```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import timeit
>>> def my_function():
    y = 3.1415
    for x in range(100):
        y = y ** 0.7
    return y

>>> print(timeit.timeit(my_function, number=100))
0.0018824000000066121
>>> print(timeit.timeit(my_function, number=100000))
1.2804172999999963
>>> print(timeit.timeit(my_function, number=1))
2.289999999760539e-05
>>> import timne
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    import timne
ModuleNotFoundError: No module named 'timne'
>>> import time
>>> dir(time)
['_STRUCT_TM_ITEMS', '__doc__', '__loader__', '__name__', '__package__', '__spec__',
'altzone', 'asctime', 'ctime', 'daylight', 'get_clock_info', 'gmtime', 'localtime',
'mktime', 'monotonic', 'monotonic_ns', 'perf_counter', 'perf_counter_ns',
'process_time', 'process_time_ns', 'sleep', 'strftime', 'strptime', 'struct_time',
'thread_time', 'thread_time_ns', 'time', 'time_ns', 'timezone', 'tzname']
>>> help(time)
Help on built-in module time:

NAME
    time - This module provides various functions to manipulate time values.

DESCRIPTION
    There are two standard representations of time.  One is the number
    of seconds since the Epoch, in UTC (a.k.a. GMT).  It may be an integer
    or a floating point number (to represent fractions of seconds).
    The Epoch is system-defined; on Unix, it is generally January 1st, 1970.
    The actual value can be retrieved by calling gmtime(0).

    The other representation is a tuple of 9 integers giving local time.
    The tuple items are:
      year (including century, e.g. 1998)
      month (1-12)
      day (1-31)
      hours (0-23)
      minutes (0-59)
      seconds (0-59)
      weekday (0-6, Monday is 0)
```

```
      Julian day (day in the year, 1-366)
      DST (Daylight Savings Time) flag (-1, 0 or 1)
    If the DST flag is 0, the time is given in the regular time zone;
    if it is 1, the time is given in the DST time zone;
    if it is -1, mktime() should guess based on the date and time.

CLASSES
    builtins.tuple(builtins.object)
        struct_time

    class struct_time(builtins.tuple)
     |  struct_time(iterable=(), /)
     |
     |  The time value as returned by gmtime(), localtime(), and strptime(), and
     |  accepted by asctime(), mktime() and strftime().  May be considered as a
     |  sequence of 9 integers.
     |
     |  Note that several fields' values are not the same as those defined by
     |  the C language standard for struct tm.  For example, the value of the
     |  field tm_year is the actual year, not year - 1900.  See individual
     |  fields' descriptions for details.
     |
     |  Method resolution order:
     |      struct_time
     |      builtins.tuple
     |      builtins.object
     |
     |  Methods defined here:
     |
     |  __reduce__(...)
     |      Helper for pickle.
     |
     |  __repr__(self, /)
     |      Return repr(self).
     |
     |  ----------------------------------------------------------------------
     |  Static methods defined here:
     |
     |  __new__(*args, **kwargs) from builtins.type
     |      Create and return a new object.  See help(type) for accurate signature.
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  tm_gmtoff
     |      offset from UTC in seconds
     |
     |  tm_hour
     |      hours, range [0, 23]
     |
```

```
|  tm_isdst
|      1 if summer time is in effect, 0 if not, and -1 if unknown
|
|  tm_mday
|      day of month, range [1, 31]
|
|  tm_min
|      minutes, range [0, 59]
|
|  tm_mon
|      month of year, range [1, 12]
|
|  tm_sec
|      seconds, range [0, 61])
|
|  tm_wday
|      day of week, range [0, 6], Monday is 0
|
|  tm_yday
|      day of year, range [1, 366]
|
|  tm_year
|      year, for example, 1993
|
|  tm_zone
|      abbreviation of timezone name
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  n_fields = 11
|
|  n_sequence_fields = 9
|
|  n_unnamed_fields = 0
|
|  ----------------------------------------------------------------------
|  Methods inherited from builtins.tuple:
|
|  __add__(self, value, /)
|      Return self+value.
|
|  __contains__(self, key, /)
|      Return key in self.
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __ge__(self, value, /)
|      Return self>=value.
```

```
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __getnewargs__(self, /)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  count(self, value, /)
 |      Return number of occurrences of value.
 |
 |  index(self, value, start=0, stop=2147483647, /)
 |      Return first index of value.
 |
 |      Raises ValueError if the value is not present.

FUNCTIONS
    asctime(...)
        asctime([tuple]) -> string

        Convert a time tuple to a string, e.g. 'Sat Jun 06 16:26:11 1998'.
        When the time tuple is not present, current time as returned by localtime()
```

```
    is used.

ctime(...)
    ctime(seconds) -> string

    Convert a time in seconds since the Epoch to a string in local time.
