1. Pythonic Thinking

3. Know the difference between bytes, str, and unicode

- python 3:
  - bytes - 8-bit values
  - str - unicode characters
  - cannot be used together with operators, etc without conversion
- python 2:
  - str - 8 - bit
  - unicode - unicode characters
  - can interoperate if str only contains 7-bit ASCII
- use helper functions
- use 'rb' and 'wb' for file access in binary mode

4. Write Helper Functions Instead of Complex Expressions

- python's syntax makes it all too easy to write single-line expressions that are overly complicated and difficult to read
- move complex expressions into helper functions, especially if you need to use the same logic repeatedly
- the if/ else expression provides a more readable alternative to using Boolean operators like or and and in expressions

5. Know How to Slice Sequences

- simplest with list, str, bytes
- can extend any type using **getitem** and **setitem**
- basic form `list[start:end]` (inclusive, exclusive)
- when slicing from zero, leave out 0 (e.g. `a[:5] == a[0:5]`)
- when slicing to the end of a list, leave out the end index (e.g. `a[5:] == a[5:len(a)]`)
- using negative numbers gives an offset relative to the end of a list

```
a[:] # [' a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
a[: 5] # [' a', 'b', 'c', 'd', 'e']
a[:-1] # [' a', 'b', 'c', 'd', 'e', 'f', 'g']
a[ 4:] # [' e', 'f', 'g', 'h']
a[-3:] # [' f', 'g', 'h']
a[ 2: 5] # [' c', 'd', 'e']
```

```
a[ 2:-1] # [' c', 'd', 'e', 'f', 'g']
a[-3:-1] # [' f', 'g']
```

- slicing handles out of bounds start and end indexes whereas element access does not
- avoid being verbose: Don’t supply 0 for the start index or the length of the sequence for the end index
- slicing is forgiving of start or end indexes that are out of bounds, making it easy to express slices on the front or back boundaries of a sequence (like `a[:20]` or `a[-20:]`)
- assigning to a list slice will replace that range in the original sequence with what’s referenced even if their lengths are different

6: Avoid Using start, end, and stride in a Single Slice

- stride of slice `list[start:end:stride]` (i.e. every nth item)
- negative stride works for byte strings and ASCII, but it will break for Unicode encoded as UTF-8
- specifying start, end, and stride in a slice can be extremely confusing
- prefer using positive stride values in slices without start or end indexes
- avoid negative stride values if possible
- avoid using start, end, and stride together in a single slice
- if you need all three parameters, consider doing two assignments (one to slice, another to stride) or using islice from the itertools built-in module

7. Use List Comprehensions Instead of map and filter

- list comprehension - syntax for deriving a list from an iterable
- clearer than map builtin with a lambda
- can use filter with map to achieve same simpler syntax as list comprehensions
- list comprehensions are clearer than the map and filter built-in functions because they don’t require extra lambda expressions
- list comprehensions allow you to easily skip items from the input list, a behavior map doesn’t support without help from filter
- dictionaries and sets also support comprehension expressions.

8. Avoid More Than Two Expressions in List Comprehensions

- support multiple levels of looping

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [x for row n matrix for x in row]
```

```
# basic looping
flat = []
for sublist1 in three_dim:
    for sublist2 in sublist1:
        flat.extend(sublist2)
```

- also support multiple ifs

```
matrix = [[ 1, 2, 3], [4, 5, 6], [7, 8, 9]]
filtered = [[ x for x in row if x % 3 = = 0] for row in matrix if sum( row) > = 10
```

- list comprehensions support multiple levels of loops and multiple conditions per loop level
- list comprehensions with more than two expressions are very difficult to read and should be avoided

9. Consider Generator Expressions for Large Comprehensions

- list comprehensions may create a whole new list for each item in the input sequence which is not fine for large inputs (takes memory)
- generator expressions - generalization of list comprehensions and generators that don't materialize the whole output sequence when run, instead generate one element at a time
- syntax is like list comprehensions but between `()`

