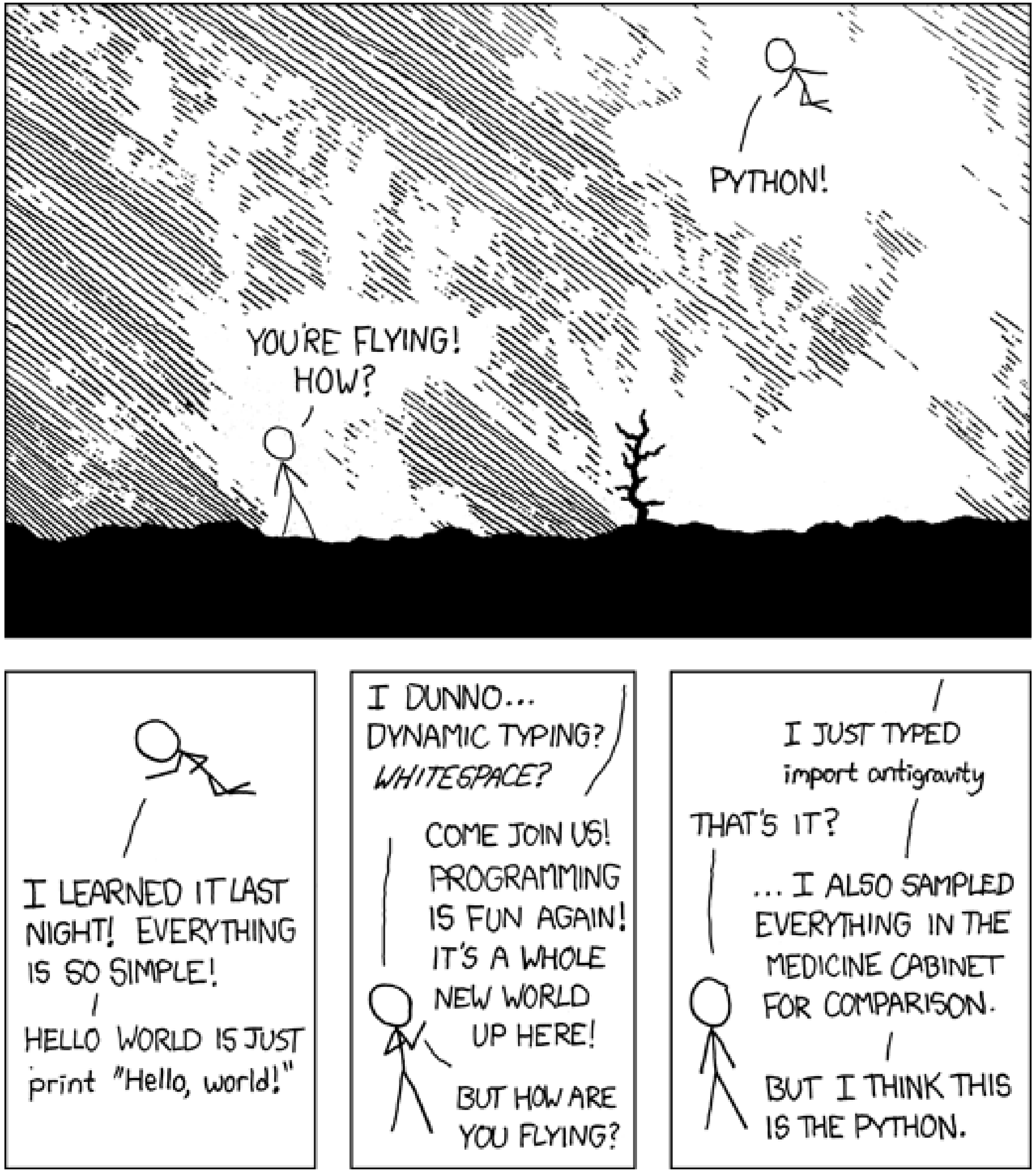

PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

xkcd.com/353

Modules and packages

- import – from – as
- __name__, "__main__"

docs.python.org/3/tutorial/modules.html

Python modules and packages

- A Python module is a *module_name*.py file containing Python code
- A Python package is a collection of modules

Why do you need modules ?

