psutil Documentation

Release 4.4.1

Giampaolo Rodola

Sep 07, 2017

Contents

CHAPTER 1

Quick links

- Home page
- Install
- Blog
- Forum
- Download
- Development guide
- What's new

CHAPTER 2

About

psutil (python system and process utilities) is a cross-platform library for retrieving information on running **processes** and **system utilization** (CPU, memory, disks, network, sensors) in **Python**. It is useful mainly for **system monitoring**, **profiling**, **limiting process resources** and the **management of running processes**. It implements many functionalities offered by command line tools such as: *ps, top, lsof, netstat, ifconfig, who, df, kill, free, nice, ionice, iostat, iotop, uptime, pidof, tty, taskset, pmap*. It currently supports **Linux, Windows, OSX, Sun Solaris, FreeBSD, OpenBSD** and **NetBSD**, both **32-bit** and **64-bit** architectures, with Python versions from **2.6 to 3.6** (users of Python 2.4 and 2.5 may use 2.1.3 version). PyPy is also known to work.

The psutil documentation you're reading is distributed as a single HTML page.

CHAPTER 3

Install

The easiest way to install psutil is via `pip`:

```
pip install psutil
```

On UNIX this requires a C compiler (e.g. gcc) installed. On Windows pip will automatically retrieve a pre-compiled wheel version from PYPI repository. Alternatively, see more detailed install instructions.

CHAPTER 4

System related functions

CPU

psutil.**cpu_times**(*percpu=False*)

Return system CPU times as a named tuple. Every attribute represents the seconds the CPU has spent in the given mode. The attributes availability varies depending on the platform:

- **user**: time spent by normal processes executing in user mode; on Linux this also includes **guest** time
- **system**: time spent by processes executing in kernel mode
- **idle**: time spent doing nothing

Platform-specific fields:

- **nice** *(UNIX)*: time spent by niced (prioritized) processes executing in user mode; on Linux this also includes **guest_nice** time
- **iowait** *(Linux)*: time spent waiting for I/O to complete
- **irq** *(Linux, BSD)*: time spent for servicing hardware interrupts
- **softirq** *(Linux)*: time spent for servicing software interrupts
- **steal** *(Linux 2.6.11+)*: time spent by other operating systems running in a virtualized environment
- **guest** *(Linux 2.6.24+)*: time spent running a virtual CPU for guest operating systems under the control of the Linux kernel
- **guest_nice** *(Linux 3.2.0+)*: time spent running a niced guest (virtual CPU for guest operating systems under the control of the Linux kernel)
- **interrupt** *(Windows)*: time spent for servicing hardware interrupts (similar to “irq” on UNIX)
- **dpc** *(Windows)*: time spent servicing deferred procedure calls (DPCs); DPCs are interrupts that run at a lower priority than standard interrupts.

When *percpu* is `True` return a list of named tuples for each logical CPU on the system. First element of the list refers to first CPU, second element to second CPU and so on. The order of the list is consistent across calls. Example output on Linux:

```
>>> import psutil
>>> psutil.cpu_times()
scputimes(user=17411.7, nice=77.99, system=3797.02, idle=51266.57, iowait=732.58,␣
↪irq=0.01, softirq=142.43, steal=0.0, guest=0.0, guest_nice=0.0)
```

Changed in version 4.1.0: added *interrupt* and *dpc* fields on Windows.

`psutil.`**`cpu_percent`**(*interval=None*, *percpu=False*)

Return a float representing the current system-wide CPU utilization as a percentage. When *interval* is > `0.0` compares system CPU times elapsed before and after the interval (blocking). When *interval* is `0.0` or `None` compares system CPU times elapsed since last call or module import, returning immediately. That means the first time this is called it will return a meaningless `0.0` value which you are supposed to ignore. In this case it is recommended for accuracy that this function be called with at least `0.1` seconds between calls. When *percpu* is `True` returns a list of floats representing the utilization as a percentage for each CPU. First element of the list refers to first CPU, second element to second CPU and so on. The order of the list is consistent across calls.

```
>>> import psutil
>>> # blocking
>>> psutil.cpu_percent(interval=1)
2.0
>>> # non-blocking (percentage since last call)
>>> psutil.cpu_percent(interval=None)
2.9
>>> # blocking, per-cpu
>>> psutil.cpu_percent(interval=1, percpu=True)
[2.0, 1.0]
>>>
```

Warning: the first time this function is called with *interval* = `0.0` or `None` it will return a meaningless `0.0` value which you are supposed to ignore.

`psutil.`**`cpu_times_percent`**(*interval=None*, *percpu=False*)

Same as *cpu_percent()* but provides utilization percentages for each specific CPU time as is returned by *psutil.cpu_times(percpu=True)*. *interval* and *percpu* arguments have the same meaning as in *cpu_percent()*. On Linux "guest" and "guest_nice" percentages are not accounted in "user" and "user_nice" percentages.

Warning: the first time this function is called with *interval* = `0.0` or `None` it will return a meaningless `0.0` value which you are supposed to ignore.

Changed in version 4.1.0: two new *interrupt* and *dpc* fields are returned on Windows.

`psutil.`**`cpu_count`**(*logical=True*)

Return the number of logical CPUs in the system (same as os.cpu_count() in Python 3.4) or `None` if undetermined. This number may not be equivalent to the number of CPUs the current process can actually use in case process CPU affinity has been changed or Linux cgroups are being used. The number of usable CPUs can be obtained with `len(psutil.Process().cpu_affinity())`. If *logical* is `False` return the number of physical cores only (hyper thread CPUs are excluded). On OpenBSD and NetBSD `psutil.cpu_count(logical=False)` always return `None`. Example on a system having 2 physical hyper-thread

CPU cores:

```
>>> import psutil
>>> psutil.cpu_count()
4
>>> psutil.cpu_count(logical=False)
2
```

Example returning the number of CPUs usable by the current process:

```
>>> len(psutil.Process().cpu_affinity())
1
```

psutil.**cpu_stats**()

Return various CPU statistics as a named tuple:

- **ctx_switches**: number of context switches (voluntary + involuntary) since boot.
- **interrupts**: number of interrupts since boot.
- **soft_interrupts**: number of software interrupts since boot. Always set to `0` on Windows and SunOS.
- **syscalls**: number of system calls since boot. Always set to `0` on Linux.

Example (Linux):

```
>>> import psutil
>>> psutil.cpu_stats()
scpustats(ctx_switches=20455687, interrupts=6598984, soft_interrupts=2134212,␣
↪syscalls=0)
```

New in version 4.1.0.

psutil.**cpu_freq**(*percpu=False*)

Return CPU frequency as a nameduple including *current*, *min* and *max* frequencies expressed in Mhz. On Linux *current* frequency reports the real-time value, on all other platforms it represents the nominal "fixed" value. If *percpu* is `True` and the system supports per-cpu frequency retrieval (Linux only) a list of frequencies is returned for each CPU, if not, a list with a single element is returned. If *min* and *max* cannot be determined they are set to `0`.

Example (Linux):

```
>>> import psutil
>>> psutil.cpu_freq()
scpufreq(current=931.42925, min=800.0, max=3500.0)
>>> psutil.cpu_freq(percpu=True)
[scpufreq(current=2394.945, min=800.0, max=3500.0),
 scpufreq(current=2236.812, min=800.0, max=3500.0),
 scpufreq(current=1703.609, min=800.0, max=3500.0),
 scpufreq(current=1754.289, min=800.0, max=3500.0)]
```

Availability: Linux, OSX, Windows

New in version 5.1.0.

Memory

psutil.**virtual_memory**()

Return statistics about system memory usage as a named tuple including the following fields, expressed in bytes.

Main metrics:

- **total**: total physical memory.
- **available**: the memory that can be given instantly to processes without the system going into swap. This is calculated by summing different memory values depending on the platform and it is supposed to be used to monitor actual memory usage in a cross platform fashion.

Other metrics:

- **used**: memory used, calculated differently depending on the platform and designed for informational purposes only. **total - free** does not necessarily match **used**.
- **free**: memory not being used at all (zeroed) that is readily available; note that this doesn't reflect the actual memory available (use **available** instead). **total - used** does not necessarily match **free**.
- **active** *(UNIX)*: memory currently in use or very recently used, and so it is in RAM.
- **inactive** *(UNIX)*: memory that is marked as not used.
- **buffers** *(Linux, BSD)*: cache for things like file system metadata.
- **cached** *(Linux, BSD)*: cache for various things.
- **shared** *(Linux, BSD)*: memory that may be simultaneously accessed by multiple processes.
- **wired** *(BSD, OSX)*: memory that is marked to always stay in RAM. It is never moved to disk.

The sum of **used** and **available** does not necessarily equal **total**. On Windows **available** and **free** are the same. See meminfo.py script providing an example on how to convert bytes in a human readable form.

Note: if you just want to know how much physical memory is left in a cross platform fashion simply rely on the **available** field.

```
>>> import psutil
>>> mem = psutil.virtual_memory()
>>> mem
svmem(total=10367352832, available=6472179712, percent=37.6, used=8186245120,␣
↪free=2181107712, active=4748992512, inactive=2758115328, buffers=790724608,␣
↪cached=3500347392, shared=787554304)
>>>
>>> THRESHOLD = 100 * 1024 * 1024  # 100MB
>>> if mem.available <= THRESHOLD:
...     print("warning")
...
>>>
```

Changed in version 4.2.0: added *shared* metrics on Linux.

Changed in version 4.4.0: *available* and *used* values on Linux are more precise and match "free" cmdline utility.

`psutil.swap_memory()`

Return system swap memory statistics as a named tuple including the following fields:

- **total**: total swap memory in bytes
- **used**: used swap memory in bytes
- **free**: free swap memory in bytes
- **percent**: the percentage usage calculated as `(total - available) / total * 100`
- **sin**: the number of bytes the system has swapped in from disk (cumulative)

- **sout**: the number of bytes the system has swapped out from disk (cumulative)

sin and **sout** on Windows are always set to `0`. See meminfo.py script providing an example on how to convert bytes in a human readable form.

```
>>> import psutil
>>> psutil.swap_memory()
sswap(total=2097147904L, used=886620160L, free=1210527744L, percent=42.3,␣
↪sin=1050411008, sout=1906720768)
```

Changed in version 5.2.3: on Linux this function relies on /proc fs instead of sysinfo() syscall so that it can be used in conjunction with *psutil.PROCFS_PATH* in order to retrieve memory info about Linux containers such as Docker and Heroku.

