301AA - Advanced Programming

Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

***AP-27**: Garbage collection, GIL, scripting*

Garbage collection in Python

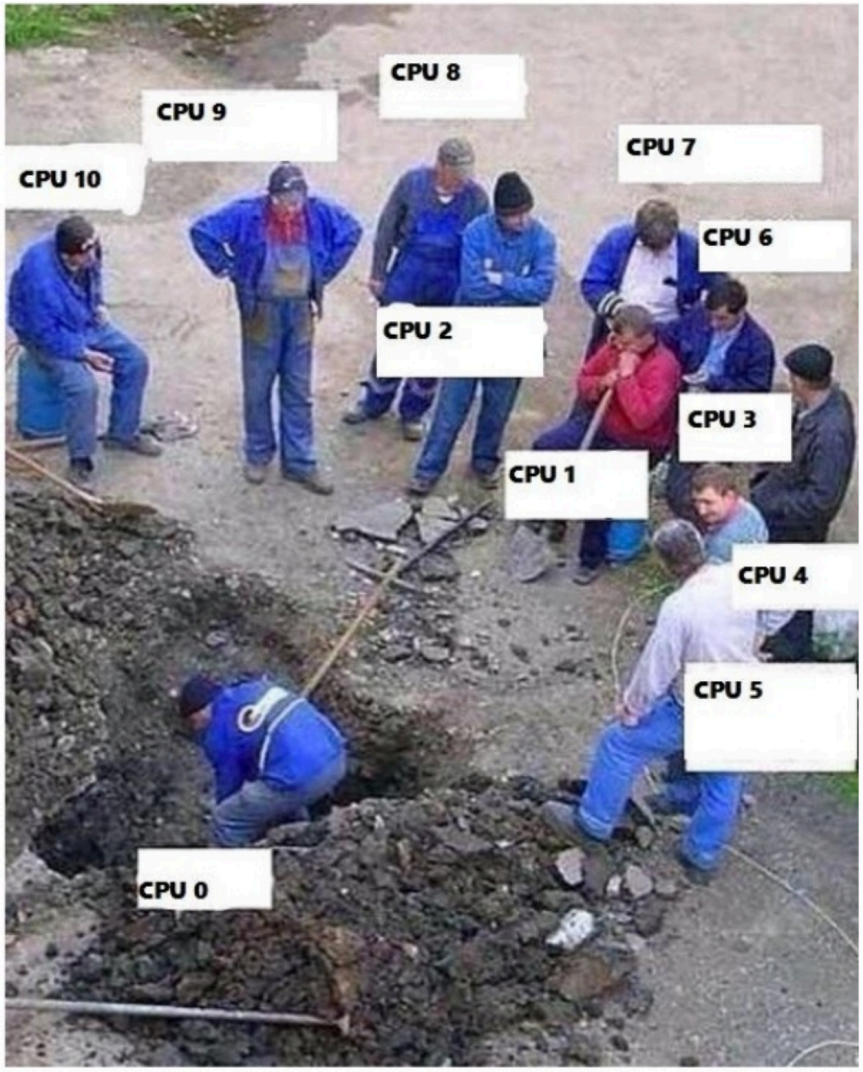
CPython manages memory with a **reference counting** + a **mark&sweep** cycle collector scheme

- Reference counting: each object has a counter storing the number of references to it. When it becomes 0, memory can be reclaimed.
- Pros: simple implementation, memory is reclaimed as soon as possible, no need to freeze execution passing control to a garbage collector
- Cons: additional memory needed for each object; cyclic structures in garbage cannot be identified (thus the need of mark&sweep)

Handling reference counters

- Updating the refcount of an object has to be done atomically
- In case of multi-threading you need to synchronize all the times you modify refcounts, or else you can have wrong values 
- Synchronization primitives are quite expensive on contemporary hardware
- Since almost every operation in CPython can cause a refcount to change somewhere, handling refcounts with some kind of synchronization would cause *spending almost all the time on synchronization* 
- As a consequence…

Concurrency in Python…



The Global Interpreter Lock (GIL)

- The CPython interpreter assures that only one thread executes Python bytecode at a time, thanks to the **Global Interpreter Lock**
- The current thread must hold the **GIL** before it can safely access Python objects
- This simplifies the CPython implementation by making the object model (including critical built-in types such as **dict**) implicitly safe against concurrent access 
- Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, **at the expense of much of the parallelism afforded by multi-processor machines**.