- A way to structure code into smaller logical units
- Encapsulation of functionality
- Reuse of code in different programs

- Your can write your own modules and packages or use any of the +100.000 existing packages from pypi.org
- The Python Standard Library consists of the modules listed on docs.python.org/3/library

Defining and importing a module

mymodule.py

```
""" This is a 'print something' module """

from random import randint

print("Running my module")

def print_something(n):
    W = ['Eat', 'Sleep', 'Rave', 'Repeat']
    words = (W[randint(0, len(W) - 1)] for _ in range(n))
    print(' '.join(words))

def the_name():
    print('__name__ = "' + __name__ +'"')
```

using_mymodule.py

```
import mymodule
mymodule.the_name()
mymodule.print_something(5)
from mymodule import print_something
print_something(5)
```

Python shell

```
Running my module
__name__ = "mymodule"
Eat Sleep Sleep Sleep Rave
Eat Sleep Rave Repeat Sleep
```

- A module is only run once when imported several times

Some modules mentioned in the course

Module (example functions)	Description
math (pi sqrt ceil log sin)	*basic math*
random (random randint)	*random number generator*
numpy (array shape)	*multi-dimensional data*
pandas	*data tables*
SQLlite	*SQL database*
scipy scipy.optimize (minimize linprog) scipy.spatial (ConvexHull)	*mathematical optimization*
matplotlib matplotlib.pyplot (plot show style) matplotlib.backends.backend_pdf (PdfPages) mpl_toolkits.mplot3d (Axes3D)	 *plotting data* *print plots to PDF* *3D plot tools*
doctest (testmod) unittest (assertEqual assertTrue)	*testing using doc strings* *unit testing*
time (time) datetime (date.today)	*current time, coversion of time values*
timeit (timeit)	*time execution of simple code*
heapq	*use a list as a heap*

Module (example functions)	Description
functools (lru_cache total_ordering)	*higher order functions and decorators*
itertools (islice permutations)	*Iterator tools*
collections (Counter deque)	*datat structures for collections*
builtins	*module containing the Python builtins*
os (path)	*operating system interface*
sys (argv path)	*system specific functions*
Tkinter PyQt	*graphic user interface*
xml	*xml files (eXtensible Markup Language)*
json	*JSON (JavaScript Object Notation) files*
csv	*comma separated files*
openpyxl	*EXCEL files*
re	*regular expression, string searching*
string (split join lower ascii_letters digits)	*string functions*

Ways of importing modules

import.py

```
# Import a module name in the current namespace
# All definitions in the module are available as <module>.<name>

import math
print(math.sqrt(2))

# Import only one or more specific definitions into current namespace

from math import sqrt, log, ceil
print(ceil(log(sqrt(100), 2)))

# Import specific modules/definitions from a module into current namespace under new names

from math import sqrt as kvadratrod, \  # long import line broken onto multiple lines
                 log as logaritme
import matplotlib.pyplot as plt
print(logaritme(kvadratrod(100)))

# Import all definitions form a module in current namespace
# Deprecated, since unclear what happens to the namespace

from math import *
print(pi)  # where did 'pi' come from?
```

Python shell

```
| 1.4142135623730951
| 4
| 2.302585092994046
| 3.141592653589793
```

Performance of different ways of importing

`from math import sqrt`

appears to be faster than

`math.sqrt`

sqrt_performance.py

```python
from time import time
import math
start = time()
x = sum(math.sqrt(x) for x in range(10000000))
end = time()
print("math.sqrt", end - start)
from math import sqrt
start = time()
x = sum(sqrt(x) for x in range(10000000))
end = time()
print("from math import sqrt", end - start)
def test(sqrt=math.sqrt):  # abuse of keyword argument
    start = time()
    x = sum(sqrt(x) for x in range(10000000))
    end = time()
    print("bind sqrt to keyword argument", end - start)
test()
```

Python shell

```
| math.sqrt 4.05187726020813
| from math import sqrt 3.5011463165283203
| bind sqrt to keyword argument 3.261594772338867
```

Listing definitions in a module: dir(*module*)

Python shell

```
> import math
> import matplotlib.pyplot as plt
> dir(math)
| ['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
  'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
  'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
  'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot',
  'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
  'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin',
  'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
> help(math)
| Help on built-in module math:
| NAME
|     math
| DESCRIPTION
| ...
```

https://docs.python.org/3/library/functions.html#dir

__name__

double.py

```
""" Module double """
def f(x):
    """
    Some doc test code:
    >>> f(21)
    42
    >>> f(7)
    14
    """
    return 2 * x
print('__name__ =', __name__)
if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

Python shell