Disks

psutil.**disk_partitions**(*all=False*)

Return all mounted disk partitions as a list of named tuples including device, mount point and filesystem type, similarly to "df" command on UNIX. If *all* parameter is `False` it tries to distinguish and return physical devices only (e.g. hard disks, cd-rom drives, USB keys) and ignore all others (e.g. memory partitions such as /dev/shm). Note that this may not be fully reliable on all systems (e.g. on BSD this parameter is ignored). Named tuple's **fstype** field is a string which varies depending on the platform. On Linux it can be one of the values found in /proc/filesystems (e.g. `'ext3'` for an ext3 hard drive o `'iso9660'` for the CD-ROM drive). On Windows it is determined via GetDriveType and can be either `"removable"`, `"fixed"`, `"remote"`, `"cdrom"`, `"unmounted"` or `"ramdisk"`. On OSX and BSD it is retrieved via getfsstat(2). See disk_usage.py script providing an example usage.

```
>>> import psutil
>>> psutil.disk_partitions()
[sdiskpart(device='/dev/sda3', mountpoint='/', fstype='ext4', opts='rw,
↪errors=remount-ro'),
 sdiskpart(device='/dev/sda7', mountpoint='/home', fstype='ext4', opts='rw')]
```

psutil.**disk_usage**(*path*)

Return disk usage statistics about the partition which contains the given *path* as a named tuple including **total**, **used** and **free** space expressed in bytes, plus the **percentage** usage. OSError is raised if *path* does not exist. Starting from Python 3.3 this is also available as shutil.disk_usage(). See disk_usage.py script providing an example usage.

```
>>> import psutil
>>> psutil.disk_usage('/')
sdiskusage(total=21378641920, used=4809781248, free=15482871808, percent=22.5)
```

Note: UNIX usually reserves 5% of the total disk space for the root user. *total* and *used* fields on UNIX refer to the overall total and used space, whereas *free* represents the space available for the **user** and *percent* represents the **user** utilization (see source code). That is why *percent* value may look 5% bigger than what you would expect it to be. Also note that both 4 values match "df" cmdline utility.

Changed in version 4.3.0: *percent* value takes root reserved space into account.

psutil.**disk_io_counters**(*perdisk=False*, *nowrap=True*)

Return system-wide disk I/O statistics as a named tuple including the following fields:

- **read_count**: number of reads

- **write_count**: number of writes
- **read_bytes**: number of bytes read
- **write_bytes**: number of bytes written

Platform-specific fields:

- **read_time**: (all except *NetBSD* and *OpenBSD*) time spent reading from disk (in milliseconds)
- **write_time**: (all except *NetBSD* and *OpenBSD*) time spent writing to disk (in milliseconds)
- **busy_time**: (*Linux*, *FreeBSD*) time spent doing actual I/Os (in milliseconds)
- **read_merged_count** (*Linux*): number of merged reads (see iostat doc)
- **write_merged_count** (*Linux*): number of merged writes (see iostats doc)

If *perdisk* is `True` return the same information for every physical disk installed on the system as a dictionary with partition names as the keys and the named tuple described above as the values. See iotop.py for an example application. On some systems such as Linux, on a very busy or long-lived system, the numbers returned by the kernel may overflow and wrap (restart from zero). If *nowrap* is `True` psutil will detect and adjust those numbers across function calls and add "old value" to "new value" so that the returned numbers will always be increasing or remain the same, but never decrease. `disk_io_counters.cache_clear()` can be used to invalidate the *nowrap* cache.

```
>>> import psutil
>>> psutil.disk_io_counters()
sdiskio(read_count=8141, write_count=2431, read_bytes=290203, write_bytes=537676,␣
↪read_time=5868, write_time=94922)
>>>
>>> psutil.disk_io_counters(perdisk=True)
{'sda1': sdiskio(read_count=920, write_count=1, read_bytes=2933248, write_
↪bytes=512, read_time=6016, write_time=4),
 'sda2': sdiskio(read_count=18707, write_count=8830, read_bytes=6060, write_
↪bytes=3443, read_time=24585, write_time=1572),
 'sdb1': sdiskio(read_count=161, write_count=0, read_bytes=786432, write_bytes=0,␣
↪read_time=44, write_time=0)}
```

Note: on Windows `"diskperf -y"` command may need to be executed first otherwise this function won't find any disk.

Changed in version 5.3.0: numbers no longer wrap (restart from zero) across calls thanks to new *nowrap* argument.

Changed in version 4.0.0: added *busy_time* (Linux, FreeBSD), *read_merged_count* and *write_merged_count* (Linux) fields.

Changed in version 4.0.0: NetBSD no longer has *read_time* and *write_time* fields.

Network

`psutil.net_io_counters`(*pernic=False*)

Return system-wide network I/O statistics as a named tuple including the following attributes:

- **bytes_sent**: number of bytes sent
- **bytes_recv**: number of bytes received

- **packets_sent**: number of packets sent
- **packets_recv**: number of packets received
- **errin**: total number of errors while receiving
- **errout**: total number of errors while sending
- **dropin**: total number of incoming packets which were dropped
- **dropout**: total number of outgoing packets which were dropped (always 0 on OSX and BSD)

If *pernic* is `True` return the same information for every network interface installed on the system as a dictionary with network interface names as the keys and the named tuple described above as the values. On some systems such as Linux, on a very busy or long-lived system, the numbers returned by the kernel may overflow and wrap (restart from zero). If *nowrap* is `True` psutil will detect and adjust those numbers across function calls and add "old value" to "new value" so that the returned numbers will always be increasing or remain the same, but never decrease. `net_io_counters.cache_clear()` can be used to invalidate the *nowrap* cache.

```
>>> import psutil
>>> psutil.net_io_counters()
snetio(bytes_sent=14508483, bytes_recv=62749361, packets_sent=84311, packets_
↪recv=94888, errin=0, errout=0, dropin=0, dropout=0)
>>>
>>> psutil.net_io_counters(pernic=True)
{'lo': snetio(bytes_sent=547971, bytes_recv=547971, packets_sent=5075, packets_
↪recv=5075, errin=0, errout=0, dropin=0, dropout=0),
'wlan0': snetio(bytes_sent=13921765, bytes_recv=62162574, packets_sent=79097,␣
↪packets_recv=89648, errin=0, errout=0, dropin=0, dropout=0)}
```

Also see nettop.py and ifconfig.py for an example application.

Changed in version 5.3.0: numbers no longer wrap (restart from zero) across calls thanks to new *nowrap* argument.

`psutil.`**`net_connections`**(*kind='inet'*)

Return system-wide socket connections as a list of named tuples. Every named tuple provides 7 attributes:

- **fd**: the socket file descriptor. If the connection refers to the current process this may be passed to socket.fromfd() to obtain a usable socket object. On Windows and SunOS this is always set to `-1`.
- **family**: the address family, either AF_INET, AF_INET6 or AF_UNIX.
- **type**: the address type, either SOCK_STREAM or SOCK_DGRAM.
- **laddr**: the local address as a `(ip, port)` named tuple or a `path` in case of AF_UNIX sockets. For UNIX sockets see notes below.
- **raddr**: the remote address as a `(ip, port)` named tuple or an absolute `path` in case of UNIX sockets. When the remote endpoint is not connected you'll get an empty tuple (AF_INET*) or `""` (AF_UNIX). For UNIX sockets see notes below.
- **status**: represents the status of a TCP connection. The return value is one of the *psutil.CONN_** constants (a string). For UDP and UNIX sockets this is always going to be *psutil.CONN_NONE*.
- **pid**: the PID of the process which opened the socket, if retrievable, else `None`. On some platforms (e.g. Linux) the availability of this field changes depending on process privileges (root is needed).

The *kind* parameter is a string which filters for connections matching the following criteria:

Kind value	Connections using
"inet"	IPv4 and IPv6
"inet4"	IPv4
"inet6"	IPv6
"tcp"	TCP
"tcp4"	TCP over IPv4
"tcp6"	TCP over IPv6
"udp"	UDP
"udp4"	UDP over IPv4
"udp6"	UDP over IPv6
"unix"	UNIX socket (both UDP and TCP protocols)
"all"	the sum of all the possible families and protocols

On OSX this function requires root privileges. To get per-process connections use *Process.connections()*. Also, see netstat.py sample script. Example:

```
>>> import psutil
>>> psutil.net_connections()
[pconn(fd=115, family=<AddressFamily.AF_INET: 2>, type=<SocketType.SOCK_STREAM: 1>
↪, laddr=addr(ip='10.0.0.1', port=48776), raddr=addr(ip='93.186.135.91',␣
↪port=80), status='ESTABLISHED', pid=1254),
 pconn(fd=117, family=<AddressFamily.AF_INET: 2>, type=<SocketType.SOCK_STREAM: 1>
↪, laddr=addr(ip='10.0.0.1', port=43761), raddr=addr(ip='72.14.234.100',␣
↪port=80), status='CLOSING', pid=2987),
 pconn(fd=-1, family=<AddressFamily.AF_INET: 2>, type=<SocketType.SOCK_STREAM: 1>,
↪ laddr=addr(ip='10.0.0.1', port=60759), raddr=addr(ip='72.14.234.104', port=80),
↪ status='ESTABLISHED', pid=None),
 pconn(fd=-1, family=<AddressFamily.AF_INET: 2>, type=<SocketType.SOCK_STREAM: 1>,
↪ laddr=addr(ip='10.0.0.1', port=51314), raddr=addr(ip='72.14.234.83', port=443),
↪ status='SYN_SENT', pid=None)
 ...]
```

Note: (OSX) *psutil.AccessDenied* is always raised unless running as root. This is a limitation of the OS and `lsof` does the same.

Note: (Solaris) UNIX sockets are not supported.

Note: (Linux, FreeBSD) "raddr" field for UNIX sockets is always set to "". This is a limitation of the OS.

Note: (OpenBSD) "laddr" and "raddr" fields for UNIX sockets are always set to "". This is a limitation of the OS.

New in version 2.1.0.

Changed in version 5.3.0: : socket "fd" is now set for real instead of being `-1`.

Changed in version 5.3.0: : "laddr" and "raddr" are named tuples.

`psutil.`**`net_if_addrs`**`()`

Return the addresses associated to each NIC (network interface card) installed on the system as a dictionary

whose keys are the NIC names and value is a list of named tuples for each address assigned to the NIC. Each named tuple includes 5 fields:

- **family**: the address family, either AF_INET, AF_INET6 or *psutil.AF_LINK*, which refers to a MAC address.
- **address**: the primary NIC address (always set).
- **netmask**: the netmask address (may be `None`).
- **broadcast**: the broadcast address (may be `None`).
- **ptp**: stands for "point to point"; it's the destination address on a point to point interface (typically a VPN). *broadcast* and *ptp* are mutually exclusive. May be `None`.

Example:

```
>>> import psutil
>>> psutil.net_if_addrs()
{'lo': [snic(family=<AddressFamily.AF_INET: 2>, address='127.0.0.1', netmask='255.
↪0.0.0', broadcast='127.0.0.1', ptp=None),
        snic(family=<AddressFamily.AF_INET6: 10>, address='::1', netmask=
↪'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff', broadcast=None, ptp=None),
        snic(family=<AddressFamily.AF_LINK: 17>, address='00:00:00:00:00:00',␣
↪netmask=None, broadcast='00:00:00:00:00:00', ptp=None)],
 'wlan0': [snic(family=<AddressFamily.AF_INET: 2>, address='192.168.1.3', netmask=
↪'255.255.255.0', broadcast='192.168.1.255', ptp=None),
          snic(family=<AddressFamily.AF_INET6: 10>, address=
↪'fe80::c685:8ff:fe45:641%wlan0', netmask='ffff:ffff:ffff:ffff::',␣
↪broadcast=None, ptp=None),
          snic(family=<AddressFamily.AF_LINK: 17>, address='c4:85:08:45:06:41',␣
↪netmask=None, broadcast='ff:ff:ff:ff:ff:ff', ptp=None)]}
>>>
```

See also nettop.py and ifconfig.py for an example application.

Note: if you're interested in others families (e.g. AF_BLUETOOTH) you can use the more powerful netifaces extension.

Note: you can have more than one address of the same family associated with each interface (that's why dict values are lists).

Note: *broadcast* and *ptp* are not supported on Windows and are always `None`.

New in version 3.0.0.

Changed in version 3.2.0: *ptp* field was added.

Changed in version 4.4.0: added support for *netmask* field on Windows which is no longer `None`.