    This is equivalent to asctime(localtime(seconds)). When the time tuple is
    not present, current time as returned by localtime() is used.

get_clock_info(...)
    get_clock_info(name: str) -> dict

    Get information of the specified clock.

gmtime(...)
    gmtime([seconds]) -> (tm_year, tm_mon, tm_mday, tm_hour, tm_min,
                          tm_sec, tm_wday, tm_yday, tm_isdst)

    Convert seconds since the Epoch to a time tuple expressing UTC (a.k.a.
    GMT).  When 'seconds' is not passed in, convert the current time instead.

    If the platform supports the tm_gmtoff and tm_zone, they are available as
    attributes only.

localtime(...)
    localtime([seconds]) -> (tm_year,tm_mon,tm_mday,tm_hour,tm_min,
                             tm_sec,tm_wday,tm_yday,tm_isdst)

    Convert seconds since the Epoch to a time tuple expressing local time.
    When 'seconds' is not passed in, convert the current time instead.

mktime(...)
    mktime(tuple) -> floating point number

    Convert a time tuple in local time to seconds since the Epoch.
    Note that mktime(gmtime(0)) will not generally return zero for most
    time zones; instead the returned value will either be equal to that
    of the timezone or altzone attributes on the time module.

monotonic(...)
    monotonic() -> float

    Monotonic clock, cannot go backward.

monotonic_ns(...)
    monotonic_ns() -> int

    Monotonic clock, cannot go backward, as nanoseconds.

perf_counter(...)
```

```
    perf_counter() -> float

    Performance counter for benchmarking.

perf_counter_ns(...)
    perf_counter_ns() -> int

    Performance counter for benchmarking as nanoseconds.

process_time(...)
    process_time() -> float

    Process time for profiling: sum of the kernel and user-space CPU time.

process_time_ns(...)
    process_time() -> int

    Process time for profiling as nanoseconds:
    sum of the kernel and user-space CPU time.

sleep(...)
    sleep(seconds)

    Delay execution for a given number of seconds.  The argument may be
    a floating point number for subsecond precision.

strftime(...)
    strftime(format[, tuple]) -> string

    Convert a time tuple to a string according to a format specification.
    See the library reference manual for formatting codes. When the time tuple
    is not present, current time as returned by localtime() is used.

    Commonly used format codes:

    %Y  Year with century as a decimal number.
    %m  Month as a decimal number [01,12].
    %d  Day of the month as a decimal number [01,31].
    %H  Hour (24-hour clock) as a decimal number [00,23].
    %M  Minute as a decimal number [00,59].
    %S  Second as a decimal number [00,61].
    %z  Time zone offset from UTC.
    %a  Locale's abbreviated weekday name.
    %A  Locale's full weekday name.
    %b  Locale's abbreviated month name.
    %B  Locale's full month name.
    %c  Locale's appropriate date and time representation.
    %I  Hour (12-hour clock) as a decimal number [01,12].
    %p  Locale's equivalent of either AM or PM.
```

```
    Other codes may be available on your platform.  See documentation for
    the C library strftime function.

strptime(...)
    strptime(string, format) -> struct_time

    Parse a string to a time tuple according to a format specification.
    See the library reference manual for formatting codes (same as
    strftime()).

    Commonly used format codes:

    %Y  Year with century as a decimal number.
    %m  Month as a decimal number [01,12].
    %d  Day of the month as a decimal number [01,31].
    %H  Hour (24-hour clock) as a decimal number [00,23].
    %M  Minute as a decimal number [00,59].
    %S  Second as a decimal number [00,61].
    %z  Time zone offset from UTC.
    %a  Locale's abbreviated weekday name.
    %A  Locale's full weekday name.
    %b  Locale's abbreviated month name.
    %B  Locale's full month name.
    %c  Locale's appropriate date and time representation.
    %I  Hour (12-hour clock) as a decimal number [01,12].
    %p  Locale's equivalent of either AM or PM.

    Other codes may be available on your platform.  See documentation for
    the C library strftime function.

thread_time(...)
    thread_time() -> float

    Thread time for profiling: sum of the kernel and user-space CPU time.

thread_time_ns(...)
    thread_time() -> int

    Thread time for profiling as nanoseconds:
    sum of the kernel and user-space CPU time.

time(...)
    time() -> floating point number

    Return the current time in seconds since the Epoch.
    Fractions of a second may be present if the system clock provides them.

time_ns(...)
    time_ns() -> int
```

```
        Return the current time in nanoseconds since the Epoch.

DATA
    altzone = 18000
    daylight = 1
    timezone = 21600
    tzname = ('Central Standard Time', 'Central Daylight Time')

FILE
    (built-in)


>>> help(timeit)
Help on module timeit:

NAME
    timeit - Tool for measuring execution time of small code snippets.