```
  __name__ = __main__
  ...
  2 passed and 0 failed.
  Test passed.
```

using_double.py

```
import double
print(__name__)
print(double.f(5))
```

Python shell

```
  __name__ = double
  __main__
  10
```

- The variable `__name__` contains the name of the module, or `'__main__'` if the file is run as the main file by the interpreter
- Can e.g. be used to test a module if the module is run independently

module **`importlib`**

- Implements the `import` statement (Python internal implementation details)
- `importlib.reload(`*module*`)`
  - Reloads a previously imported *module*. Relevant if you have edited the code for the module and want to load the new version in the Python interpreter, without restarting the full program from scratch.

a_constant.py

```
the_constant = 7
```

Python shell

```
> import a_constant  # import module
> a_constant.the_constant
| 7
> from a_constant import the_constant
> the_constant
| 7
# Update 7 to 42 in a_constant.py
> a_constant.the_constant  # new value not reflected
| 7
> import a_constant  # void, module already loaded
> a_constant.the_constant
| 7  # unchanged
> import importlib
> importlib.reload(a_constant)
| <module 'a_constant' from 'C:\\...\\a_constant.py'>
> a_constant.the_constant
| 42
> the_constant
| 7  # imported attributes are not updated by reload
> from a_constant import the_constant  # force update
> the_constant
| 42  # the new value
```

Packages

- A package is a collection of modules (and subpackages) in a folder = package name
- Only folders having an `__init__.py` file are considered packages
- The `__init__.py` can be empty, or contain code that will be loaded when the package is imported, e.g. importing specific modules

mypackage/__init__.py

```
```

mypackage/a.py

```python
print("Loading mypackage.a")
def f():
    print("mypackage.a.f")
```

using_mypackage.py

```python
import mypackage.a
mypackage.a.f()
```

Python shell

```
| Loading mypackage.a
| mypackage.a.f
```

A package with a subpackage

mypackage/__init__.py

```
print('loading mypackage')
```

mypackage/a.py

```
print('Loading mypackage.a')
def f():
    print('mypackage.a.f')
```

mypackage/mysubpackage/__init__.py

```
print('loading mypackage.mysubpackage')
import mypackage.mysubpackage.b
```

mypackage/mysubpackage/b.py

```
print('Loading mypackage.mysubpackage.b')
def g():
    print('mypackage.mysubpackage.b.g')
```

using_mysubpackage.py

```
import mypackage.a
mypackage.a.f()
import mypackage.mysubpackage
mypackage.mysubpackage.b.g()
from mypackage.mysubpackage.b import g
g()
```

Python shell