`psutil.`**`net_if_stats`**`()`

Return information about each NIC (network interface card) installed on the system as a dictionary whose keys are the NIC names and value is a named tuple with the following fields:

- **isup**: a bool indicating whether the NIC is up and running.

- **duplex**: the duplex communication type; it can be either *NIC_DUPLEX_FULL*, *NIC_DUPLEX_HALF* or *NIC_DUPLEX_UNKNOWN*.
- **speed**: the NIC speed expressed in mega bits (MB), if it can't be determined (e.g. 'localhost') it will be set to `0`.
- **mtu**: NIC's maximum transmission unit expressed in bytes.

Example:

```
>>> import psutil
>>> psutil.net_if_stats()
{'eth0': snicstats(isup=True, duplex=<NicDuplex.NIC_DUPLEX_FULL: 2>, speed=100,␣
↪mtu=1500),
 'lo': snicstats(isup=True, duplex=<NicDuplex.NIC_DUPLEX_UNKNOWN: 0>, speed=0,␣
↪mtu=65536)}
```

Also see nettop.py and ifconfig.py for an example application.

New in version 3.0.0.

Sensors

`psutil.`**`sensors_temperatures`**(*fahrenheit=False*)

Return hardware temperatures. Each entry is a named tuple representing a certain hardware temperature sensor (it may be a CPU, an hard disk or something else, depending on the OS and its configuration). All temperatures are expressed in celsius unless *fahrenheit* is set to `True`. If sensors are not supported by the OS an empty dict is returned. Example:

```
>>> import psutil
>>> psutil.sensors_temperatures()
{'acpitz': [shwtemp(label='', current=47.0, high=103.0, critical=103.0)],
 'asus': [shwtemp(label='', current=47.0, high=None, critical=None)],
 'coretemp': [shwtemp(label='Physical id 0', current=52.0, high=100.0,␣
↪critical=100.0),
              shwtemp(label='Core 0', current=45.0, high=100.0, critical=100.0),
              shwtemp(label='Core 1', current=52.0, high=100.0, critical=100.0),
              shwtemp(label='Core 2', current=45.0, high=100.0, critical=100.0),
              shwtemp(label='Core 3', current=47.0, high=100.0, critical=100.0)]}
```

See also temperatures.py and sensors.py for an example application.

Availability: Linux

New in version 5.1.0.

> **Warning:** This API is experimental. Backward incompatible changes may occur if deemed necessary.

`psutil.`**`sensors_fans`**`()`

Return hardware fans speed. Each entry is a named tuple representing a certain hardware sensor fan. Fan speed is expressed in RPM (rounds per minute). If sensors are not supported by the OS an empty dict is returned. Example:

```
>>> import psutil
>>> psutil.sensors_fans()
{'asus': [sfan(label='cpu_fan', current=3200)]}
```

See also fans.py and sensors.py for an example application.

Availability: Linux

New in version 5.2.0.

> **Warning:** This API is experimental. Backward incompatible changes may occur if deemed necessary.

psutil.**sensors_battery**()
Return battery status information as a named tuple including the following values. If no battery is installed or metrics can't be determined `None` is returned.

- **percent**: battery power left as a percentage.
- **secsleft**: a rough approximation of how many seconds are left before the battery runs out of power. If the AC power cable is connected this is set to *psutil.POWER_TIME_UNLIMITED*. If it can't be determined it is set to *psutil.POWER_TIME_UNKNOWN*.
- **power_plugged**: `True` if the AC power cable is connected, `False` if not or `None` if it can't be determined.

Example:

```
>>> import psutil
>>>
>>> def secs2hours(secs):
...     mm, ss = divmod(secs, 60)
...     hh, mm = divmod(mm, 60)
...     return "%d:%02d:%02d" % (hh, mm, ss)
...
>>> battery = psutil.sensors_battery()
>>> battery
sbattery(percent=93, secsleft=16628, power_plugged=False)
>>> print("charge = %s%%, time left = %s" % (batt.percent, secs2hours(batt.
↪secsleft)))
charge = 93%, time left = 4:37:08
```

See also battery.py and sensors.py for an example application.

Availability: Linux, Windows, FreeBSD

New in version 5.1.0.

> **Warning:** This API is experimental. Backward incompatible changes may occur if deemed necessary.

Other system info

psutil.**boot_time**()
Return the system boot time expressed in seconds since the epoch. Example:

```
>>> import psutil, datetime
>>> psutil.boot_time()
1389563460.0
```

```
>>> datetime.datetime.fromtimestamp(psutil.boot_time()).strftime("%Y-%m-%d %H:%M:
↪%S")
'2014-01-12 22:51:00'
```

Note: on Windows this function may return a time which is off by 1 second if it's used across different processes (see issue #1007).

psutil.**users**()

Return users currently connected on the system as a list of named tuples including the following fields:

- **user**: the name of the user.
- **terminal**: the tty or pseudo-tty associated with the user, if any, else None.
- **host**: the host name associated with the entry, if any.
- **started**: the creation time as a floating point number expressed in seconds since the epoch.
- **pid**: the PID of the login process (like sshd, tmux, gdm-session-worker, ...). On Windows and OpenBSD this is always set to None.

Example:

```
>>> import psutil
>>> psutil.users()
[suser(name='giampaolo', terminal='pts/2', host='localhost', started=1340737536.0,
↪ pid=1352),
 suser(name='giampaolo', terminal='pts/3', host='localhost', started=1340737792.0,
↪ pid=1788)]
```

Changed in version 5.3.0: added "pid" field

CHAPTER 5

Processes

Functions

psutil.**pids**()

Return a list of current running PIDs. To iterate over all processes and avoid race conditions *process_iter()* should be preferred.

```
>>> import psutil
>>> psutil.pids()
[1, 2, 3, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, ..., 32498]
```

psutil.**process_iter**(*attrs=None*, *ad_value=None*)

Return an iterator yielding a *Process* class instance for all running processes on the local machine. Every instance is only created once and then cached into an internal table which is updated every time an element is yielded. Cached *Process* instances are checked for identity so that you're safe in case a PID has been reused by another process, in which case the cached instance is updated. This is should be preferred over *psutil.pids()* for iterating over processes. Sorting order in which processes are returned is based on their PID. *attrs* and *ad_value* have the same meaning as in *Process.as_dict()*. If *attrs* is specified *Process.as_dict()* is called and the resulting dict is stored as a `info` attribute which is attached to the returned *Process* instance. If *attrs* is an empty list it will retrieve all process info (slow). Example usage:

```
>>> import psutil
>>> for proc in psutil.process_iter():
...     try:
...         pinfo = proc.as_dict(attrs=['pid', 'name', 'username'])
...     except psutil.NoSuchProcess:
...         pass
...     else:
...         print(pinfo)
...
{'name': 'systemd', 'pid': 1, 'username': 'root'}
{'name': 'kthreadd', 'pid': 2, 'username': 'root'}
{'name': 'ksoftirqd/0', 'pid': 3, 'username': 'root'}
...
```

More compact version using *attrs* parameter:

```
>>> import psutil
>>> for proc in psutil.process_iter(attrs=['pid', 'name', 'username']):
...     print(proc.info)
...
{'name': 'systemd', 'pid': 1, 'username': 'root'}
{'name': 'kthreadd', 'pid': 2, 'username': 'root'}
{'name': 'ksoftirqd/0', 'pid': 3, 'username': 'root'}
...
```

Example of a dict comprehensions to create a `{pid: info, ...}` data structure:

```
>>> import psutil
>>> procs = {p.pid: p.info for p in psutil.process_iter(attrs=['name', 'username
↪'])}
>>> procs
{1: {'name': 'systemd', 'username': 'root'},
 2: {'name': 'kthreadd', 'username': 'root'},
 3: {'name': 'ksoftirqd/0', 'username': 'root'},
 ...}
```

Example showing how to filter processes by name:

```
>>> import psutil
>>> [p.info for p in psutil.process_iter(attrs=['pid', 'name']) if 'python' in p.
↪info['name']]
[{'name': 'python3', 'pid': 21947},
 {'name': 'python', 'pid': 23835}]
```

See also *process filtering* section for more examples.

Changed in version 5.3.0: added "attrs" and "ad_value" parameters.

`psutil.`**`pid_exists`**(*pid*)

Check whether the given PID exists in the current process list. This is faster than doing `pid in psutil.pids()` and should be preferred.

`psutil.`**`wait_procs`**(*procs*, *timeout=None*, *callback=None*)

Convenience function which waits for a list of *Process* instances to terminate. Return a `(gone, alive)` tuple indicating which processes are gone and which ones are still alive. The *gone* ones will have a new *returncode* attribute indicating process exit status (will be `None` for processes which are not our children). `callback` is a function which gets called when one of the processes being waited on is terminated and a *Process* instance is passed as callback argument). This tunction will return as soon as all processes terminate or when *timeout* occurs, if specified. Differently from *Process.wait()* it will not raise *TimeoutExpired* if timeout occurs. A typical use case may be:

- send SIGTERM to a list of processes
- give them some time to terminate
- send SIGKILL to those ones which are still alive

Example which terminates and waits all the children of this process:

```
import psutil

def on_terminate(proc):
    print("process {} terminated with exit code {}".format(proc, proc.returncode))
```

```
procs = psutil.Process().children()
for p in procs:
    p.terminate()
gone, alive = psutil.wait_procs(procs, timeout=3, callback=on_terminate)
for p in alive:
    p.kill()
```

Exceptions

class psutil.**Error**
Base exception class. All other exceptions inherit from this one.

class psutil.**NoSuchProcess**(*pid*, *name=None*, *msg=None*)
Raised by *Process* class methods when no process with the given *pid* is found in the current process list or when a process no longer exists. *name* is the name the process had before disappearing and gets set only if *Process.name()* was previously called.

class psutil.**ZombieProcess**(*pid*, *name=None*, *ppid=None*, *msg=None*)
This may be raised by *Process* class methods when querying a zombie process on UNIX (Windows doesn't have zombie processes). Depending on the method called the OS may be able to succeed in retrieving the process information or not. Note: this is a subclass of *NoSuchProcess* so if you're not interested in retrieving zombies (e.g. when using *process_iter()*) you can ignore this exception and just catch *NoSuchProcess*.

New in version 3.0.0.

class psutil.**AccessDenied**(*pid=None*, *name=None*, *msg=None*)
Raised by *Process* class methods when permission to perform an action is denied. "name" is the name of the process (may be None).

class psutil.**TimeoutExpired**(*seconds*, *pid=None*, *name=None*, *msg=None*)
Raised by *Process.wait()* if timeout expires and process is still alive.

Process class

class psutil.**Process**(*pid=None*)
Represents an OS process with the given *pid*. If *pid* is omitted current process *pid* (os.getpid()) is used. Raise *NoSuchProcess* if *pid* does not exist. On Linux *pid* can also refer to a thread ID (the *id* field returned by *threads()* method). When accessing methods of this class always be prepared to catch *NoSuchProcess*, *ZombieProcess* and *AccessDenied* exceptions. hash() builtin can be used against instances of this class in order to identify a process univocally over time (the hash is determined by mixing process PID and creation time). As such it can also be used with set()s.

Note: In order to efficiently fetch more than one information about the process at the same time, make sure to use either *as_dict()* or *oneshot()* context manager.