DESCRIPTION
    This module avoids a number of common traps for measuring execution
    times.  See also Tim Peters' introduction to the Algorithms chapter in
    the Python Cookbook, published by O'Reilly.

    Library usage: see the Timer class.

    Command line usage:
        python timeit.py [-n N] [-r N] [-s S] [-p] [-h] [--] [statement]

    Options:
      -n/--number N: how many times to execute 'statement' (default: see below)
      -r/--repeat N: how many times to repeat the timer (default 5)
      -s/--setup S: statement to be executed once initially (default 'pass').
                    Execution time of this setup statement is NOT timed.
      -p/--process: use time.process_time() (default is time.perf_counter())
      -v/--verbose: print raw timing results; repeat for more digits precision
      -u/--unit: set the output time unit (nsec, usec, msec, or sec)
      -h/--help: print this usage message and exit
      --: separate options from statement, use when statement starts with -
      statement: statement to be timed (default 'pass')

    A multi-line statement may be given by specifying each line as a
    separate argument; indented lines are possible by enclosing an
    argument in quotes and using leading spaces.  Multiple -s options are
    treated similarly.

    If -n is not given, a suitable number of loops is calculated by trying
    successive powers of 10 until the total time is at least 0.2 seconds.

    Note: there is a certain baseline overhead associated with executing a
    pass statement.  It differs between versions.  The code here doesn't try
```

```
    to hide it, but you should be aware of it.  The baseline overhead can be
    measured by invoking the program without arguments.

    Classes:

        Timer

    Functions:

        timeit(string, string) -> float
        repeat(string, string) -> list
        default_timer() -> float

CLASSES
    builtins.object
        Timer

    class Timer(builtins.object)
     |  Timer(stmt='pass', setup='pass', timer=<built-in function perf_counter>,
globals=None)
     |
     |  Class for timing execution speed of small code snippets.
     |
     |  The constructor takes a statement to be timed, an additional
     |  statement used for setup, and a timer function.  Both statements
     |  default to 'pass'; the timer function is platform-dependent (see
     |  module doc string).  If 'globals' is specified, the code will be
     |  executed within that namespace (as opposed to inside timeit's
     |  namespace).
     |
     |  To measure the execution time of the first statement, use the
     |  timeit() method.  The repeat() method is a convenience to call
     |  timeit() multiple times and return a list of results.
     |
     |  The statements may contain newlines, as long as they don't contain
     |  multi-line string literals.
     |
     |  Methods defined here:
     |
     |  __init__(self, stmt='pass', setup='pass', timer=<built-in function
perf_counter>, globals=None)
     |      Constructor.  See class doc string.
     |
     |  autorange(self, callback=None)
     |      Return the number of loops and time taken so that total time >= 0.2.
     |
     |      Calls the timeit method with increasing numbers from the sequence
     |      1, 2, 5, 10, 20, 50, ... until the time taken is at least 0.2
     |      second.  Returns (number, time_taken).
     |
```

```
|      If *callback* is given and is not None, it will be called after
|      each trial with two arguments: ``callback(number, time_taken)``.
|
|  print_exc(self, file=None)
|      Helper to print a traceback from the timed code.
|
|      Typical use:
|
|          t = Timer(...)       # outside the try/except
|          try:
|              t.timeit(...)    # or t.repeat(...)
|          except:
|              t.print_exc()
|
|      The advantage over the standard traceback is that source lines
|      in the compiled template will be displayed.
|
|      The optional file argument directs where the traceback is
|      sent; it defaults to sys.stderr.
|
|  repeat(self, repeat=5, number=1000000)
|      Call timeit() a few times.
|
|      This is a convenience function that calls the timeit()
|      repeatedly, returning a list of results.  The first argument
|      specifies how many times to call timeit(), defaulting to 5;
|      the second argument specifies the timer argument, defaulting
|      to one million.
|
|      Note: it's tempting to calculate mean and standard deviation
|      from the result vector and report these.  However, this is not
|      very useful.  In a typical case, the lowest value gives a
|      lower bound for how fast your machine can run the given code
|      snippet; higher values in the result vector are typically not
|      caused by variability in Python's speed, but by other
|      processes interfering with your timing accuracy.  So the min()
|      of the result is probably the only number you should be
|      interested in.  After that, you should look at the entire
|      vector and apply common sense rather than statistics.
|
|  timeit(self, number=1000000)
|      Time 'number' executions of the main statement.
|
|      To be precise, this executes the setup statement once, and
|      then returns the time it takes to execute the main statement
|      a number of times, as a float measured in seconds.  The
|      argument is the number of times through the loop, defaulting
|      to one million.  The main statement, the setup statement and
|      the timer function to be used are passed to the constructor.
|
```

```
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)

FUNCTIONS
    default_timer = perf_counter(...)
        perf_counter() -> float

        Performance counter for benchmarking.