```
it = (len(x) for x in open('/tmp.txt'))
```

- generator expressions can be composed together
- list comprehensions can cause problems for large inputs by using too much memory
- generator expressions avoid memory issues by producing outputs one at a time as an iterator
- generator expressions can be composed by passing the iterator from one generator expression into the for subexpression of another
- generator expressions execute very quickly when chained together

10. Prefer enumerate Over range

- enumerate provides concise syntax for looping over an iterator and getting the index of each item from the iterator as you go
- prefer enumerate instead of looping over a range and indexing into a sequence

- you can supply a second parameter to enumerate to specify the number from which to begin counting (zero is the default)

11. Use zip to Process Iterators in Parallel

- in Python 3, zip wraps two or more iterators with a lazy generator which yields tuples that contain the next value from each iterator
- the zip built-in function can be used to iterate over multiple iterators in parallel
- in Python 3, zip is a lazy generator that produces tuples
- in Python 2, zip returns the full result as a list of tuples
- zip truncates its output silently if you supply it with iterators of different lengths
- the zip_longest function from the itertools built-in module lets you iterate over multiple iterators in parallel regardless of their lengths

12. Avoid else Blocks After for and while Loops

- can put an else block after a loop's repeated interior block

```
for i in range(3):
    print('Loop %d' % i)
else:
    print('Else block!')
```

- else block here does not happen with a call to break
- executes immediately when for statement is passed an empty list
- also runs with while loops when the while loop is initially False
- python has special syntax that allows else blocks to immediately follow for and while loop interior blocks
- the else block after a loop only runs if the loop body did not encounter a break statement
- avoid using else blocks after loops because their behavior isn't intuitive and can be confusing

13. Take Advantage of Each Block in try/except/else/finally

- each block serves a different unique function
- finally - for reliably running clean-up code before an exception propagates
- else - else in try/except/else blocks run when try block does not raise an exception
    - minimizes amount of code in try block
    - improves readability
- the try/ finally compound statement lets you run cleanup code regardless of whether exceptions were raised in the try block
- the else block helps you minimize the amount of code in try blocks and visually distinguish the success case from the try/except blocks

- an else block can be used to perform additional actions after a successful try block but before common cleanup in a finally block

2. Functions

14. Prefer Exceptions to Returning None

- none commonly used to avoid throwing exceptions but is ultimately unclear across APIs
- functions that return None to indicate special meaning are error prone because None and other values (e.g., zero, the empty string) all evaluate to False in conditional expressions
- raise exceptions to indicate special situations instead of returning None
- expect the calling code to handle exceptions properly when they're documented

15. Know How Closures Interact with Variable Scope

- python supports closures
    - closures - functions that refer to variables from the scope in which they were defined
- functions are first-class objects in Python, meaning you can refer to them directly, assign them to variables, pass them as arguments to other functions, compare them in expressions and if statements, etc.
- python has specific rules for comparing tuples
    - first compares items in index zero, then index one, then index two, and so on
- python interpreter traverses the scope to resolve a variable reference in the following order
    i. current function's scope
    ii. Any enclosing scopes (like other containing functions)
    iii. scope of the module that contains the code (also called the global scope)
    iv. built-in scope (that contains functions like len and str)
- `nonlocal` keyword can be used for getting data from outside of a closure in Python 3

```
def sort_priority3(numbers, group):
    found = False
    def helper(x):
        nonlocal found
        if x in group:
            found = True
            return (0, x)
        return (1, x)
    numbers.sort(key = helper)
    return found
```

- python 2 does not support nonlocal so need to use a work-around
- closure functions can refer to variables from any of the scopes in which they were defined
- by default, closures can't affect enclosing scopes by assigning variables

- in Python 3, use the nonlocal statement to indicate when a closure can modify a variable in its enclosing scopes
- in Python 2, use a mutable value (like a single-item list) to work around the lack of the nonlocal statement
- avoid using nonlocal statements for anything beyond simple functions

16. Consider Generators Instead of Returning Lists

- using generators can be clearer than the alternative of returning lists of accumulated results
- the iterator returned by a generator produces the set of values passed to yield expressions within the generator function's body
- generators can produce a sequence of outputs for arbitrarily large inputs because their working memory doesn't include all inputs and outputs

17. Be Defensive When Iterating Over Arguments

- convert potentially passed generators using list() ctor
- better to use iterable objects than lambdas (define **iter** as a generator)
- beware of functions that iterate over input arguments multiple times
- if these arguments are iterators, you may see strange behavior and missing values
- python's iterator protocol defines how containers and iterators interact with the iter and next built-in functions, for loops, and related expressions
- can easily define iterable container type by implementing the **iter** method as a generator
- can detect that a value is an iterator (instead of a container) if calling iter on it twice produces the same result, which can then be progressed with the next built-in function

18. Reduce Visual Noise with Variable Positional Arguments

- optional positional arguments (often called star args (*args)) can remove visual noise