Warning: the way this class is bound to a process is via its **PID**. That means that if the *Process* instance is old enough and the PID has been reused in the meantime you might end up interacting with another process. The only exceptions for which process identity is preemptively checked (via PID + creation time) and guaranteed are for *nice()* (set), *ionice()* (set), *cpu_affinity()* (set), *rlimit()* (set), *children()*, *parent()*, *suspend()* *resume()*, *send_signal()*, *terminate()*, and *kill()* methods. To

prevent this problem for all other methods you can use *is_running()* before querying the process or use *process_iter()* in case you're iterating over all processes.

oneshot()

Utility context manager which considerably speeds up the retrieval of multiple process information at the same time. Internally different process info (e.g. *name()*, *ppid()*, *uids()*, *create_time()*, ...) may be fetched by using the same routine, but only one value is returned and the others are discarded. When using this context manager the internal routine is executed once (in the example below on *name()*) the value of interest is returned and the others are cached. The subsequent calls sharing the same internal routine will return the cached value. The cache is cleared when exiting the context manager block. The advice is to use this every time you retrieve more than one information about the process. If you're lucky, you'll get a hell of a speedup. Example:

```
>>> import psutil
>>> p = psutil.Process()
>>> with p.oneshot():
...     p.name()  # execute internal routine once collecting multiple info
...     p.cpu_times()  # return cached value
...     p.cpu_percent()  # return cached value
...     p.create_time()  # return cached value
...     p.ppid()  # return cached value
...     p.status()  # return cached value
...
>>>
```

Here's a list of methods which can take advantage of the speedup depending on what platform you're on. In the table below horizontal emtpy rows indicate what process methods can be efficiently grouped together internally. The last column (speedup) shows an approximation of the speedup you can get if you call all the methods together (best case scenario).

Linux	Windows	OSX	BSD	SunOS
cpu_num()	*cpu_percent()*	*cpu_percent()*	*cpu_num()*	*name()*
cpu_percent()	*cpu_times()*	*cpu_times()*	*cpu_percent()*	*cmdline()*
cpu_times()	*io_counters()*	*memory_info()*	*cpu_times()*	*create_time()*
create_time()	*ionice()*	*memory_percent()*	*create_time()*	
name()	*memory_info()*	*num_ctx_switches()*	*gids()*	*memory_info()*
ppid()	*nice()*	*num_threads()*	*io_counters()*	*memory_percent()*
status()	*memory_maps()*		*name()*	*nice()*
terminal()	*num_ctx_switches()*	*create_time()*	*memory_info()*	*num_threads()*
	num_handles()	*gids()*	*memory_percent()*	*ppid()*
gids()	*num_threads()*	*name()*	*num_ctx_switches()*	*status()*
num_ctx_switches()	*username()*	*ppid()*	*ppid()*	*terminal()*
num_threads()		*status()*	*status()*	
uids()		*terminal()*	*terminal()*	*gids()*
username()		*uids()*	*uids()*	*uids()*
		username()	*username()*	*username()*
memory_full_info()				
memory_maps()				
speedup: +2.6x	*speedup: +1.8x / +6.5x*	*speedup: +1.9x*	*speedup: +2.0x*	*speedup: +1.3x*

New in version 5.0.0.

pid

The process PID. This is the only (read-only) attribute of the class.

ppid()
The process parent PID. On Windows the return value is cached after first call. Not on POSIX because ppid may change if process becomes a zombie. See also *parent()* method.

name()
The process name. On Windows the return value is cached after first call. Not on POSIX because the process name may change. See also how to *find a process by name*.

exe()
The process executable as an absolute path. On some systems this may also be an empty string. The return value is cached after first call.

```
>>> import psutil
>>> psutil.Process().exe()
'/usr/bin/python2.7'
```

cmdline()
The command line this process has been called with as a list of strings. The return value is not cached because the cmdline of a process may change.

```
>>> import psutil
>>> psutil.Process().cmdline()
['python', 'manage.py', 'runserver']
```

environ()
The environment variables of the process as a dict. Note: this might not reflect changes made after the process started.

```
>>> import psutil
>>> psutil.Process().environ()
{'LC_NUMERIC': 'it_IT.UTF-8', 'QT_QPA_PLATFORMTHEME': 'appmenu-qt5', 'IM_
↪CONFIG_PHASE': '1', 'XDG_GREETER_DATA_DIR': '/var/lib/lightdm-data/giampaolo
↪', 'GNOME_DESKTOP_SESSION_ID': 'this-is-deprecated', 'XDG_CURRENT_DESKTOP':
↪'Unity', 'UPSTART_EVENTS': 'started starting', 'GNOME_KEYRING_PID': '',
↪'XDG_VTNR': '7', 'QT_IM_MODULE': 'ibus', 'LOGNAME': 'giampaolo', 'USER':
↪'giampaolo', 'PATH': '/home/giampaolo/bin:/usr/local/sbin:/usr/local/bin:/
↪usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/home/
↪giampaolo/svn/sysconf/bin', 'LC_PAPER': 'it_IT.UTF-8', 'GNOME_KEYRING_
↪CONTROL': '', 'GTK_IM_MODULE': 'ibus', 'DISPLAY': ':0', 'LANG': 'en_US.UTF-8
↪', 'LESS_TERMCAP_se': '\x1b[0m', 'TERM': 'xterm-256color', 'SHELL': '/bin/
↪bash', 'XDG_SESSION_PATH': '/org/freedesktop/DisplayManager/Session0',
↪'XAUTHORITY': '/home/giampaolo/.Xauthority', 'LANGUAGE': 'en_US', 'COMPIZ_
↪CONFIG_PROFILE': 'ubuntu', 'LC_MONETARY': 'it_IT.UTF-8', 'QT_LINUX_
↪ACCESSIBILITY_ALWAYS_ON': '1', 'LESS_TERMCAP_me': '\x1b[0m', 'LESS_TERMCAP_
↪md': '\x1b[01;38;5;74m', 'LESS_TERMCAP_mb': '\x1b[01;31m', 'HISTSIZE':
↪'100000', 'UPSTART_INSTANCE': '', 'CLUTTER_IM_MODULE': 'xim', 'WINDOWID':
↪'58786407', 'EDITOR': 'vim', 'SESSIONTYPE': 'gnome-session', 'XMODIFIERS':
↪'@im=ibus', 'GPG_AGENT_INFO': '/home/giampaolo/.gnupg/S.gpg-agent:0:1',
↪'HOME': '/home/giampaolo', 'HISTFILESIZE': '100000', 'QT4_IM_MODULE': 'xim',
↪ 'GTK2_MODULES': 'overlay-scrollbar', 'XDG_SESSION_DESKTOP': 'ubuntu',
↪'SHLVL': '1', 'XDG_RUNTIME_DIR': '/run/user/1000', 'INSTANCE': 'Unity', 'LC_
↪ADDRESS': 'it_IT.UTF-8', 'SSH_AUTH_SOCK': '/run/user/1000/keyring/ssh',
↪'VTE_VERSION': '4205', 'GDMSESSION': 'ubuntu', 'MANDATORY_PATH': '/usr/
↪share/gconf/ubuntu.mandatory.path', 'VISUAL': 'vim', 'DESKTOP_SESSION':
↪'ubuntu', 'QT_ACCESSIBILITY': '1', 'XDG_SEAT_PATH': '/org/freedesktop/
↪DisplayManager/Seat0', 'LESSCLOSE': '/usr/bin/lesspipe %s %s', 'LESSOPEN':
↪'| /usr/bin/lesspipe %s', 'XDG_SESSION_ID': 'c2', 'DBUS_SESSION_BUS_ADDRESS
↪': 'unix:abstract=/tmp/dbus-9GAJpvnt8r', '_': '/usr/bin/python', 'DEFAULTS_
↪PATH': '/usr/share/gconf/ubuntu.default.path', 'LC_IDENTIFICATION': 'it_IT.
↪UTF-8', 'LESS_TERMCAP_ue': '\x1b[0m', 'UPSTART_SESSION': 'unix:abstract=/
↪com/ubuntu/upstart-session/1000/1294', 'XDG_CONFIG_DIRS': '/etc/xdg/xdg-
↪Unity:/usr/share/upstart/xdg:/etc/xdg', 'GTK_MODULES': 'gail:atk-
↪bridge:unity-gtk-module', 'XDG_SESSION_TYPE': 'x11', 'PYTHONSTARTUP': '/
↪home/giampaolo/.pythonstart', 'LC_NAME': 'it_IT.UTF-8', 'OLDPWD': '/home/
↪giampaolo/svn/curio_giampaolo/tests', 'GDM_LANG': 'en_US', 'LC_TELEPHONE':
↪'it_IT.UTF-8', 'HISTCONTROL': 'ignoredups:erasedups', 'LC_MEASUREMENT': 'it_
```

Availability: Linux, OSX, Windows, SunOS

New in version 4.0.0.

Changed in version 5.3.0:: added SunOS support

create_time()
The process creation time as a floating point number expressed in seconds since the epoch, in UTC. The return value is cached after first call.

```
>>> import psutil, datetime
>>> p = psutil.Process()
>>> p.create_time()
1307289803.47
>>> datetime.datetime.fromtimestamp(p.create_time()).strftime("%Y-%m-%d %H:%M:
↪%S")
'2011-03-05 18:03:52'
```

as_dict(*attrs=None*, *ad_value=None*)
Utility method retrieving multiple process information as a dictionary. If *attrs* is specified it must be a list of strings reflecting available *Process* class's attribute names (e.g. `['cpu_times', 'name']`), else all public (read only) attributes are assumed. *ad_value* is the value which gets assigned to a dict key in case *AccessDenied* or *ZombieProcess* exception is raised when retrieving that particular process information. Internally, *as_dict()* uses *oneshot()* context manager so there's no need you use it also.

```
>>> import psutil
>>> p = psutil.Process()
>>> p.as_dict(attrs=['pid', 'name', 'username'])
{'username': 'giampaolo', 'pid': 12366, 'name': 'python'}
```

Changed in version 3.0.0: *ad_value* is used also when incurring into *ZombieProcess* exception, not only *AccessDenied*

Changed in version 4.5.0: *as_dict()* is considerably faster thanks to *oneshot()* context manager.

parent()
Utility method which returns the parent process as a *Process* object preemptively checking whether PID has been reused. If no parent PID is known return `None`. See also *ppid()* method.

status()
The current process status as a string. The returned string is one of the *psutil.STATUS_** constants.

cwd()
The process current working directory as an absolute path.

username()
The name of the user that owns the process. On UNIX this is calculated by using real process uid.

uids()
The real, effective and saved user ids of this process as a named tuple. This is the same as os.getresuid() but can be used for any process PID.

Availability: UNIX

gids()
The real, effective and saved group ids of this process as a named tuple. This is the same as os.getresgid() but can be used for any process PID.

Availability: UNIX

terminal()
The terminal associated with this process, if any, else `None`. This is similar to "tty" command but can be used for any process PID.

