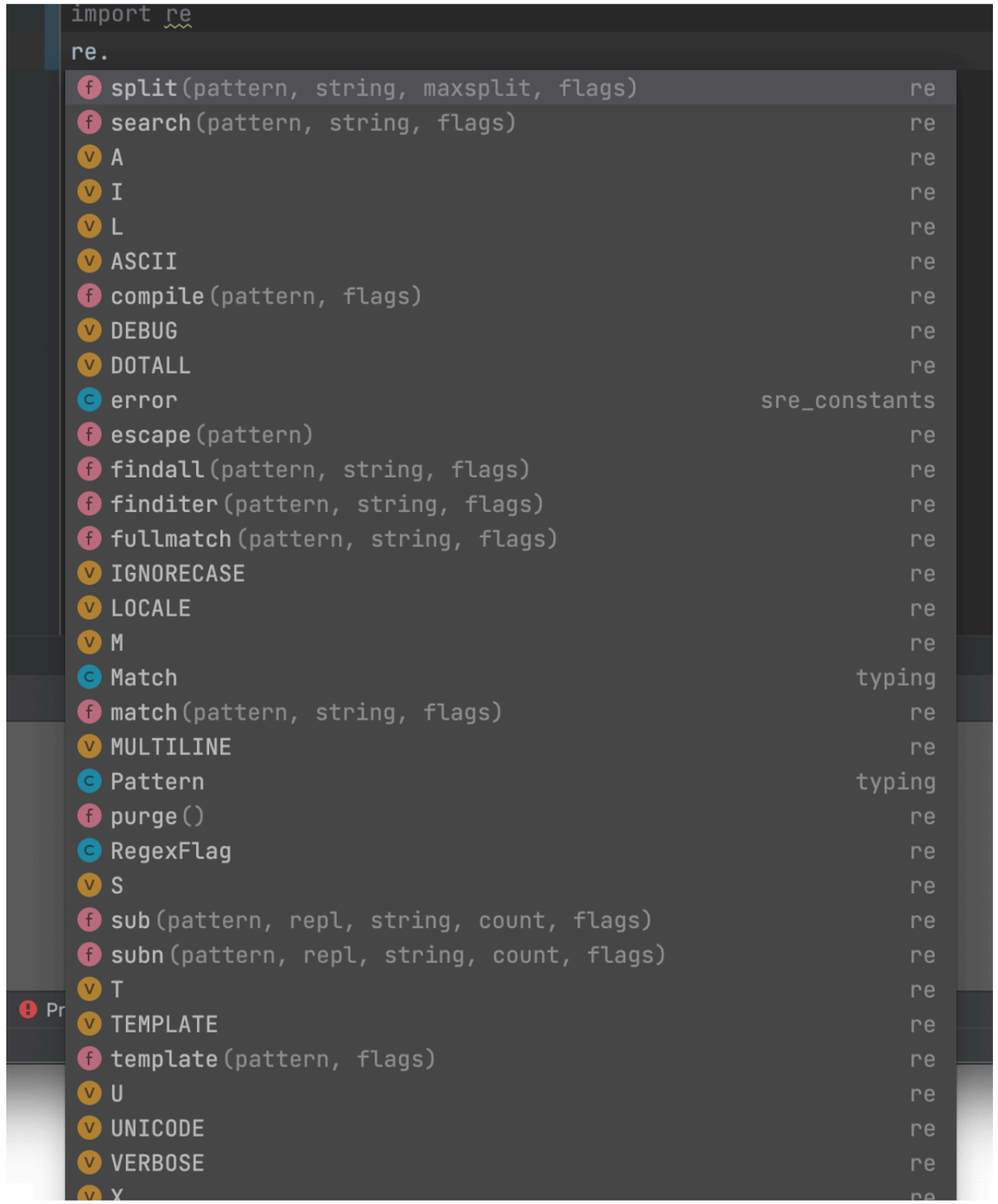
matching characters

`*`	Matches the preceding element *zero* or more times. For example, `ab*c` matches "ac", "abc", "abbb c", etc. `[xyz]*` matches "", "x", "y", "z", "zx", "zyx", "xyzzy", and so on. `(ab)*` matches "", "ab", "ab ab", "ababab", and so on.
`+`	Matches the preceding element *one* or more times. For example, `ab+c` matches "abc", "abbc", "abb bc", and so on, but not "ac".
`?`	Matches the preceding element zero or one time. For example, `ab?c` matches only "ac" or "abc".
`|`	The choice (also known as alternation or set union) operator matches either the expression before o r the expression after this operator. For example, `abc|def` can match either "abc" or "def".
`.`	Matches any single character (many applications exclude newlines, and exactly which characters ar e considered newlines is flavor-, character-encoding-, and platform-specific, but it is safe to assume that the line feed character is included). Within POSIX bracket expressions, the dot character match es a literal dot. For example, `a.c` matches "abc", etc., but `[a.c]` matches only "a", ".", or "c".
`^`	Matches the *starting* position in the string, like the `startsWith()` function. In line-based tools, it ma tches the starting position of any line.
`?`	Matches the *ending* position of the string or the position just before a string-ending newline, like the `endsWith()` function. In line-based tools, it matches the ending position of any line.

re - regular expression operations

Matching method

1. matching method
    1. re.split()
    2. re.search()
    3. re.compile()
    4. re.escape()
    5. re.findall()
    6. re.finditer()
    7. re.fullmatch()
    8. re.match()
    9. re.purge()
    10. re.sub()
    11. re.subn()
    12. re.template()



Python Regular Expressions

SPECIAL CHARACTERS

`^` | Matches the expression to its right at the start of a string. It matches every such instance before each `\n` in the string.