```
def log(message, *values):
    if not values:
        print(message)
     else:
        values_str = ', '.join(str(x) for x in values)
        print('%s: %s' % (message, values_str))
```

- call with *arg when inputting at call site
- the variable arguments are always turned into a tuple before they are passed to your function
    - if the caller of the function uses the * operator on a generator, it will be iterated until it's exhausted
    - resulting tuple will include every value from the generator, which could consume a lot of memory and cause program to crash

- functions can accept a variable number of positional arguments by using *args in the def statement
- can use the items from a sequence as the positional arguments for a function with the * operator
- using the * operator with a generator may cause your program to run out of memory and crash
- adding new positional parameters to functions that accept *args can introduce hard-to-find bugs

19. Provide Optional Behavior with Keyword Arguments

- all positional arguments can also be passed by keyword
- best practice is to always specify optional arguments using the keyword names and never pass them as positional arguments
- nOTE: backwards compatibility using optional keyword arguments is important for functions that accept *args
    - an even better practice is to use keyword-only arguments
- function arguments can be specified by position or by keyword
- keywords make it clear what the purpose of each argument is when it would be confusing with only positional arguments
- keyword arguments with default values make it easy to add new behaviors to a function, especially when the function has existing callers
- optional keyword arguments should always be passed by keyword instead of by position

20. Use None and Docstrings to Specify Dynamic Default Arguments

- default arguments are only evaluated once: during function definition at module load time
- this can cause odd behaviors for dynamic values (like {} or [])
- use None as the default value for keyword arguments that have a dynamic value
- document the actual default behavior in the function's docstring

21. Enforce Clarity with Keyword-Only Arguments

- in Python 3, you can demand clarity by defining your functions with keyword-only arguments
    - these arguments can only be supplied by keyword, never by position
- the * symbol in the argument list indicates the end of positional arguments and the beginning of keyword-only arguments