More on the GIL 

- However the GIL can degrade performance even when it is not a bottleneck. The system call overhead is significant, especially on multicore hardware.
- Two threads calling a function may take twice as much time as a single thread calling the function twice.
- The GIL can cause I/O-bound threads to be scheduled ahead of CPU-bound threads. And it prevents signals from being delivered.
- Some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing.
- Also, the GIL is always released when doing I/O.

Alternatives to the GIL?

- Past efforts to create a “free-threaded” interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case.
- It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.
- Guido van Rossum has said he will reject any proposal in this direction that slows down single-threaded programs.
- **Jython** (on JVM, -> 2017, Python 2.7) and **IronPython** (on .NET) have no GIL and can fully exploit multiprocessor systems
- **PyPy** (Python in Python, supporting JIT) currently has a GIL like CPython
- in **Cython** (compiled, for CPython extension modules) the GIL exists, but can be released temporarily using a "with" statement

Criticisms to Python: syntax of tuples

```
>>> type((1,2,3))
<class 'tuple'>
>>> type(())
<class 'tuple'>
>>> type((1))
<class 'int'>
>>> type((1,))
<class 'tuple'>
```

- Tuples are made by the commas, not by ( )
- With the exception of the empty tuple…

Criticisms to Python: indentation

- Lack of brackets makes the syntax "weaker" than in other languages: accidental changes of indentation may change the semantics, leaving the program syntactically correct.

```
def foo(x):
    if x == 0:
        bar()
        baz()
    else:
        qux(x)
        foo(x - 1)
```

```
def foo(x):
    if x == 0:
        bar()
        baz()
    else:
        qux(x)
    foo(x - 1)
```

- Mixed use of tabs and blanks may cause bugs almost impossible to detect

Criticisms to Python: indentation

- Lack of brackets makes it harder to refactor the code or insert new one
- "When I want to refactor a bulk of code in Python, I need to be very careful. Because if lost, I'm not sure what I'm editing belongs to which part of the code. Python depends on indentation, so if I have mistakenly removed some indentation, I totally have no idea whether the correct code should belong to that **if** clause or this **while** clause." 
- Will Python change in the future?

```
>>> from __future__ import braces
  File "<stdin>", line 1
SyntaxError: not a chance
>>>
```

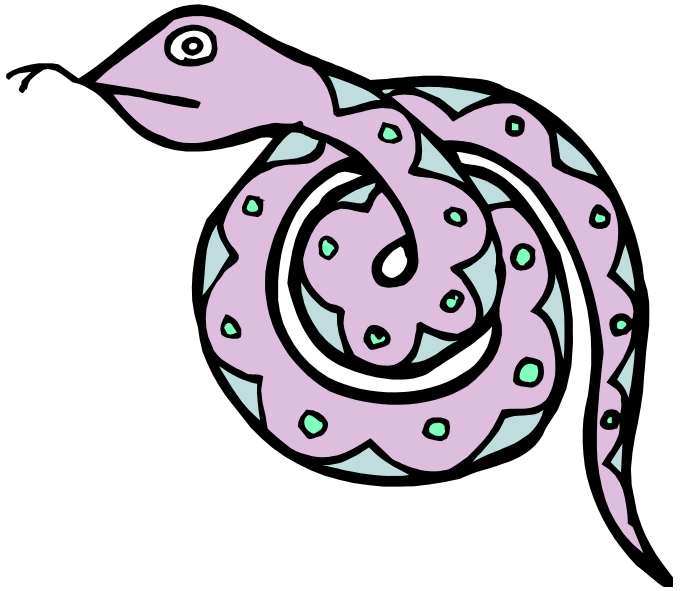
Builtins & Libraries

- The Python ecosystem is extremely rich and in fast evolution
- For available functions, classes and modules browse:
  - **Builtin Functions**
    - https://docs.python.org/3.8/library/functions.html
  - **Standard library**
    - https://docs.python.org/3.8/tutorial/stdlib.html
- There are dozens of other libraries, mainly for scientific computing, machine learning, computational biology, data manipulation and analysis, natural language processing, statistics, symbolic computation, etc.