```
| loading mypackage
| Loading mypackage.a
| mypackage.a.f
| loading mypackage.mysubpackage
| Loading mypackage.mysubpackage.b
| mypackage.mysubpackage.b.g
| mypackage.mysubpackage.b.g
```

__pycache__ folder

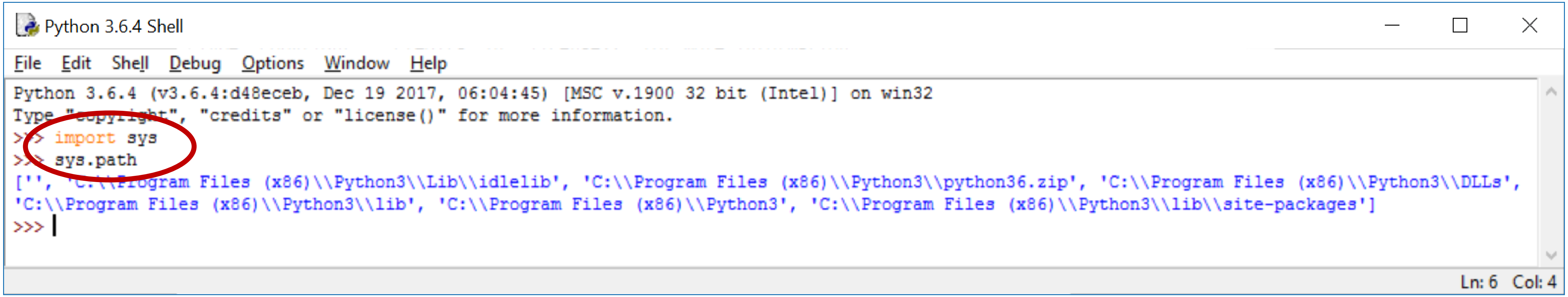
- When Python loads a module the first time it is *compiled* to some intermediate code, and stored as a .pyc file in the __pycache__ folder.
- If a .pyc file exists for a module, and the .pyc file is newer than the .py file, then import loads .pyc – saving time to load the module (but does not make the program itself faster).
- It is safe to delete the __pycache__ folder – but it will be created again next time a module is loaded.

Path to modules

Python searches the following folders for a module in the following order:

1) The directory containing the input script / current directory
2) *Environment* variable `PYTHONPATH`
3) Installation defaults

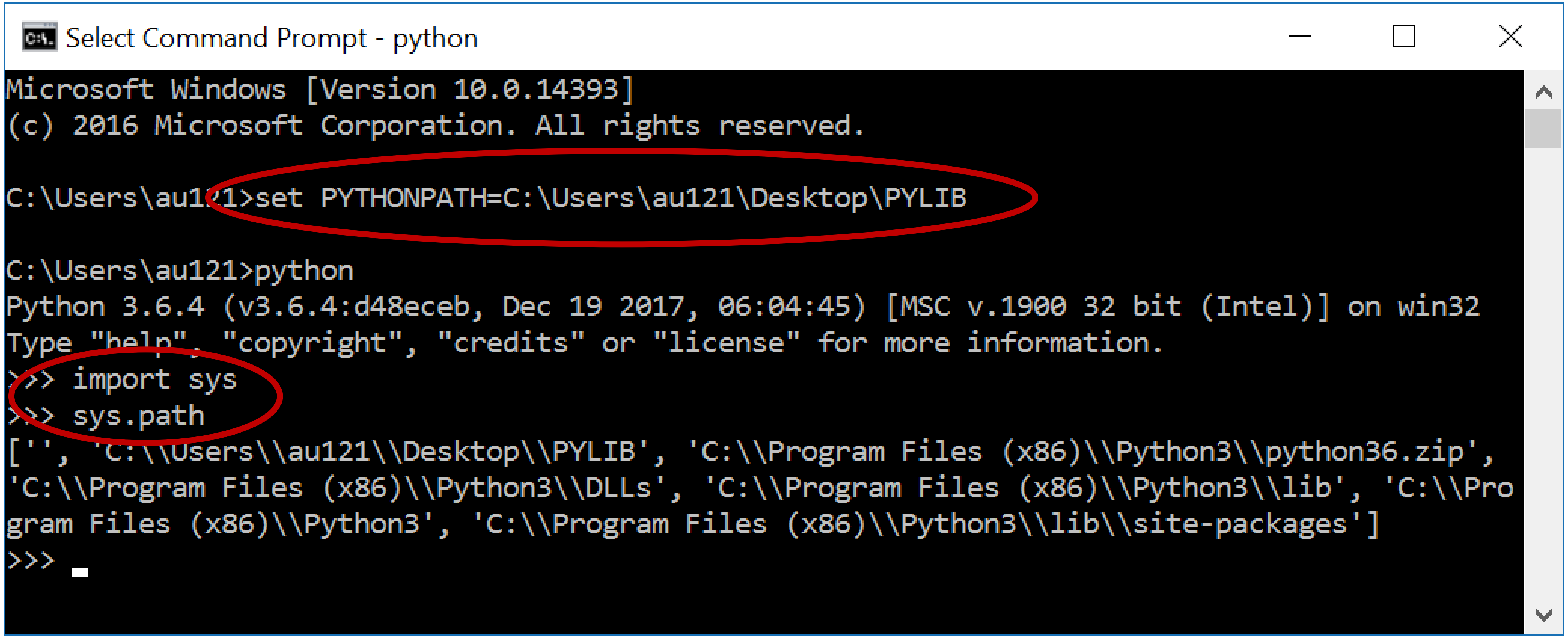
The function `path` in the modul `sys` returns a list of the paths