```
def safe_division_c( number, divisor, *, ignore_overflow = False, ignore_zero_divi
```

- need to use **kwargs in Python 2

- keyword arguments make the intention of a function call more clear
- use keyword-only arguments to force callers to supply keyword arguments for potentially confusing functions, especially those that accept multiple Boolean flags
- python 3 supports explicit syntax for keyword-only arguments in functions
- python 2 can emulate keyword-only arguments for functions by using ** kwargs and manually raising TypeError exceptions

3. Classes and Inheritance

22. Prefer Helper Classes Over Bookkeeping with Dictionaries and Tuples

- avoid dictionaries that contain dictionaries
    - as soon as bookkeeping gets complicated, break a structure into helper classes
- collections.namedtuple is useful but extending it indefinitely makes becomes an anti-pattern
- limitations of namedtuple
    - can't specify default argument values for namedtuple classes
    - become unwieldy when data may have many optional properties
    - if using more than a handful of attributes, defining your own class may be a better choice
    - attribute values of namedtuple instances are still accessible using numerical indexes and iteration
    - especially in externalized APIs, this can lead to unintentional usage that makes it harder to move to a real class later
    - if not in control of all of the usage of namedtuple instances, better to define your own class
- avoid making dictionaries with values that are other dictionaries or long tuples
- use namedtuple for lightweight, immutable data containers before you need the flexibility of a full class
- move your bookkeeping code to use multiple helper classes when your internal state dictionaries get complicated

23. Accept Functions for Simple Interfaces Instead of Classes

- instead of defining and instantiating classes, functions are often all you need for simple interfaces between components in Python
- references to functions and methods in Python are first class, meaning they can be used in expressions like any other type
- the **call** special method enables instances of a class to be called like plain Python functions

- when you need a function to maintain state, consider defining a class that provides the **call** method instead of defining a stateful closure

24. Use @classmethod Polymorphism to Construct Objects Generically

- polymorphism is a way for multiple classes in a hierarchy to implement their own unique versions of a method

- allows many classes to fulfill the same interface or abstract base class while providing different functionality

- python only supports a single constructor per class, the **init** method

- use @classmethod to define alternative constructors for your classes

- use class method polymorphism to provide generic ways to build and connect concrete subclasses

- tODO: show examples here

25. Initialize Parent Classes with super

- if your class is affected by multiple inheritance (something to avoid in general calling the superclasses' **init** methods directly can lead to unpredictable behavior
- one problem is that the **init** call order isn't specified across all subclasses
- another problem occurs with diamond inheritance
  - diamond inheritance happens when a subclass inherits from two separate classes that have the same superclass somewhere in the hierarchy
  - diamond inheritance causes the common superclass's **init** method to run multiple times, causing unexpected behavior
- the super built-in function works well, but it still has two noticeable problems in Python 2:
  - its syntax is a bit verbose - have to specify the class you're in, the self object, the method name (usually **init**), and all the arguments
  - have to specify the current class by name in the call to super
  - if the class's name is changed also need to update every call to super
- python 3 fixes these issues by making calls to super with no arguments equivalent to calling super with **class** and self specified
- in Python 3, always use super because it's clear, concise, and always does the right thing
- python's standard method resolution order (MRO) solves the problems of superclass initialization order and diamond inheritance

- always use the super built-in function to initialize parent classes

26. Use Multiple Inheritance Only for Mix-in Utility Classes

- avoid using multiple inheritance if mix-in classes can achieve the same outcome
- use pluggable behaviors at the instance level to provide per-class customization when mix-in classes may require it
- compose mix-ins to create complex functionality from simple behaviors

27. Prefer Public Attributes Over Private Ones

- private attributes aren't rigorously enforced by the Python compiler
- plan from the beginning to allow subclasses to do more with your internal APIs and attributes instead of locking them out by default
- use documentation of protected fields to guide subclasses instead of trying to force access control with private attributes
- only consider using private attributes to avoid naming conflicts with subclasses that are out of your control

28. Inherit from collections.abc for Custom Container Types

- reports errors when a method is not overridden that should be implemented
- inherit directly from Python's container types (like list or dict) for simple use cases
- beware of the large number of methods required to implement custom container types correctly
- have your custom container types inherit from the interfaces defined in collections.abc to ensure that your classes match required interfaces and behaviors

4. Metaclasses and Attributes

- allow interception of class statement and provide special behavior each time a class is defined
- supplies built-in features for dynamically customizing attribute access
- dynamic attributes enable you to override objects and cause unexpected side effects
- metaclasses can create extremely bizarre behaviors that are unapproachable to newcomers
- important to follow the rule of least surprise and only use these mechanisms to implement well-understood idioms

29. Use Plain Attributes Instead of Get and Set Methods

- define new class interfaces using simple public attributes, and avoid set and get methods

- use @property to define special behavior when attributes are accessed on your objects, if necessary
- follow the rule of least surprise and avoid weird side effects in your @property methods
- ensure that @property methods are fast; do slow or complex work using normal methods

30. Consider @property Instead of Refactoring Attributes