Python Scripts for System Administrators

https://www.ibm.com/developerworks/aix/library/au-python/index.html

The next slides present tested Python 3 code essentially equivalent to the examples in the URL above



Example 1: Search for files and show permissions in a friendly format

```
import stat, sys, os, subprocess            # Python 3

# Getting search pattern from user and assigning it to a list

try:
    # run a 'find' command and assign results to a variable
    pattern = input("Enter the file pattern to search for:\n")
    commandString = "find " + pattern
    commandOutput = subprocess.getoutput(commandString)
    findResults = commandOutput.split("\n")

    # output find results, along with permissions
    print ("Files:")
    print (commandOutput)
    print ("================================")
    for file in findResults:
        mode = stat.S_IMODE(os.lstat(file)[stat.ST_MODE])
        print ("\nPermissions for file ", file, ":")
        for level in "USR", "GRP", "OTH":
            for perm in "R", "W", "X":
                if mode & getattr(stat,"S_I"+perm+level): # bitwise and
                    print (level, " has ", perm, " permission")
                else:
                    print (level, " does NOT have ", perm, " permission")
except:
    print ("There was a problem - check the message above")
```

Example 2: Perform operations on a tar archive that is based on menu selection

```
import tarfile, sys                 # Python 3

try:
    #open tarfile
    tar = tarfile.open(sys.argv[1], "r:tar")    # <class 'tarfile.TarFile'>

    #present menu and get selection
    selection = input("Enter\n\
    1 to extract a file\n\
    2 to display information on a file in the archive\n\
    3 to list all the files in the archive\n\n")

    #perform actions based on selection above
    if selection == "1":
        filename = input("enter the filename to extract:  ")
        tar.extract(filename)
    elif selection == "2":
        filename = input("enter the filename to inspect:  ")
        for tarinfo in tar:                  # <class 'tarfile.TarInfo'>
            if tarinfo.name == filename:
                print( "\n\
                Filename:\t\t", tarinfo.name, "\n\
                Size:\t\t", tarinfo.size, "bytes\n")
    elif selection == "3":
        print (tar.list(verbose=True))
except:
    print ("There was a problem running the program")
```

Example 3: Check for a running process and show information in a friendly format

```
import subprocess, os                    # Python 3

program = input("Enter the name of the program to check: ")

try:
    #perform a ps command and assign results to a list
    output = subprocess.getoutput("ps -f|grep " + program)
    processes = output.split("\n")
    for process in processes:
        proginfo = process.split()
        #display results
        print ("\n\                                        # Note: correctness depends
        Full path:\t\t", proginfo[7], "\n\                 # on the structure of the
        Owner:\t\t\t", proginfo[0], "\n\                   # output of ps
        Process ID:\t\t", proginfo[1], "\n\
        Parent process ID:\t", proginfo[2], "\n\
        Time started:\t\t", proginfo[4], "\n\
        **************************")
except:
    print ("There was a problem with the program.")
```

Example 4: Check userids and passwords for policy compliance

```
import pwd
#initialize lists
erroruser = []
errorpass = []
#get password database
passwd_db = pwd.getpwall()          # a list of <class 'pwd.struct_passwd'>
try:
    #check each user and password for validity
    for entry in passwd_db:         # <class 'pwd.struct_passwd'>
        username = entry[0]         # also  entry.pw_name
        password = entry [1]        # also  entry.pw_passwd
        if len(username) < 6:
            erroruser.append(username)
        if len(password) < 8:
            errorpass.append(username)
    #print results to screen
    print ("The following users have an invalid userid (< six characters):")
    for item in erroruser:
        print (item)
    print ("\nThe following users have invalid password(< eight characters):")
    for item in errorpass:
        print (item)
except:
    print ("There was a problem running the script.")
```

Example 5: Force "quit" of selected processes (from Ch 13 of Scott: Programming Language Pragmatics)

```
import sys, os, re, time
if len(sys.argv) != 2:
    sys.stderr.write('usage: ' + sys.argv[0] + ' pattern\n')
    sys.exit(1)
PS = os.popen("/bin/ps -w -w -x -o'pid,command'") # opens a pipe!
line = PS.readline()                  # discard header line
line = PS.readline().rstrip()         # prime pump
while line != "":
    proc = int(re.search('\S+', line).group()) # first occurrence of non-blanks
    if re.search(sys.argv[1], line) and proc != os.getpid():
        print (line + '? ',end='', flush=True)
        answer = sys.stdin.readline()
        while not re.search('^[yn]', answer, re.I):
            print ('? ',end='', flush=True)
            answer = sys.stdin.readline()
        if re.search('^y', answer, re.I):
            os.kill(proc, 9)
            time.sleep(1)
            try:                      # expect exception if process
                os.kill(proc, 0)      # no longer exists
                sys.stderr.write("unsuccessful; sorry\n"); sys.exit(1)
            except: pass            # do nothing
        sys.stdout.write('')      # inhibit prepended blank on next print
    line = PS.readline().rstrip()
```