```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import sys
>>> sys.path
['', 'C:\\Program Files (x86)\\Python3\\Lib\\idlelib', 'C:\\Program Files (x86)\\Python3\\python36.zip', 'C:\\Program Files (x86)\\Python3\\DLLs',
'C:\\Program Files (x86)\\Python3\\lib', 'C:\\Program Files (x86)\\Python3', 'C:\\Program Files (x86)\\Python3\\lib\\site-packages']
>>> 
```

Setting PYTHONPATH from windows shell

- set PYTHONPATH=*paths separated by semicolon*
  (only valid until shell is closed)



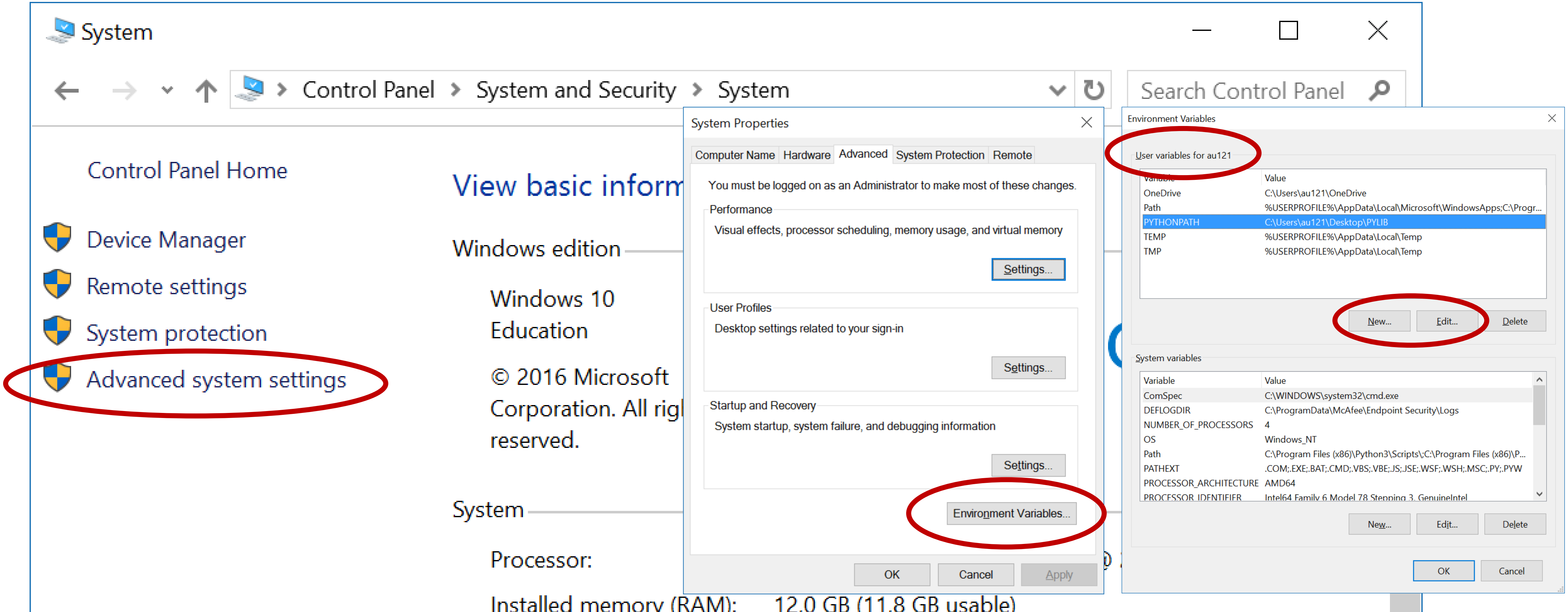
```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\au121>set PYTHONPATH=C:\Users\au121\Desktop\PYLIB

C:\Users\au121>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', 'C:\\Users\\au121\\Desktop\\PYLIB', 'C:\\Program Files (x86)\\Python3\\python36.zip',
'C:\\Program Files (x86)\\Python3\\DLLs', 'C:\\Program Files (x86)\\Python3\\lib', 'C:\\Pro
gram Files (x86)\\Python3', 'C:\\Program Files (x86)\\Python3\\lib\\site-packages']
>>>
```

Setting PYTHONPATH from control panel

- Control panel > System > Advanced system settings > Environment Variables > User variables > Edit or New PYTHONPATH



View basic inform

Windows edition

Windows 10 Education

© 2016 Microsoft Corporation. All rigl reserved.

System

Processor:

Installed memory (RAM): 12.0 GB (11.8 GB usable)

System Properties

You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles

Desktop settings related to your sign-in

Startup and Recovery

System startup, system failure, and debugging information

Environment Variables

User variables for au121

Variable	Value
OneDrive	C:\Users\au121\OneDrive
Path	%USERPROFILE%\AppData\Local\Microsoft\WindowsApps;C:\Progr...
PYTHONPATH	C:\Users\au121\Desktop\PYLIB
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

System variables

Variable	Value
ComSpec	C:\WINDOWS\system32\cmd.exe
DEFLOGDIR	C:\ProgramData\McAfee\Endpoint Security\Logs
NUMBER_OF_PROCESSORS	4
OS	Windows_NT
Path	C:\Program Files (x86)\Python3\Scripts\;C:\Program Files (x86)\P...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC;.PY;.PYW
PROCESSOR_ARCHITECTURE	AMD64
PROCESSOR_IDENTIFIER	Intel64 Family 6 Model 78 Stepping 3, GenuineIntel

Python shell www.python.org/dev/peps/pep-0020/