Availability: UNIX

nice(*value=None*)
Get or set process niceness (priority). On UNIX this is a number which usually goes from `-20` to `20`. The higher the nice value, the lower the priority of the process.

```
>>> import psutil
>>> p = psutil.Process()
>>> p.nice(10)  # set
>>> p.nice()  # get
10
>>>
```

Starting from Python 3.3 this functionality is also available as os.getpriority() and os.setpriority() (UNIX only). On Windows this is implemented via GetPriorityClass and SetPriorityClass Windows APIs and *value* is one of the *psutil.*_PRIORITY_CLASS* constants reflecting the MSDN documentation. Example which increases process priority on Windows:

```
>>> p.nice(psutil.HIGH_PRIORITY_CLASS)
```

ionice(*ioclass=None*, *value=None*)
Get or set process I/O niceness (priority). On Linux *ioclass* is one of the *psutil.IOPRIO_CLASS_** constants. *value* is a number which goes from `0` to `7`. The higher the value, the lower the I/O priority of the process. On Windows only *ioclass* is used and it can be set to `2` (normal), `1` (low) or `0` (very low). The example below sets IDLE priority class for the current process, meaning it will only get I/O time when no other process needs the disk:

```
>>> import psutil
>>> p = psutil.Process()
>>> p.ionice(psutil.IOPRIO_CLASS_IDLE)  # set
>>> p.ionice()  # get
pionice(ioclass=<IOPriority.IOPRIO_CLASS_IDLE: 3>, value=0)
>>>
```

On Windows only *ioclass* is used and it can be set to `2` (normal), `1` (low) or `0` (very low). Also it returns an integer instead of a named tuple.

Availability: Linux and Windows > Vista

Changed in version 3.0.0: on Python >= 3.4 the returned `ioclass` constant is an enum instead of a plain integer.

rlimit(*resource*, *limits=None*)
Get or set process resource limits (see man prlimit). *resource* is one of the *psutil.RLIMIT_** constants. *limits* is a `(soft, hard)` tuple. This is the same as resource.getrlimit() and resource.setrlimit() but can be used for any process PID, not only os.getpid(). For get, return value is a `(soft, hard)` tuple. Each value may be either and integer or *psutil.RLIMIT_**. Example:

```
>>> import psutil
>>> p = psutil.Process()
>>> # process may open no more than 128 file descriptors
>>> p.rlimit(psutil.RLIMIT_NOFILE, (128, 128))
>>> # process may create files no bigger than 1024 bytes
>>> p.rlimit(psutil.RLIMIT_FSIZE, (1024, 1024))
>>> # get
```

```
>>> p.rlimit(psutil.RLIMIT_FSIZE)
(1024, 1024)
>>>
```

Availability: Linux

io_counters()

Return process I/O statistics as a named tuple. For Linux you can refer to /proc filesysem documentation.

- **read_count**: the number of read operations performed (cumulative). This is supposed to count the number of read-related syscalls such as `read()` and `pread()` on UNIX.
- **write_count**: the number of write operations performed (cumulative). This is supposed to count the number of write-related syscalls such as `write()` and `pwrite()` on UNIX.
- **read_bytes**: the number of bytes read (cumulative). Always `-1` on BSD.
- **write_bytes**: the number of bytes written (cumulative). Always `-1` on BSD.

Linux specific:

- **read_chars** *(Linux)*: the amount of bytes which this process passed to `read()` and `pread()` syscalls (cumulative). Differently from *read_bytes* it doesn't care whether or not actual physical disk I/O occurred.
- **write_chars** *(Linux)*: the amount of bytes which this process passed to `write()` and `pwrite()` syscalls (cumulative). Differently from *write_bytes* it doesn't care whether or not actual physical disk I/O occurred.

Windows specific:

- **other_count** *(Windows)*: the number of I/O operations performed other than read and write operations.
- **other_bytes** *(Windows)*: the number of bytes transferred during operations other than read and write operations.

```
>>> import psutil
>>> p = psutil.Process()
>>> p.io_counters()
pio(read_count=454556, write_count=3456, read_bytes=110592, write_bytes=0,␣
↪read_chars=769931, write_chars=203)
```

Availability: all platforms except OSX and Solaris

Changed in version 5.2.0: added *read_chars* and *write_chars* on Linux; added *other_count* and *other_bytes* on Windows.

num_ctx_switches()

The number voluntary and involuntary context switches performed by this process (cumulative).

num_fds()

The number of file descriptors currently opened by this process (non cumulative).

Availability: UNIX

num_handles()

The number of handles currently used by this process (non cumulative).

Availability: Windows

num_threads()

The number of threads currently used by this process (non cumulative).

threads()
Return threads opened by process as a list of named tuples including thread id and thread CPU times (user/system). On OpenBSD this method requires root privileges.

cpu_times()
Return a *(user, system, children_user, children_system)* named tuple representing the accumulated process time, in seconds (see explanation). On Windows and OSX only *user* and *system* are filled, the others are set to `0`. This is similar to os.times() but can be used for any process PID.

Changed in version 4.1.0: return two extra fields: *children_user* and *children_system*.

cpu_percent(*interval=None*)
Return a float representing the process CPU utilization as a percentage which can also be `> 100.0` in case of a process running multiple threads on different CPUs. When *interval* is `> 0.0` compares process times to system CPU times elapsed before and after the interval (blocking). When interval is `0.0` or `None` compares process times to system CPU times elapsed since last call, returning immediately. That means the first time this is called it will return a meaningless `0.0` value which you are supposed to ignore. In this case is recommended for accuracy that this function be called a second time with at least `0.1` seconds between calls. Example:

```
>>> import psutil
>>> p = psutil.Process()
>>> # blocking
>>> p.cpu_percent(interval=1)
2.0
>>> # non-blocking (percentage since last call)
>>> p.cpu_percent(interval=None)
2.9
```

Note: the returned value can be > 100.0 in case of a process running multiple threads on different CPU cores.

Note: the returned value is explicitly *not* split evenly between all available CPUs (differently from *psutil.cpu_percent()*). This means that a busy loop process running on a system with 2 logical CPUs will be reported as having 100% CPU utilization instead of 50%. This was done in order to be consistent with `top` UNIX utility and also to make it easier to identify processes hogging CPU resources independently from the number of CPUs. It must be noted that `taskmgr.exe` on Windows does not behave like this (it would report 50% usage instead). To emulate Windows `taskmgr.exe` behavior you can do: `p.cpu_percent() / psutil.cpu_count()`.

Warning: the first time this method is called with interval = `0.0` or `None` it will return a meaningless `0.0` value which you are supposed to ignore.

cpu_affinity(*cpus=None*)
Get or set process current CPU affinity. CPU affinity consists in telling the OS to run a process on a limited set of CPUs only (on Linux cmdline, `taskset` command is typically used). If no argument is passed it returns the current CPU affinity as a list of integers. If passed it must be a list of integers specifying the new CPUs affinity. If an empty list is passed all eligible CPUs are assumed (and set). On some systems such as Linux this may not necessarily mean all available logical CPUs as in `list(range(psutil.cpu_count()))`.

```
>>> import psutil
>>> psutil.cpu_count()
4
>>> p = psutil.Process()
>>> # get
>>> p.cpu_affinity()
[0, 1, 2, 3]
>>> # set; from now on, process will run on CPU #0 and #1 only
>>> p.cpu_affinity([0, 1])
>>> p.cpu_affinity()
[0, 1]
>>> # reset affinity against all eligible CPUs
>>> p.cpu_affinity([])
```

Availability: Linux, Windows, FreeBSD

Changed in version 2.2.0: added support for FreeBSD

Changed in version 5.1.0: an empty list can be passed to set affinity against all eligible CPUs.

cpu_num()

Return what CPU this process is currently running on. The returned number should be <= *psutil.cpu_count()*. On FreeBSD certain kernel process may return `-1`. It may be used in conjunction with `psutil.cpu_percent(percpu=True)` to observe the system workload distributed across multiple CPUs as shown by cpu_distribution.py example script.

Availability: Linux, FreeBSD, SunOS

New in version 5.1.0.

memory_info()

Return a named tuple with variable fields depending on the platform representing memory information about the process. The "portable" fields available on all plaforms are *rss* and *vms*. All numbers are expressed in bytes.

Linux	OSX	BSD	Solaris	Windows
rss	rss	rss	rss	rss (alias for `wset`)
vms	vms	vms	vms	vms (alias for `pagefile`)
shared	pfaults	text		num_page_faults
text	pageins	data		peak_wset
lib		stack		wset
data				peak_paged_pool
dirty				paged_pool
				peak_nonpaged_pool
				nonpaged_pool
				pagefile
				peak_pagefile
				private

- **rss**: aka "Resident Set Size", this is the non-swapped physical memory a process has used. On UNIX it matches "top"'s RES column (see doc). On Windows this is an alias for *wset* field and it matches "Mem Usage" column of taskmgr.exe.
- **vms**: aka "Virtual Memory Size", this is the total amount of virtual memory used by the process. On UNIX it matches "top"'s VIRT column (see doc). On Windows this is an alias for *pagefile* field and it matches "Mem Usage" "VM Size" column of taskmgr.exe.
- **shared**: *(Linux)* memory that could be potentially shared with other processes. This matches "top"'s SHR column (see doc).

- **text** *(Linux, BSD)*: aka TRS (text resident set) the amount of memory devoted to executable code. This matches "top"'s CODE column (see doc).
- **data** *(Linux, BSD)*: aka DRS (data resident set) the amount of physical memory devoted to other than executable code. It matches "top"'s DATA column (see doc).
- **lib** *(Linux)*: the memory used by shared libraries.
- **dirty** *(Linux)*: the number of dirty pages.
- **pfaults** *(OSX)*: number of page faults.
- **pageins** *(OSX)*: number of actual pageins.

For on explanation of Windows fields rely on PROCESS_MEMORY_COUNTERS_EX structure doc. Example on Linux:

```
>>> import psutil
>>> p = psutil.Process()
>>> p.memory_info()
pmem(rss=15491072, vms=84025344, shared=5206016, text=2555904, lib=0,␣
↪data=9891840, dirty=0)
```

Changed in version 4.0.0: multiple fields are returned, not only *rss* and *vms*.

memory_info_ex()

Same as *memory_info()* (deprecated).

> **Warning:** deprecated in version 4.0.0; use *memory_info()* instead.

memory_full_info()

This method returns the same information as *memory_info()*, plus, on some platform (Linux, OSX, Windows), also provides additional metrics (USS, PSS and swap). The additional metrics provide a better representation of "effective" process memory consumption (in case of USS) as explained in detail in this blog post. It does so by passing through the whole process address. As such it usually requires higher user privileges than *memory_info()* and is considerably slower. On platforms where extra fields are not implemented this simply returns the same metrics as *memory_info()*.

- **uss** *(Linux, OSX, Windows)*: aka "Unique Set Size", this is the memory which is unique to a process and which would be freed if the process was terminated right now.
- **pss** *(Linux)*: aka "Proportional Set Size", is the amount of memory shared with other processes, accounted in a way that the amount is divided evenly between the processes that share it. I.e. if a process has 10 MBs all to itself and 10 MBs shared with another process its PSS will be 15 MBs.
- **swap** *(Linux)*: amount of memory that has been swapped out to disk.

Note: *uss* is probably the most representative metric for determining how much memory is actually being used by a process. It represents the amount of memory that would be freed if the process was terminated right now.

Example on Linux:

```
>>> import psutil
>>> p = psutil.Process()
>>> p.memory_full_info()
```

```
pfullmem(rss=10199040, vms=52133888, shared=3887104, text=2867200, lib=0,␣
↪data=5967872, dirty=0, uss=6545408, pss=6872064, swap=0)
>>>
```

See also procsmem.py for an example application.

New in version 4.0.0.

memory_percent (*memtype="rss"*)

Compare process memory to total physical system memory and calculate process memory utilization as a percentage. *memtype* argument is a string that dictates what type of process memory you want to compare against. You can choose between the named tuple field names returned by *memory_info()* and *memory_full_info()* (defaults to `"rss"`).