Example: Create a PDF file containing the source code of a list of Java, Haskell and Python files (uses 'a2ps' and 'ps2pdf')

➔ **compare with the next script**

```
import sys, subprocess
# checks the number of arguments
if len(sys.argv) != 3:
    sys.stderr.write('usage: ' + sys.argv[0] + ' paths_file ps_file_name\n')
    sys.exit(1)
# reads the file, assuming that it contains relative paths of source files
paths_file = open(sys.argv[1])
#strips "\n" from each file name
stripped = (f.strip("\n") for f in  paths_file)
# filters out files of wrong type (with suffix not in {.java,.py,.hs})
checked = (file for file in stripped if file.endswith((".java",".hs",".py")))
# concatenates the file names
a2ps_files = " ".join(checked)
# using the 'a2ps' utility, generates a single PostScript file containing
# a pretty printed version of all the files passed as first argument.
command = "a2ps -A fill -o "+ sys.argv[2] + ".ps " +  a2ps_files
result = subprocess.getstatusoutput(command)
if result[0]!= 0 :
    print("There was an error... Result of a2ps: " + result[1])
else :
    print("Result of a2ps: " + result[1])
# converts a PostScript file to PDF using 'ps2pdf'
command = "ps2pdf "+ sys.argv[2] + ".ps "
result = subprocess.getstatusoutput(command)
print("Postscript file " + (sys.argv[2] + ".ps") + " converted to PDF.")
```

Example: The previous script, structured as a set of functions corresponding to basic operations. It can be invoked in the interpreter or as command line arg to **python**

```
import sys, subprocess

def check_args() -> None :
    # checks the number of arguments
    if len(sys.argv) != 3:
       sys.stderr.write('usage: ' + sys.argv[0] + ' paths_file ps_file_name\n')
       sys.exit(1)
def prepare_a2ps_args(paths_file:str) -> str :
    # builds a string listing the files to be printed
    # reads the file, assuming that it contains relative paths of source files
    pathsFile = open(paths_file)
    stripped = (f.strip("\n") for f in paths_file) # strips "\n" from file name
    # filters out files of wrong type (with suffix not in {.java,.py,.hs})
    checked = (fi for fi in stripped if fi.endswith((".java",".hs",".py")))
    # concatenates the file names
    return " ".join(checked)

def generate_postscript(file_names:[str], ps_file_name: str) -> str :
    # using the 'a2ps' utility, generates a single PostScript file containing
    # a pretty printed version of all the files passed as first argument.
    command = "a2ps -A fill -o "+ ps_file_name + ".ps " +  file_names
    result = subprocess.getstatusoutput(command)
    if result[0]!= 0 :
        return ("c'e' stato un errore... Result of a2ps: " + result[2])
    else :
        return ("Result of a2ps: " + result[1])
```

Example: The previous script, structured as a set of functions corresponding to basic operations. (cont.)

```
# (continue)

def generate_PDF(ps_file_name: str) -> str :
    # converts a PostScript file to PDF using 'ps2pdf'
    command = "ps2pdf "+ ps_file_name + ".ps "
    result = subprocess.getstatusoutput(command)
    print("Postscript file " + (sys.argv[2] + ".ps") + " converted to PDF.")

def main():
    # to be run in the interpreter: arguments are asked interactively
    file_list = input("Name of file with list of files to print? ")
    ps_name = input("Name of PS/PDF file? ")
    a2ps_args = prepare_A2psArgs(file_list)
    print(generate_postscript(a2ps_args, ps_name))
    print(generate_PDF(psName))

if __name__ == "__main__" :
    # executed when passed as argument to 'python3'
    check_args()
    a2psArgs = prepare_a2ps_args(sys.argv[1])
    print(generate_postscript(a2ps_args, sys.argv[2]))
    print(generate_PDF(sys.argv[2]))
```

Concluding remarks

When writing scripts, try hard to meet the following goals:

- **Portability**: make as few assumptions as possible on the underlying operating system, possibly none 
- **Readability:** comment the code, and annotate function arguments and the function result with the expected types
- **Reusability**: the script should be made of a set of functions, each implementing a small and well identified task
- **Executability**: make the script executable in a variety of modes: as stand-alone executable, as command-line argument to **python**, interactively in the Python interpreter