- use @property to give existing instance attributes new functionality
- make incremental progress toward better data models by using @property
- consider refactoring a class and all call sites when you find yourself using @property too heavily

31. Use Descriptors for Reusable @property Methods

- reuse the behavior and validation of @property methods by defining your own descriptor classes
- use WeakKeyDictionary to ensure that your descriptor classes don't cause memory leaks
- don't get bogged down trying to understand exactly how **getattribute** uses the descriptor protocol for getting and setting attributes.

32. Use getattr , getattribute , and setattr for Lazy Attributes

- use **getattr** and **setattr** to lazily load and save attributes for an object
- understand that **getattr** only gets called once when accessing a missing attribute, whereas **getattribute** gets called every time an attribute is accessed
- avoid infinite recursion in **getattribute** and **setattr** by using methods from super() (i.e., the object class) to access instance attributes directly.

33. Validate Subclasses with Metaclasses

- use metaclasses to ensure that subclasses are well formed at the time they are defined, before objects of their type are constructed
- metaclasses have slightly different syntax in Python 2 vs
- python 3. The **new** method of metaclasses is run after the class statement's entire body has been processed.

34. Register Class Existence with Metaclasses

- class registration is a helpful pattern for building modular Python programs
- metaclasses let you run registration code automatically each time your base class is subclassed in a program
- using metaclasses for class registration avoids errors by ensuring that you never miss a registration call.

35. Annotate Class Attributes with Metaclasses

- metaclasses enable you to modify a class's attributes before the class is fully defined
- descriptors and metaclasses make a powerful combination for declarative behavior and runtime introspection
- you can avoid both memory leaks and the weakref module by using metaclasses along with descriptors.

5. Concurrency and Parallelism

36. Use subprocess to Manage Child Processes

- use the subprocess module to run child processes and manage their input and output streams
- child processes run in parallel with the Python interpreter, enabling you to maximize your CPU usage
- use the timeout parameter with communicate to avoid deadlocks and hanging child processes.

37. Use Threads for Blocking I/O, Avoid for Parallelism

- python threads can't run bytecode in parallel on multiple CPU cores because of the global interpreter lock (GIL)
- python threads are still useful despite the GIL because they provide an easy way to do multiple things at seemingly the same time
- use Python threads to make multiple system calls in parallel
- this allows you to do blocking I/ O at the same time as computation

38. Use Lock to Prevent Data Races in Threads

- even though Python has a global interpreter lock, you're still responsible for protecting against data races between the threads in your programs
- your programs will corrupt their data structures if you allow multiple threads to modify the same objects without locks
- the Lock class in the threading built-in module is Python's standard mutual exclusion lock implementation

39. Use Queue to Coordinate Work Between Threads

- pipelines are a great way to organize sequences of work that run concurrently using multiple Python threads

- be aware of the many problems in building concurrent pipelines: busy waiting, stopping workers, and memory explosion
- the Queue class has all of the facilities you need to build robust pipelines: blocking operations, buffer sizes, and joining

40. Consider Coroutines to Run Many Functions Concurrently

- coroutines provide an efficient way to run tens of thousands of functions seemingly at the same time
- within a generator, the value of the yield expression will be whatever value was passed to the generator's send method from the exterior code
- coroutines give you a powerful tool for separating the core logic of your program from its interaction with the surrounding environment
- python 2 doesn't support yield from or returning values from generators

41. Consider concurrent.futures for True Parallelism

- moving CPU bottlenecks to C-extension modules can be an effective way to improve performance while maximizing your investment in Python code
- however, the cost of doing so is high and may introduce bugs
- the multiprocessing module provides powerful tools that can parallelize certain types of Python computation with minimal effort
- the power of multiprocessing is best accessed through the concurrent.futures built-in module and its simple ProcessPoolExecutor class
- the advanced parts of the multiprocessing module should be avoided because they are so complex

6. Built-in Modules

42. Define Function Decorators with functools.wraps

- decorators are Python syntax for allowing one function to modify another function at runtime
- using decorators can cause strange behaviors in tools that do introspection, such as debuggers
- use the wraps decorator from the functools built-in module when you define your own decorators to avoid any issues

43. Consider contextlib and with Statements for Reusable try/finally Behavior

- the with statement allows you to reuse logic from try/ finally blocks and reduce visual noise

- the contextlib built-in module provides a contextmanager decorator that makes it easy to use your own functions in with statements
- the value yielded by context managers is supplied to the as part of the with statement
- it's useful for letting your code directly access the cause of the special context

44. Make pickle Reliable with copyreg

- the pickle built-in module is only useful for serializing and deserializing objects between trusted programs
- the pickle module may break down when used for more than trivial use cases
- use the copyreg built-in module with pickle to add missing attribute values, allow versioning of classes, and provide stable import paths

45. Use datetime Instead of time for Local Clocks

- avoid using the time module for translating between different time zones
- use the datetime built-in module along with the pytz module to reliably convert between times in different time zones
- always represent time in UTC and do conversions to local time as the final step before presentation.

46. Use Built-in Algorithms and Data Structures

- use Python's built-in modules for algorithms and data structures