```
> import this
| The Zen of Python, by Tim Peters
|
| Beautiful is better than ugly.
| Explicit is better than implicit.
| Simple is better than complex.
| Complex is better than complicated.
| Flat is better than nested.
| Sparse is better than dense.
| Readability counts.
| Special cases aren't special enough to break the rules.
| Although practicality beats purity.
| Errors should never pass silently.
| Unless explicitly silenced.
| In the face of ambiguity, refuse the temptation to guess.
| There should be one-- and preferably only one --obvious way to do it.
| Although that way may not be obvious at first unless you're Dutch.
| Now is better than never.
| Although never is often better than *right* now.
| If the implementation is hard to explain, it's a bad idea.
| If the implementation is easy to explain, it may be a good idea.
| Namespaces are one honking great idea -- let's do more of those!
```

module **heapq** (Priority Queue)

- Implements a binary **heap** (Williams 1964).
- Stores a set of elements in a standard list, where arbitrary elements can be inserted efficiently and the smallest element can be extracted efficiently

heapq.heappush
heapq.heappop

docs.python.org/3/library/heapq.html

J. W. J. Williams. *Algorithm 232: Heapsort*. Communications of the ACM (1964)

heap.py

```
import heapq
from random import random

H = []  # a heap is just a list

for _ in range(10):
    heapq.heappush(H, random())