    repeat(stmt='pass', setup='pass', timer=<built-in function perf_counter>,
repeat=5, number=1000000, globals=None)
        Convenience function to create Timer object and call repeat method.

    timeit(stmt='pass', setup='pass', timer=<built-in function perf_counter>,
number=1000000, globals=None)
        Convenience function to create Timer object and call timeit method.

DATA
    __all__ = ['Timer', 'timeit', 'repeat', 'default_timer']

FILE
    c:\users\wabbo\appdata\local\programs\python\python38-32\lib\timeit.py


>>> 2.2899999997*e-5
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    2.2899999997*e-5
NameError: name 'e' is not defined
>>> 2.289999999760539e-05
2.289999999760539e-05
>>> 0.000022899999997
2.2899999997e-05
>>> 0.000022899
2.2899e-05
>>> print(timeit.timeit(my_function, number=100000) / 100000)
1.2745420999999623e-05

>>> print(timeit.timeit(my_function, number=100000) / 100000)
1.2764709000002768e-05
>>> print(timeit.timeit(my_function, number=100000) / 100000)
1.2686349000000518e-05
>>> %timeit -n 100000 my_function()
SyntaxError: invalid syntax
```

```
>>> print(timeit.timeit(my_function, number=1) / 100000)
2.2499999886349543e-10
>>> print(timeit.timeit(my_function, number=1))
2.20999995690363e-05
>>> print(timeit.timeit(my_function, number=1) * 1000)
0.022599999738304177
>>> print(timeit.timeit(my_function, number=100000) * 1000)
1272.3511999997754
>>> print(timeit.timeit(my_function, number=1) * 1000000000)
23199.99975952669
>>> print(timeit.timeit(my_function, number=1) * 1000000000)
16899.999991437653
>>> print(timeit.timeit(my_function, number=1) * 1000000000)
26499.999876250513
>>> from statistics import mean
>>> for i in range(100):
        lst[i] = timeit.timeit(my_function, number=1) * 1000000000


Traceback (most recent call last):
  File "<pyshell#29>", line 2, in <module>
    lst[i] = timeit.timeit(my_function, number=1) * 1000000000
NameError: name 'lst' is not defined
>>> lst[]
SyntaxError: invalid syntax
>>> lst = []
>>> for i in range(100):
        lst.append(timeit.timeit(my_function, number=1) * 1000000000)


>>> lst
[22100.00002378365, 18400.000044493936, 18499.99989644857, 18300.0001925393,
18099.99957913533, 18299.99973779195, 18199.999885837315, 18300.0001925393,
18100.00003388268, 18199.999885837315, 18199.999885837315, 18199.999885837315,
18100.00003388268, 18199.999885837315, 18100.00003388268, 18000.000181928044,
18100.00003388268, 18199.999885837315, 18100.00003388268, 18300.0001925393,
18100.00003388268, 18000.000181928044, 18100.00003388268, 18100.00003388268,
18100.00003388268, 18000.000181928044, 17999.999727180693, 18000.000181928044,
18100.00003388268, 18100.00003388268, 18299.99973779195, 18199.999885837315,
18299.99973779195, 18199.999885837315, 18200.000340584666, 18000.000181928044,
18100.00003388268, 18100.00003388268, 18100.00003388268, 18100.00003388268,
17999.999727180693, 17999.999727180693, 18100.00003388268, 18100.00003388268,
17999.999727180693, 18100.00003388268, 18000.000181928044, 18100.00003388268,
18100.00003388268, 18199.999885837315, 18000.000181928044, 17999.999727180693,
18000.000181928044, 18199.999885837315, 18100.00003388268, 17999.999727180693,
18000.000181928044, 18100.00003388268, 18199.999885837315, 17999.999727180693,
18199.999885837315, 18199.999885837315, 18199.999885837315, 18000.000181928044,
18100.00003388268, 18000.000181928044, 17999.999727180693, 18299.99973779195,
18000.000181928044, 17899.999875226058, 18000.000181928044, 18100.00003388268,
17900.00032997341, 17999.999727180693, 18000.000181928044, 19499.999780236976,
```

```
18199.999885837315, 18000.000181928044, 18100.00003388268, 27200.000204175012,
13700.000181415817, 13700.000181415817, 13800.000033370452, 13699.999726668466,
13799.999578623101, 13899.999885325087, 13699.999726668466, 13600.000329461182,
13800.000033370452, 13799.999578623101, 13700.000181415817, 13800.000033370452,
14900.000223860843, 13800.000033370452, 13899.999885325087, 13700.000181415817,
13699.999726668466, 14199.999895936344, 13699.999726668466, 13700.000181415817]
>>> mean(lst)
17399.999983354064
>>>
```