- don't reimplement this functionality yourself, it's hard to get right

47. Use decimal When Precision Is Paramount

- python has built-in types and classes in modules that can represent practically every type of numerical value
- the Decimal class is ideal for situations that require high precision and exact rounding behavior, such as computations of monetary values

48. Know Where to Find Community-Built Modules

- pip is installed by default in Python 3.4 and above; you must install it yourself for older versions

- the majority of PyPI modules are free and open source software

7. Collaboration

49. Write Docstrings for Every Function, Class, and Module

- write documentation for every module, class, and function using docstrings
- keep them up to date as your code changes
- for modules: Introduce the contents of the module and any important classes or functions all users should know about
- for classes: Document behavior, important attributes, and subclass behavior in the docstring following the class statement
- for functions and methods: Document every argument, returned value, raised exception, and other behaviors in the docstring following the def statement

50. Use Packages to Organize Modules and Provide Stable APIs

- packages in Python are modules that contain other modules
- packages allow you to organize your code into separate, non-conflicting namespaces with unique absolute module names
- simple packages are defined by adding an **init**
- py file to a directory that contains other source files
- these files become the child modules of the directory's package
- package directories may also contain other packages
- you can provide an explicit API for a module by listing its publicly visible names in its **all** spec ial attribute
- you can hide a package's internal implementation by only importing public names in the package's **init**
- py file or by naming internal-only members with a leading underscore
- when collaborating within a single team or on a single codebase, using **all** for explicit APIs is probably unnecessary

51. Define a Root Exception to Insulate Callers from APIs

- defining root exceptions for your modules allows API consumers to insulate themselves from your API
- catching root exceptions can help you find bugs in code that consumes an API
- catching the Python Exception base class can help you find bugs in API implementations
- intermediate root exceptions let you add more specific types of exceptions in the future without breaking your API consumers

52. Know How to Break Circular Dependencies

- circular dependencies happen when two modules must call into each other at import time
- they can cause your program to crash at startup
- the best way to break a circular dependency is refactoring mutual dependencies into a separate module at the bottom of the dependency tree
- dynamic imports are the simplest solution for breaking a circular dependency between modules while minimizing refactoring and complexity

53. Use Virtual Environments for Isolated and Reproducible Dependencies

- virtual environments allow you to use pip to install many different versions of the same package on the same machine without conflicts
- virtual environments are created with pyvenv, enabled with source bin/ activate, and disabled with deactivate
- you can dump all of the requirements of an environment with pip freeze
- you can reproduce the environment by supplying the requirements.txt file to pip install -r
- in versions of Python before 3.4, the pyvenv tool must be downloaded and installed separately
- the command-line tool is called virtualenv instead of pyvenv

8. Production

54. Consider Module-Scoped Code to Configure Deployment Environments

- programs often need to run in multiple deployment environments that each have unique assumptions and configurations
- you can tailor a module's contents to different deployment environments by using normal Python statements in module scope
- module contents can be the product of any external condition, including host introspection through the sys and os modules

55. Use repr Strings for Debugging Output

- calling print on built-in Python types will produce the human-readable string version of a value, which hides type information
- calling repr on built-in Python types will produce the printable string version of a value
- these repr strings could be passed to the eval built-in function to get back the original value

- %s in format strings will produce human-readable strings like str
- %r will produce printable strings like repr
- you can define the **repr** method to customize the printable representation of a class and provide more detailed debugging information
- you can reach into any object's **dict** attribute to view its internals

56. Test Everything with unittest

- the only way to have confidence in a Python program is to write tests
- the unittest built-in module provides most of the facilities you'll need to write good tests
- you can define tests by subclassing TestCase and defining one method per behavior you'd like to test
- test methods on TestCase classes must start with the word test
- it's important to write both unit tests (for isolated functionality) and integration tests (for modules that interact)

57. Consider Interactive Debugging with pdb

- you can initiate the Python interactive debugger at a point of interest directly in your program with the import pdb; pdb.set_trace() statements
- the Python debugger prompt is a full Python shell that lets you inspect and modify the state of a running program
- pdb shell commands let you precisely control program execution, allowing you to alternate between inspecting program state and progressing program execution

58. Profile Before Optimizing

- it's important to profile Python programs before optimizing because the source of slowdowns is often obscure
- use the cProfile module instead of the profile module because it provides more accurate profiling information
- the Profile object's runcall method provides everything you need to profile a tree of function calls in isolation
- the Stats object lets you select and print the subset of profiling information you need to see to understand your program's performance

59. Use tracemalloc to Understand Memory Usage and Leaks

- it can be difficult to understand how Python programs use and leak memory
- the gc module can help you understand which objects exist, but it has no information about how they were allocated

- the tracemalloc built-in module provides powerful tools for understanding the source of memory usage
- tracemalloc is only available in Python 3.4 and above