while True:
    x = heapq.heappop(H)
    print(x)
    heapq.heappush(H, x + random())
```

Python shell

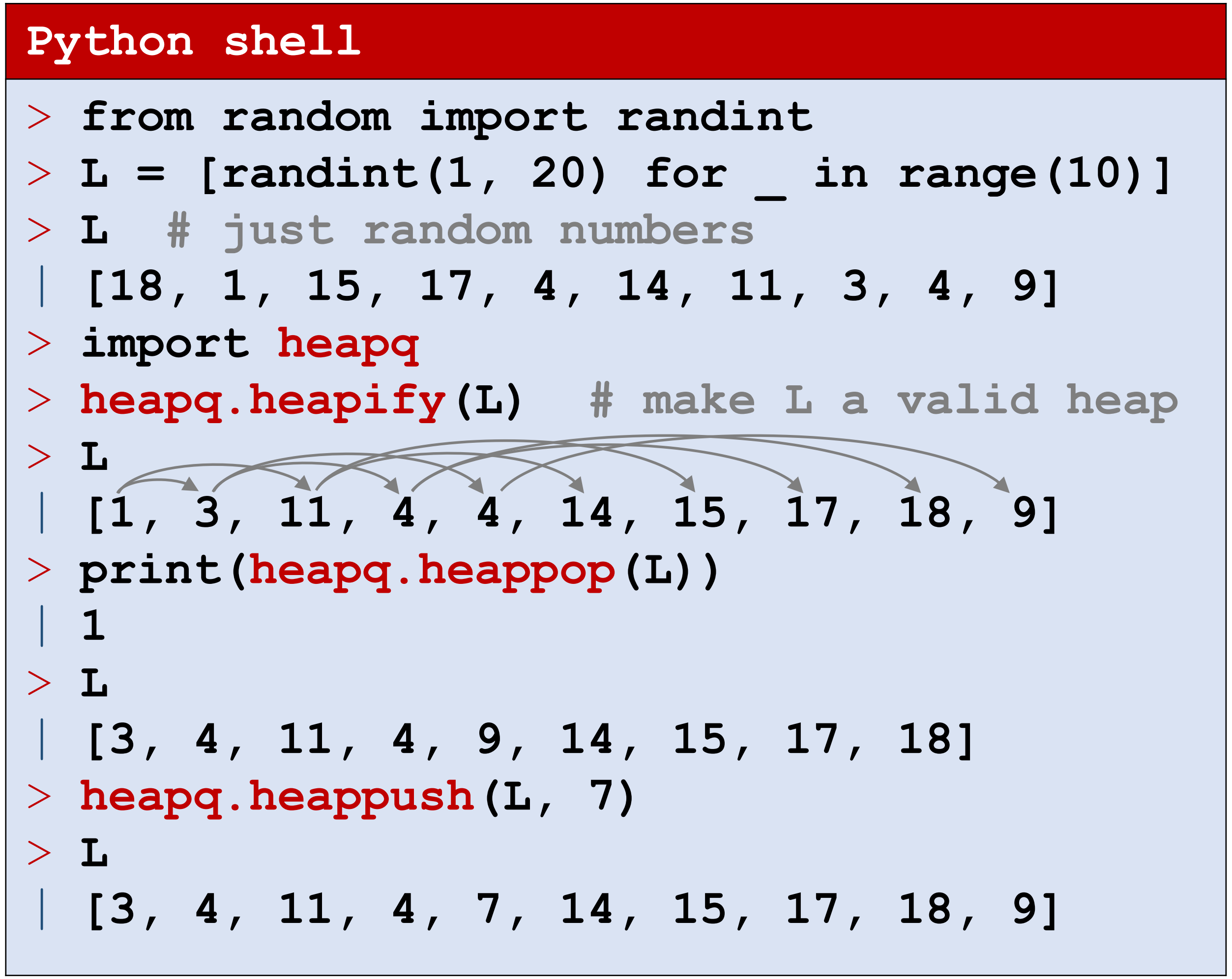
```
  0.2056993389276445 8
  0.2705781933961617 4
  0.3111561536287623 7
  0.484106227215225 9
  0.505428095600535 7
  0.50938711752407 6
  0.59864719548046 2
  0.703515073555502 7
  0.707392968582622 1
  0.709122401281532 5
  0.71421349612731 8
  0.72786848129127 1
  0.805127541375987 3
  0.827952376728290 3
  0.862602236320289 5
  0.937663123626386 9
```

Valid heap

- A *valid heap* satisfies for all i:

  L[i] ≤ L[2·i +1] and L[i] ≤ L[2·i + 2]

- **heapify(L)** rearranges the elements in a list to make the list a valid heap

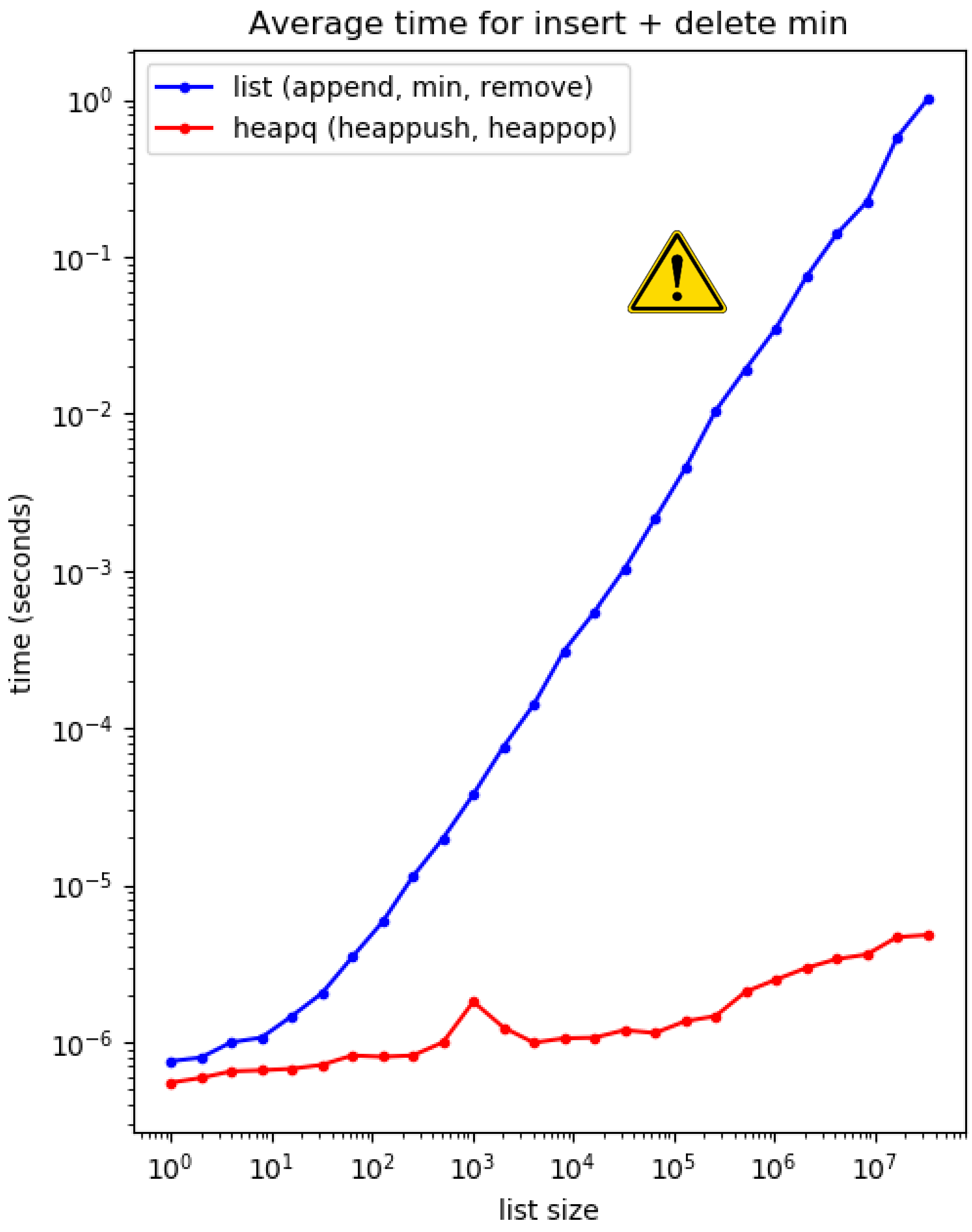


Python shell