Changed in version 4.0.0: added *memtype* parameter.

memory_maps (*grouped=True*)

Return process's mapped memory regions as a list of named tuples whose fields are variable depending on the platform. This method is useful to obtain a detailed representation of process memory usage as explained here (the most important value is "private" memory). If *grouped* is `True` the mapped regions with the same *path* are grouped together and the different memory fields are summed. If *grouped* is `False` each mapped region is shown as a single entity and the named tuple will also include the mapped region's address space (*addr*) and permission set (*perms*). See pmap.py for an example application.

Linux	OSX	Windows	Solaris	FreeBSD
rss	rss	rss	rss	rss
size	private		anonymous	private
pss	swapped		locked	ref_count
shared_clean	dirtied			shadow_count
shared_dirty	ref_count			
private_clean	shadow_depth			
private_dirty				
referenced				
anonymous				
swap				

```
>>> import psutil
>>> p = psutil.Process()
>>> p.memory_maps()
[pmmap_grouped(path='/lib/x8664-linux-gnu/libutil-2.15.so', rss=32768,␣
↪size=2125824, pss=32768, shared_clean=0, shared_dirty=0, private_
↪clean=20480, private_dirty=12288, referenced=32768, anonymous=12288,␣
↪swap=0),
 pmmap_grouped(path='/lib/x8664-linux-gnu/libc-2.15.so', rss=3821568,␣
↪size=3842048, pss=3821568, shared_clean=0, shared_dirty=0, private_clean=0,␣
↪private_dirty=3821568, referenced=3575808, anonymous=3821568, swap=0),
 pmmap_grouped(path='/lib/x8664-linux-gnu/libcrypto.so.0.1', rss=34124,␣
↪rss=32768, size=2134016, pss=15360, shared_clean=24576, shared_dirty=0,␣
↪private_clean=0, private_dirty=8192, referenced=24576, anonymous=8192,␣
↪swap=0),
 pmmap_grouped(path='[heap]',  rss=32768, size=139264, pss=32768, shared_
↪clean=0, shared_dirty=0, private_clean=0, private_dirty=32768,␣
↪referenced=32768, anonymous=32768, swap=0),
 pmmap_grouped(path='[stack]', rss=2465792, size=2494464, pss=2465792, shared_
↪clean=0, shared_dirty=0, private_clean=0, private_dirty=2465792,␣
↪referenced=2277376, anonymous=2465792, swap=0),
 ...]
>>> p.memory_maps(grouped=False)
```

```
[pmmap_ext(addr='00400000-006ea000', perms='r-xp', path='/usr/bin/python2.7',␣
↪rss=2293760, size=3055616, pss=1157120, shared_clean=2273280, shared_
↪dirty=0, private_clean=20480, private_dirty=0, referenced=2293760,␣
↪anonymous=0, swap=0),
 pmmap_ext(addr='008e9000-008eb000', perms='r--p', path='/usr/bin/python2.7',␣
↪rss=8192, size=8192, pss=6144, shared_clean=4096, shared_dirty=0, private_
↪clean=0, private_dirty=4096, referenced=8192, anonymous=4096, swap=0),
 pmmap_ext(addr='008eb000-00962000', perms='rw-p', path='/usr/bin/python2.7',␣
↪rss=417792, size=487424, pss=317440, shared_clean=200704, shared_dirty=0,␣
↪private_clean=16384, private_dirty=200704, referenced=417792,␣
↪anonymous=200704, swap=0),
 pmmap_ext(addr='00962000-00985000', perms='rw-p', path='[anon]', rss=139264,␣
↪size=143360, pss=139264, shared_clean=0, shared_dirty=0, private_clean=0,␣
↪private_dirty=139264, referenced=139264, anonymous=139264, swap=0),
 pmmap_ext(addr='02829000-02ccf000', perms='rw-p', path='[heap]', rss=4743168,
↪ size=4874240, pss=4743168, shared_clean=0, shared_dirty=0, private_clean=0,
↪ private_dirty=4743168, referenced=4718592, anonymous=4743168, swap=0),
 ...]
```

Availability: All platforms except OpenBSD and NetBSD.

children(*recursive=False*)

Return the children of this process as a list of *Process* objects, preemptively checking whether PID has been reused. If recursive is *True* return all the parent descendants. Pseudo code example assuming *A == this process*:

```
A -
   |
   - B (child) -
   |             - X (grandchild) -
   |                                - Y (great grandchild)
   - C (child)
   - D (child)

>>> p.children()
B, C, D
>>> p.children(recursive=True)
B, X, Y, C, D
```

Note that in the example above if process X disappears process Y won't be returned either as the reference to process A is lost. This concept is well summaried by this unit test. See also how to *kill a process tree* and *terminate my children*.

open_files()

Return regular files opened by process as a list of named tuples including the following fields:

- **path**: the absolute file name.
- **fd**: the file descriptor number; on Windows this is always `-1`.

Linux only:

- **position** (*Linux*): the file (offset) position.
- **mode** (*Linux*): a string indicating how the file was opened, similarly open's `mode` argument. Possible values are `'r'`, `'w'`, `'a'`, `'r+'` and `'a+'`. There's no distinction between files opened in bynary or text mode (`"b"` or `"t"`).
- **flags** (*Linux*): the flags which were passed to the underlying os.open C call when the file was opened (e.g. os.O_RDONLY, os.O_TRUNC, etc).

```
>>> import psutil
>>> f = open('file.ext', 'w')
>>> p = psutil.Process()
>>> p.open_files()
[popenfile(path='/home/giampaolo/svn/psutil/file.ext', fd=3, position=0, mode=
↪'w', flags=32769)]
```

Warning: on Windows this method is not reliable due to some limitations of the underlying Windows API which may hang when retrieving certain file handles. In order to work around that psutil spawns a thread for each handle and kills it if it's not responding after 100ms. That implies that this method on Windows is not guaranteed to enumerate all regular file handles (see issue 597). Also, it will only list files living in the C:\ drive (see issue 1020).

Warning: on BSD this method can return files with a null path ("") due to a kernel bug, hence it's not reliable (see issue 595).

Changed in version 3.1.0: no longer hangs on Windows.

Changed in version 4.1.0: new *position*, *mode* and *flags* fields on Linux.

connections(*kind="inet"*)

Return socket connections opened by process as a list of named tuples. To get system-wide connections use `psutil.net_connections()`. Every named tuple provides 6 attributes:

- **fd**: the socket file descriptor. This can be passed to socket.fromfd() to obtain a usable socket object. On Windows, FreeBSD and SunOS this is always set to `-1`.
- **family**: the address family, either AF_INET, AF_INET6 or AF_UNIX.
- **type**: the address type, either SOCK_STREAM or SOCK_DGRAM.
- **laddr**: the local address as a `(ip, port)` named tuple or a `path` in case of AF_UNIX sockets. For UNIX sockets see notes below.
- **raddr**: the remote address as a `(ip, port)` named tuple or an absolute `path` in case of UNIX sockets. When the remote endpoint is not connected you'll get an empty tuple (AF_INET*) or `""` (AF_UNIX). For UNIX sockets see notes below.
- **status**: represents the status of a TCP connection. The return value is one of the `psutil.CONN_*` constants. For UDP and UNIX sockets this is always going to be `psutil.CONN_NONE`.

The *kind* parameter is a string which filters for connections that fit the following criteria:

Kind value	Connections using
`"inet"`	IPv4 and IPv6
`"inet4"`	IPv4
`"inet6"`	IPv6
`"tcp"`	TCP
`"tcp4"`	TCP over IPv4
`"tcp6"`	TCP over IPv6
`"udp"`	UDP
`"udp4"`	UDP over IPv4
`"udp6"`	UDP over IPv6
`"unix"`	UNIX socket (both UDP and TCP protocols)
`"all"`	the sum of all the possible families and protocols

Example:

```
>>> import psutil
>>> p = psutil.Process(1694)
>>> p.name()
'firefox'
>>> p.connections()
[pconn(fd=115, family=<AddressFamily.AF_INET: 2>, type=<SocketType.SOCK_
↪STREAM: 1>, laddr=addr(ip='10.0.0.1', port=48776), raddr=addr(ip='93.186.
↪135.91', port=80), status='ESTABLISHED'),
 pconn(fd=117, family=<AddressFamily.AF_INET: 2>, type=<SocketType.SOCK_
↪STREAM: 1>, laddr=addr(ip='10.0.0.1', port=43761), raddr=addr(ip='72.14.234.
↪100', port=80), status='CLOSING'),
 pconn(fd=119, family=<AddressFamily.AF_INET: 2>, type=<SocketType.SOCK_
↪STREAM: 1>, laddr=addr(ip='10.0.0.1', port=60759), raddr=addr(ip='72.14.234.
↪104', port=80), status='ESTABLISHED'),
 pconn(fd=123, family=<AddressFamily.AF_INET: 2>, type=<SocketType.SOCK_
↪STREAM: 1>, laddr=addr(ip='10.0.0.1', port=51314), raddr=addr(ip='72.14.234.
↪83', port=443), status='SYN_SENT')]
```

Note: (Solaris) UNIX sockets are not supported.

Note: (Linux, FreeBSD) "raddr" field for UNIX sockets is always set to "". This is a limitation of the OS.

Note: (OpenBSD) "laddr" and "raddr" fields for UNIX sockets are always set to "". This is a limitation of the OS.

Changed in version 5.3.0: : "laddr" and "raddr" are named tuples.

is_running()
Return whether the current process is running in the current process list. This is reliable also in case the process is gone and its PID reused by another process, therefore it must be preferred over doing `psutil.pid_exists(p.pid)`.

Note: this will return `True` also if the process is a zombie (`p.status() == psutil.STATUS_ZOMBIE`).

send_signal(*signal*)
Send a signal to process (see signal module constants) preemptively checking whether PID has been reused. On UNIX this is the same as `os.kill(pid, sig)`. On Windows only *SIGTERM*, *CTRL_C_EVENT* and *CTRL_BREAK_EVENT* signals are supported and *SIGTERM* is treated as an alias for *kill()*. See also how to *kill a process tree* and *terminate my children*.

Changed in version 3.2.0: support for CTRL_C_EVENT and CTRL_BREAK_EVENT signals on Windows was added.

suspend()
Suspend process execution with *SIGSTOP* signal preemptively checking whether PID has been reused. On UNIX this is the same as `os.kill(pid, signal.SIGSTOP)`. On Windows this is done by suspending all process threads execution.

resume()
Resume process execution with *SIGCONT* signal preemptively checking whether PID has been reused. On UNIX this is the same as `os.kill(pid, signal.SIGCONT)`. On Windows this is done by resuming all process threads execution.

terminate()
Terminate the process with *SIGTERM* signal preemptively checking whether PID has been reused. On UNIX this is the same as `os.kill(pid, signal.SIGTERM)`. On Windows this is an alias for *kill()*. See also how to *kill a process tree* and *terminate my children*.

kill()
Kill the current process by using *SIGKILL* signal preemptively checking whether PID has been reused. On UNIX this is the same as `os.kill(pid, signal.SIGKILL)`. On Windows this is done by using TerminateProcess. See also how to *kill a process tree* and *terminate my children*.

wait(*timeout=None*)
Wait for process termination and if the process is a children of the current one also return the exit code, else `None`. On Windows there's no such limitation (exit code is always returned). If the process is already terminated immediately return `None` instead of raising *NoSuchProcess*. If *timeout* is specified and process is still alive raise *TimeoutExpired* exception. It can also be used in a non-blocking fashion by specifying `timeout=0` in which case it will either return immediately or raise *TimeoutExpired*. To wait for multiple processes use *psutil.wait_procs()*.

```
>>> import psutil
>>> p = psutil.Process(9891)
>>> p.terminate()
>>> p.wait()
```

Popen class

class `psutil.`**Popen**(**args*, ***kwargs*)
A more convenient interface to stdlib subprocess.Popen. It starts a sub process and you deal with it exactly as when using subprocess.Popen but in addition it also provides all the methods of *psutil.Process* class. For method names common to both classes such as *send_signal()*, *terminate()* and *kill()* *psutil.Process* implementation takes precedence. For a complete documentation refer to subprocess module documentation.