`$` | Matches the expression to its left at the end of a string. It matches every such instance before each `\n` in the string.

`.` | Matches any character except line terminators like `\n`.

`\` | Escapes special characters or denotes character classes.

`A|B` | Matches expression `A` or `B`. If A is matched first, B is left untried.

`+` | Greedily matches the expression to its left 1 or more times.

`*` | Greedily matches the expression to its left 0 or more times.

`?` | Greedily matches the expression to its left 0 or 1 times. But if `?` is added to qualifiers (`+`, `*`, and `?` itself) it will perform matches in a non-greedy manner.

`{m}` | Matches the expression to its left `m` times, and not less.

`{m,n}` | Matches the expression to its left `m` to `n` times, and not less.

`{m,n}?` | Matches the expression to its left `m` times, and ignores `n`. See `?` above.

CHARACTER CLASSES [A.K.A. SPECIAL SEQUENCES]

`\w` | Matches alphanumeric characters, which means `a-z`, `A-Z`, and `0-9`. It also matches the underscore, `_`.

`\d` | Matches digits, which means `0-9`.

`\D` | Matches any non-digits.

`\s` | Matches whitespace characters, which include the `\t`, `\n`, `\r`, and space characters.

`\S` | Matches non-whitespace characters.

`\b` | Matches the boundary (or empty string) at the start and end of a word, that is, between `\w` and `\W`.

`\B` | Matches where `\b` does not, that is, the boundary of `\w` characters.

`\A` | Matches the expression to its right at the absolute start of a string whether in single or multi-line mode.

`\Z` | Matches the expression to its left at the absolute end of a string whether in single or multi-line mode.

SETS

`[ ]` | Contains a set of characters to match.

`[amk]` | Matches either `a`, `m`, or `k`. It does not match `amk`.

`[a-z]` | Matches any alphabet from `a` to `z`.

`[a\-z]` | Matches `a`, `-`, or `z`. It matches `-` because `\` escapes it.

`[a-]` | Matches `a` or `-`, because `-` is not being used to indicate a series of characters.

`[-a]` | As above, matches `a` or `-`.

`[a-z0-9]` | Matches characters from `a` to `z` and also from `0` to `9`.

`[(+*)]` | Special characters become literal inside a set, so this matches `(`, `+`, `*`, and `)`.

`[^ab5]` | Adding `^` excludes any character in the set. Here, it matches characters that are not `a`, `b`, or `5`.

GROUPS

`( )` | Matches the expression inside the parentheses and groups it.

`(?)` | Inside parentheses like this, `?` acts as an extension notation. Its meaning depends on the character immediately to its right.

`(?PAB)` | Matches the expression `AB`, and it can be accessed with the group name.

`(?aiLmsux)` | Here, `a`, `i`, `L`, `m`, `s`, `u`, and `x` are flags:
- `a` — Matches ASCII only
- `i` — Ignore case
- `L` — Locale dependent
- `m` — Multi-line
- `s` — Matches all
- `u` — Matches unicode
- `x` — Verbose

`(?:A)` | Matches the expression as represented by `A`, but unlike `(?PAB)`, it cannot be retrieved afterwards.

`(?#...)` | A comment. Contents are for us to read, not for matching.

`A(?=B)` | Lookahead assertion. This matches the expression `A` only if it is followed by `B`.

`A(?!B)` | Negative lookahead assertion. This matches the expression `A` only if it is not followed by `B`.

`(?<=B)A` | Positive lookbehind assertion. This matches the expression `A` only if `B` is immediately to its left. This can only matched fixed length expressions.

`(?<!B)A` | Negative lookbehind assertion. This matches the expression `A` only if `B` is not immediately to its left. This can only matched fixed length expressions.

`(?P=name)` | Matches the expression matched by an earlier group named "name".

`(...)\1` | The number `1` corresponds to the first group to be matched. If we want to match more instances of the same expression, simply use its number instead of writing out the whole expression again. We can use from `1` up to `99` such groups and their corresponding numbers.

POPULAR PYTHON RE MODULE FUNCTIONS

`re.findall(A, B)` | Matches all instances of an expression `A` in a string `B` and returns them in a list.

`re.search(A, B)` | Matches the first instance of an expression `A` in a string `B`, and returns it as a re match object.

`re.split(A, B)` | Split a string `B` into a list using the delimiter `A`.

`re.sub(A, B, C)` | Replace `A` with `B` in the string `C`.

re - regular expression operations

Regular expression objects

1. How to generate a regular expression object?
    1. use re.compile()
    2. Compile a regular expression pattern into a regular expression object
    3. the expression is some kind of pattern
2. what is regular expression used for?
    1. pattern.search()
    2. match
    3. split
    4. find
    5. flags
    6. groups