```
> from random import randint
> L = [randint(1, 20) for _ in range(10)]
> L  # just random numbers
| [18, 1, 15, 17, 4, 14, 11, 3, 4, 9]
> import heapq
> heapq.heapify(L)  # make L a valid heap
> L
| [1, 3, 11, 4, 4, 14, 15, 17, 18, 9]
> print(heapq.heappop(L))
| 1
> L
| [3, 4, 11, 4, 9, 14, 15, 17, 18]
> heapq.heappush(L, 7)
> L
| [3, 4, 11, 4, 7, 14, 15, 17, 18, 9]
```

Why **heapq** ?

- `min` and `remove` on a list take *linear time* (runs through the whole list)
- heapq supports `heappush` and `heappop` in *logarithmic time*
- For lists of length 30.000.000 the performance gain is a factor 200.000



heap_performance.py (generating plot on previous slide)

```
import heapq
from random import random
import matplotlib.pyplot as plt
from time import time
import gc  # garbage collection

size = []
time_heap = []
time_list = []

for i in range(26):
    n = 2 ** i
    size.append(n)

    L = [random() for _ in range(n)]
    R = max(1, 2 ** 23 // n)
  Ⓑ gc.collect()
    start = time()
    for _ in range(R):
        L.append(random())
        x = min(L)
        L.remove(x)
    end = time()
    time_list.append((end - start) / R)

  Ⓐ L = None  # avoid MemoryError
    L = [random() for _ in range(n)]
    heapq.heapify(L)  # make L a legal heap
  Ⓑ gc.collect()
    start = time()
    for _ in range(100000):
        heapq.heappush(L, random())
        x = heapq.heappop(L)
    end = time()
    time_heap.append((end - start) / 100000)

plt.title("Average time for insert + delete min")
plt.xlabel("list size")
plt.ylabel("time (seconds)")
plt.plot(size, time_list, 'b.-',
         label='list (append, min, remove)')
plt.plot(size, time_heap, 'r.-',
         label='heapq (heappush, heappop)')
plt.xscale('log')
plt.yscale('log')
plt.legend()
plt.show()
```

Ⓐ Avoid out of memory error for largest experiment, by allowing old **L** to be garbage collected

Ⓑ Reduce noise in experiments by forcing Python garbage collection before measurement