Note: Unlike subprocess.Popen this class preemptively checks whether PID has been reused on *send_signal()*, *terminate()* and *kill()* so that you can't accidentally terminate another process, fixing http://bugs.python.org/issue6973.

```
>>> import psutil
>>> from subprocess import PIPE
>>>
>>> p = psutil.Popen(["/usr/bin/python", "-c", "print('hello')"], stdout=PIPE)
>>> p.name()
'python'
>>> p.username()
'giampaolo'
>>> p.communicate()
('hello\n', None)
>>> p.wait(timeout=2)
```

```
0
>>>
```

psutil.Popen objects are supported as context managers via the with statement: on exit, standard file descriptors are closed, and the process is waited for. This is supported on all Python versions.

```
>>> import psutil, subprocess
>>> with psutil.Popen(["ifconfig"], stdout=subprocess.PIPE) as proc:
>>>     log.write(proc.stdout.read())
```

Changed in version 4.4.0: added context manager support

CHAPTER 6

Windows services

psutil.**win_service_iter**()
Return an iterator yielding a *WindowsService* class instance for all Windows services installed.

New in version 4.2.0.

Availability: Windows

psutil.**win_service_get**(*name*)
Get a Windows service by name, returning a *WindowsService* instance. Raise *psutil.NoSuchProcess* if no service with such name exists.

New in version 4.2.0.

Availability: Windows

class psutil.**WindowsService**
Represents a Windows service with the given *name*. This class is returned by *win_service_iter()* and *win_service_get()* functions and it is not supposed to be instantiated directly.

name()
The service name. This string is how a service is referenced and can be passed to *win_service_get()* to get a new *WindowsService* instance.

display_name()
The service display name. The value is cached when this class is instantiated.

binpath()
The fully qualified path to the service binary/exe file as a string, including command line arguments.

username()
The name of the user that owns this service.

start_type()
A string which can either be *"automatic"*, *"manual"* or *"disabled"*.

pid()
The process PID, if any, else *None*. This can be passed to *Process* class to control the service's process.

status()
Service status as a string, which may be either *"running"*, *"paused"*, *"start_pending"*, *"pause_pending"*, *"continue_pending"*, *"stop_pending"* or *"stopped"*.

description()
Service long description.

as_dict()
Utility method retrieving all the information above as a dictionary.

New in version 4.2.0.

Availability: Windows

Example code:

```
>>> import psutil
>>> list(psutil.win_service_iter())
[<WindowsService(name='AeLookupSvc', display_name='Application Experience') at␣
↪38850096>,
 <WindowsService(name='ALG', display_name='Application Layer Gateway Service') at␣
↪38850128>,
 <WindowsService(name='APNMCP', display_name='Ask Update Service') at 38850160>,
 <WindowsService(name='AppIDSvc', display_name='Application Identity') at 38850192>,
 ...]
>>> s = psutil.win_service_get('alg')
>>> s.as_dict()
{'binpath': 'C:\\Windows\\System32\\alg.exe',
 'description': 'Provides support for 3rd party protocol plug-ins for Internet␣
↪Connection Sharing',
 'display_name': 'Application Layer Gateway Service',
 'name': 'alg',
 'pid': None,
 'start_type': 'manual',
 'status': 'stopped',
 'username': 'NT AUTHORITY\\LocalService'}
```

CHAPTER 7

Constants

psutil.**POSIX**

psutil.**WINDOWS**

psutil.**LINUX**

psutil.**OSX**

psutil.**FREEBSD**

psutil.**NETBSD**

psutil.**OPENBSD**

psutil.**BSD**

psutil.**SUNOS**
`bool` constants which define what platform you're on. E.g. if on Windows, *`WINDOWS`* constant will be `True`, all others will be `False`.

New in version 4.0.0.

psutil.**PROCFS_PATH**
The path of the /proc filesystem on Linux and Solaris (defaults to `"/proc"`). You may want to re-set this constant right after importing psutil in case your /proc filesystem is mounted elsewhere or if you want to retrieve information about Linux containers such as Docker, Heroku or LXC (see here for more info). It must be noted that this trick works only for APIs which rely on /proc filesystem (e.g. *memory* APIs and most *`Process`* class methods).

Availability: Linux, Solaris

New in version 3.2.3.

Changed in version 3.4.2: also available on Solaris.

psutil.**STATUS_RUNNING**

psutil.**STATUS_SLEEPING**

psutil.**STATUS_DISK_SLEEP**

psutil.**STATUS_STOPPED**

psutil.**STATUS_TRACING_STOP**

psutil.**STATUS_ZOMBIE**

psutil.**STATUS_DEAD**

psutil.**STATUS_WAKE_KILL**

psutil.**STATUS_WAKING**

psutil.**STATUS_IDLE** (*OSX*, *FreeBSD*)

psutil.**STATUS_LOCKED** (*FreeBSD*)

psutil.**STATUS_WAITING** (*FreeBSD*)

psutil.**STATUS_SUSPENDED** (*NetBSD*)

A set of strings representing the status of a process. Returned by *psutil.Process.status()*.

New in version 3.4.1: STATUS_SUSPENDED (NetBSD)

psutil.**CONN_ESTABLISHED**

psutil.**CONN_SYN_SENT**

psutil.**CONN_SYN_RECV**

psutil.**CONN_FIN_WAIT1**

psutil.**CONN_FIN_WAIT2**

psutil.**CONN_TIME_WAIT**

psutil.**CONN_CLOSE**

psutil.**CONN_CLOSE_WAIT**

psutil.**CONN_LAST_ACK**

psutil.**CONN_LISTEN**

psutil.**CONN_CLOSING**

psutil.**CONN_NONE**

psutil.**CONN_DELETE_TCB** (*Windows*)

psutil.**CONN_IDLE** (*Solaris*)

psutil.**CONN_BOUND** (*Solaris*)

A set of strings representing the status of a TCP connection. Returned by *psutil.Process.connections()* (*status* field).

psutil.**ABOVE_NORMAL_PRIORITY_CLASS**

psutil.**BELOW_NORMAL_PRIORITY_CLASS**

psutil.**HIGH_PRIORITY_CLASS**

psutil.**IDLE_PRIORITY_CLASS**

psutil.**NORMAL_PRIORITY_CLASS**

psutil.**REALTIME_PRIORITY_CLASS**

A set of integers representing the priority of a process on Windows (see MSDN documentation). They can be used in conjunction with *psutil.Process.nice()* to get or set process priority.

Availability: Windows

Changed in version 3.0.0: on Python >= 3.4 these constants are enums instead of a plain integer.

psutil.**IOPRIO_CLASS_NONE**

psutil.**IOPRIO_CLASS_RT**

psutil.**IOPRIO_CLASS_BE**

psutil.**IOPRIO_CLASS_IDLE**

A set of integers representing the I/O priority of a process on Linux. They can be used in conjunction with *psutil.Process.ionice()* to get or set process I/O priority. *IOPRIO_CLASS_NONE* and *IOPRIO_CLASS_BE* (best effort) is the default for any process that hasn't set a specific I/O priority. *IOPRIO_CLASS_RT* (real time) means the process is given first access to the disk, regardless of what else is going on in the system. *IOPRIO_CLASS_IDLE* means the process will get I/O time when no-one else needs the disk. For further information refer to manuals of ionice command line utility or ioprio_get system call.

Availability: Linux

Changed in version 3.0.0: on Python >= 3.4 these constants are enums instead of a plain integer.

psutil.**RLIM_INFINITY**

psutil.**RLIMIT_AS**

psutil.**RLIMIT_CORE**

psutil.**RLIMIT_CPU**

psutil.**RLIMIT_DATA**

psutil.**RLIMIT_FSIZE**

psutil.**RLIMIT_LOCKS**

psutil.**RLIMIT_MEMLOCK**

psutil.**RLIMIT_MSGQUEUE**

psutil.**RLIMIT_NICE**

psutil.**RLIMIT_NOFILE**

psutil.**RLIMIT_NPROC**

psutil.**RLIMIT_RSS**

psutil.**RLIMIT_RTPRIO**

psutil.**RLIMIT_RTTIME**

psutil.**RLIMIT_SIGPENDING**

psutil.**RLIMIT_STACK**

Constants used for getting and setting process resource limits to be used in conjunction with *psutil.Process.rlimit()*. See man prlimit for further information.

Availability: Linux

psutil.**AF_LINK**

Constant which identifies a MAC address associated with a network interface. To be used in conjunction with *psutil.net_if_addrs()*.

New in version 3.0.0.

psutil.**NIC_DUPLEX_FULL**

psutil.**NIC_DUPLEX_HALF**

psutil.**NIC_DUPLEX_UNKNOWN**

Constants which identifies whether a NIC (network interface card) has full or half mode speed. NIC_DUPLEX_FULL means the NIC is able to send and receive data (files) simultaneously, NIC_DUPLEX_FULL means the NIC can either send or receive data at a time. To be used in conjunction with *psutil.net_if_stats()*.

New in version 3.0.0.

psutil.**POWER_TIME_UNKNOWN**

psutil.**POWER_TIME_UNLIMITED**

Whether the remaining time of the battery cannot be determined or is unlimited. May be assigned to *psutil.sensors_battery()*'s *secsleft* field.

New in version 5.1.0.

psutil.**version_info**

A tuple to check psutil installed version. Example:

```
>>> import psutil
>>> if psutil.version_info >= (4, 5):
...     pass
```

CHAPTER 8

Unicode

Starting from version 5.3.0 psutil fully supports unicode, see issue #1040. The notes below apply to *any* API returning a string such as *Process.exe()* or *Process.cwd()*, including non-filesystem related methods such as *Process.username()* or *WindowsService.description()*:

- all strings are encoded by using the OS filesystem encoding (`sys.getfilesystemencoding()`) which varies depending on the platform (e.g. "UTF-8" on OSX, "mbcs" on Win)
- no API call is supposed to crash with `UnicodeDecodeError`
- **instead, in case of badly encoded data returned by the OS, the following error handlers are used to replace the corrupted c**
  - Python 3: `sys.getfilesystemencodeerrors()` (PY 3.6+) or `"surrogatescape"` on POSIX and `"replace"` on Windows
  - Python 2: `"replace"`
- on Python 2 all APIs return bytes (`str` type), never `unicode`
- on Python 2, you can go back to `unicode` by doing:

```
>>> unicode(p.exe(), sys.getdefaultencoding(), errors="replace")
```

Example which filters processes with a funky name working with both Python 2 and 3:

```
# -*- coding: utf-8 -*-
import psutil, sys

PY3 = sys.version_info[0] == 2
LOOKFOR = u"főő"
for proc in psutil.process_iter(attrs=['name']):
    name = proc.info['name']
    if not PY3:
        name = unicode(name, sys.getdefaultencoding(), errors="replace")
    if LOOKFOR == name:
         print("process %s found" % p)
```

CHAPTER 9

Recipes

Follows a collection of utilities and examples which are common but not generic enough to be part of the public API.

Find process by name

Check string against *Process.name()*:

```
import psutil

def find_procs_by_name(name):
    "Return a list of processes matching 'name'."
    ls = []
    for p in psutil.process_iter(attrs=['name']):
        if p.info['name'] == name:
            ls.append(p)
    return ls
```

A bit more advanced, check string against *Process.name()*, *Process.exe()* and *Process.cmdline()*:

```
import os
import psutil

def find_procs_by_name(name):
    "Return a list of processes matching 'name'."
    ls = []
    for p in psutil.process_iter(attrs=["name", "exe", "cmdline"]):
        if name == p.info['name'] or \
                p.info['exe'] and os.path.basename(p.info['exe']) == name or \
                p.info['cmdline'] and p.info['cmdline'][0] == name:
            ls.append(p)
    return ls
```

Kill process tree

```
import os
import signal
import psutil

def kill_proc_tree(pid, sig=signal.SIGTERM, include_parent=True,
                   timeout=None, on_terminate=None):
    """Kill a process tree (including grandchildren) with signal
    "sig" and return a (gone, still_alive) tuple.
    "on_terminate", if specified, is a callabck function which is
    called as soon as a child terminates.
    """
    if pid == os.getpid():
        raise RuntimeError("I refuse to kill myself")
    parent = psutil.Process(pid)
    children = parent.children(recursive=True)
    if include_parent:
        children.append(parent)
    for p in children:
        p.send_signal(sig)
    gone, alive = psutil.wait_procs(children, timeout=timeout,
                                    callback=on_terminate)
    return (gone, alive)
```

Terminate my children

This may be useful in unit tests whenever sub-processes are started. This will help ensure that no extra children (zombies) stick around to hog resources.

```
import psutil

def reap_children(timeout=3):
    "Tries hard to terminate and ultimately kill all the children of this process."
    def on_terminate(proc):
        print("process {} terminated with exit code {}".format(proc, proc.returncode))

    procs = psutil.Process().children()
    # send SIGTERM
    for p in procs:
        p.terminate()
    gone, alive = psutil.wait_procs(procs, timeout=timeout, callback=on_terminate)
    if not alive:
        # send SIGKILL
        for p in alive:
            print("process {} survived SIGTERM; trying SIGKILL" % p)
            p.kill()
        gone, alive = psutil.wait_procs(alive, timeout=timeout, callback=on_terminate)
        if not alive:
            # give up
            for p in alive:
                print("process {} survived SIGKILL; giving up" % p)
```

Filtering and sorting processes

This is a collection of one-liners showing how to use *process_iter()* in order to filter for processes and sort them.

Setup:

```
>>> import psutil
>>> from pprint import pprint as pp
```

Processes having "python" in their name:

```
>>> pp([p.info for p in psutil.process_iter(attrs=['pid', 'name']) if 'python' in p.
↪info['name']])
[{'name': 'python3', 'pid': 21947},
 {'name': 'python', 'pid': 23835}]
```

Processes owned by user:

```
>>> import getpass
>>> pp([(p.pid, p.info['name']) for p in psutil.process_iter(attrs=['name', 'username
↪']) if p.info['username'] == getpass.getuser()])
(16832, 'bash'),
(19772, 'ssh'),
(20492, 'python')]
```

Processes actively running:

```
>>> pp([(p.pid, p.info) for p in psutil.process_iter(attrs=['name', 'status']) if p.
↪info['status'] == psutil.STATUS_RUNNING])
[(1150, {'name': 'Xorg', 'status': 'running'}),
 (1776, {'name': 'unity-panel-service', 'status': 'running'}),
 (20492, {'name': 'python', 'status': 'running'})]
```

Processes using log files:

```
>>> import os
>>> import psutil
>>> for p in psutil.process_iter(attrs=['name', 'open_files']):
...     for file in p.info['open_files'] or []:
...         if os.path.splitext(file.path)[1] == '.log':
...             print("%-5s %-10s %s" % (p.pid, p.info['name'][:10], file.path))
...
1510  upstart    /home/giampaolo/.cache/upstart/unity-settings-daemon.log
2174  nautilus   /home/giampaolo/.local/share/gvfs-metadata/home-ce08efac.log
2650  chrome     /home/giampaolo/.config/google-chrome/Default/data_reduction_proxy_
↪leveldb/000003.log
```

Processes consuming more than 500M of memory:

```
>>> pp([(p.pid, p.info['name'], p.info['memory_info'].rss) for p in psutil.process_
↪iter(attrs=['name', 'memory_info']) if p.info['memory_info'].rss > 500 * 1024 *␣
↪1024])
[(2650, 'chrome', 532324352),
 (3038, 'chrome', 1120088064),
 (21915, 'sublime_text', 615407616)]
```

Top 3 most memory consuming processes:

```
>>> pp([(p.pid, p.info) for p in sorted(psutil.process_iter(attrs=['name', 'memory_
↪percent']), key=lambda p: p.info['memory_percent'])][-3:])
[(21915, {'memory_percent': 3.6815453247662737, 'name': 'sublime_text'}),
 (3038, {'memory_percent': 6.732935429979187, 'name': 'chrome'}),
 (3249, {'memory_percent': 8.994554843376399, 'name': 'chrome'})]
```

Top 3 processes which consumed the most CPU time:

```
>>> pp([(p.pid, p.info['name'], sum(p.info['cpu_times'])) for p in sorted(psutil.
↪process_iter(attrs=['name', 'cpu_times']), key=lambda p: sum(p.info['cpu_times
↪'][:2]))][-3:])
[(2721, 'chrome', 10219.73),
 (1150, 'Xorg', 11116.989999999998),
 (2650, 'chrome', 18451.97)]
```

Top 3 processes which caused the most I/O:

```
>>> pp([(p.pid, p.info['name']) for p in sorted(psutil.process_iter(attrs=['name',
↪'io_counters']), key=lambda p: p.info['io_counters'] and p.info['io_counters
↪'][:2])][-3:])
[(21915, 'sublime_text'),
 (1871, 'pulseaudio'),
 (1510, 'upstart')]
```

Top 3 processes opening more file descriptors:

```
 >>> pp([(p.pid, p.info) for p in sorted(psutil.process_iter(attrs=['name', 'num_fds
↪']), key=lambda p: p.info['num_fds'])][-3:])
[(21915, {'name': 'sublime_text', 'num_fds': 105}),
 (2721, {'name': 'chrome', 'num_fds': 185}),
 (2650, {'name': 'chrome', 'num_fds': 354})]
```

CHAPTER 10

Q&A

- Q: What Windows versions are supported?
- A: From Windows **Vista** onwards, both 32 and 64 bit versions. Latest binary (wheel / exe) release which supports Windows **2000**, **XP** and **2003 server** is psutil 3.4.2. On such old systems psutil is no longer tested or maintained, but it can still be compiled from sources (you'll need Visual Studio) and it should "work" (more or less).

- Q: What SunOS versions are supported?
- A: From Solaris 10 onwards.

- Q: Why do I get `AccessDenied` for certain processes?
- A: This may happen when you query processess owned by another user, especially on OSX and Windows. Unfortunately there's not much you can do about this except running the Python process with higher privileges. On Unix you may run the the Python process as root or use the SUID bit (this is the trick used by tools such as `ps` and `netstat`). On Windows you may run the Python process as NT AUTHORITY\SYSTEM or install the Python script as a Windows service (this is the trick used by tools such as ProcessHacker).

- Q: What about load average?
- A: psutil does not expose any load average function as it's already available in python as os.getloadavg

CHAPTER 11

Running tests

There are two ways of running tests. If psutil is already installed use:

```
$ python -m psutil.tests
```

You can use this method as a quick way to make sure psutil fully works on your platform. If you have a copy of the source code you can also use:

```
$ make test
```

CHAPTER 12

Development guide

If you plan on hacking on psutil (e.g. want to add a new feature or fix a bug) take a look at the development guide.

CHAPTER 13

Timeline

- 2017-09-01: 5.3.0 - what's new - diff
- 2017-04-10: 5.2.2 - what's new - diff
- 2017-03-24: 5.2.1 - what's new - diff
- 2017-03-05: 5.2.0 - what's new - diff
- 2017-02-07: 5.1.3 - what's new - diff
- 2017-02-03: 5.1.2 - what's new - diff
- 2017-02-03: 5.1.1 - what's new - diff
- 2017-02-01: 5.1.0 - what's new - diff
- 2016-12-21: 5.0.1 - what's new - diff
- 2016-11-06: 5.0.0 - what's new - diff
- 2016-10-05: 4.4.2 - what's new - diff
- 2016-10-25: 4.4.1 - what's new - diff
- 2016-10-23: 4.4.0 - what's new - diff
- 2016-09-01: 4.3.1 - what's new - diff
- 2016-06-18: 4.3.0 - what's new - diff
- 2016-05-14: 4.2.0 - what's new - diff
- 2016-03-12: 4.1.0 - what's new - diff
- 2016-02-17: 4.0.0 - what's new - diff
- 2016-01-20: 3.4.2 - what's new - diff
- 2016-01-15: 3.4.1 - what's new - diff
- 2015-11-25: 3.3.0 - what's new - diff
- 2015-10-04: 3.2.2 - what's new - diff

- 2015-09-03: 3.2.1 - what's new - diff
- 2015-09-02: 3.2.0 - what's new - diff
- 2015-07-15: 3.1.1 - what's new - diff
- 2015-07-15: 3.1.0 - what's new - diff
- 2015-06-18: 3.0.1 - what's new - diff
- 2015-06-13: 3.0.0 - what's new - diff
- 2015-02-02: 2.2.1 - what's new - diff
- 2015-01-06: 2.2.0 - what's new - diff
- 2014-09-26: 2.1.3 - what's new - diff
- 2014-09-21: 2.1.2 - what's new - diff
- 2014-04-30: 2.1.1 - what's new - diff
- 2014-04-08: 2.1.0 - what's new - diff
- 2014-03-10: 2.0.0 - what's new - diff
- 2013-11-25: 1.2.1 - what's new - diff
- 2013-11-20: 1.2.0 - what's new - diff
- 2013-10-22: 1.1.2 - what's new - diff
- 2013-10-08: 1.1.1 - what's new - diff
- 2013-09-28: 1.1.0 - what's new - diff
- 2013-07-12: 1.0.1 - what's new - diff
- 2013-07-10: 1.0.0 - what's new - diff
- 2013-05-03: 0.7.1 - what's new - diff
- 2013-04-12: 0.7.0 - what's new - diff
- 2012-08-16: 0.6.1 - what's new - diff
- 2012-08-13: 0.6.0 - what's new - diff
- 2012-06-29: 0.5.1 - what's new - diff
- 2012-06-27: 0.5.0 - what's new - diff
- 2011-12-14: 0.4.1 - what's new - diff
- 2011-10-29: 0.4.0 - what's new - diff
- 2011-07-08: 0.3.0 - what's new - diff
- 2011-03-20: 0.2.1 - what's new - diff
- 2010-11-13: 0.2.0 - what's new - diff
- 2010-03-02: 0.1.3 - what's new - diff
- 2009-05-06: 0.1.2 - what's new - diff
- 2009-03-06: 0.1.1 - what's new - diff
- 2009-01-27: 0.1.0 - what's new - diff

Python Module Index

p

Index

U

V

W

Z