Architektura systemów Bioinformatycznych

Narzędzia do debugowania, Śledzenia i profilowania programów, i logowania błędów

Bartek Wilczyński

18.III 2020

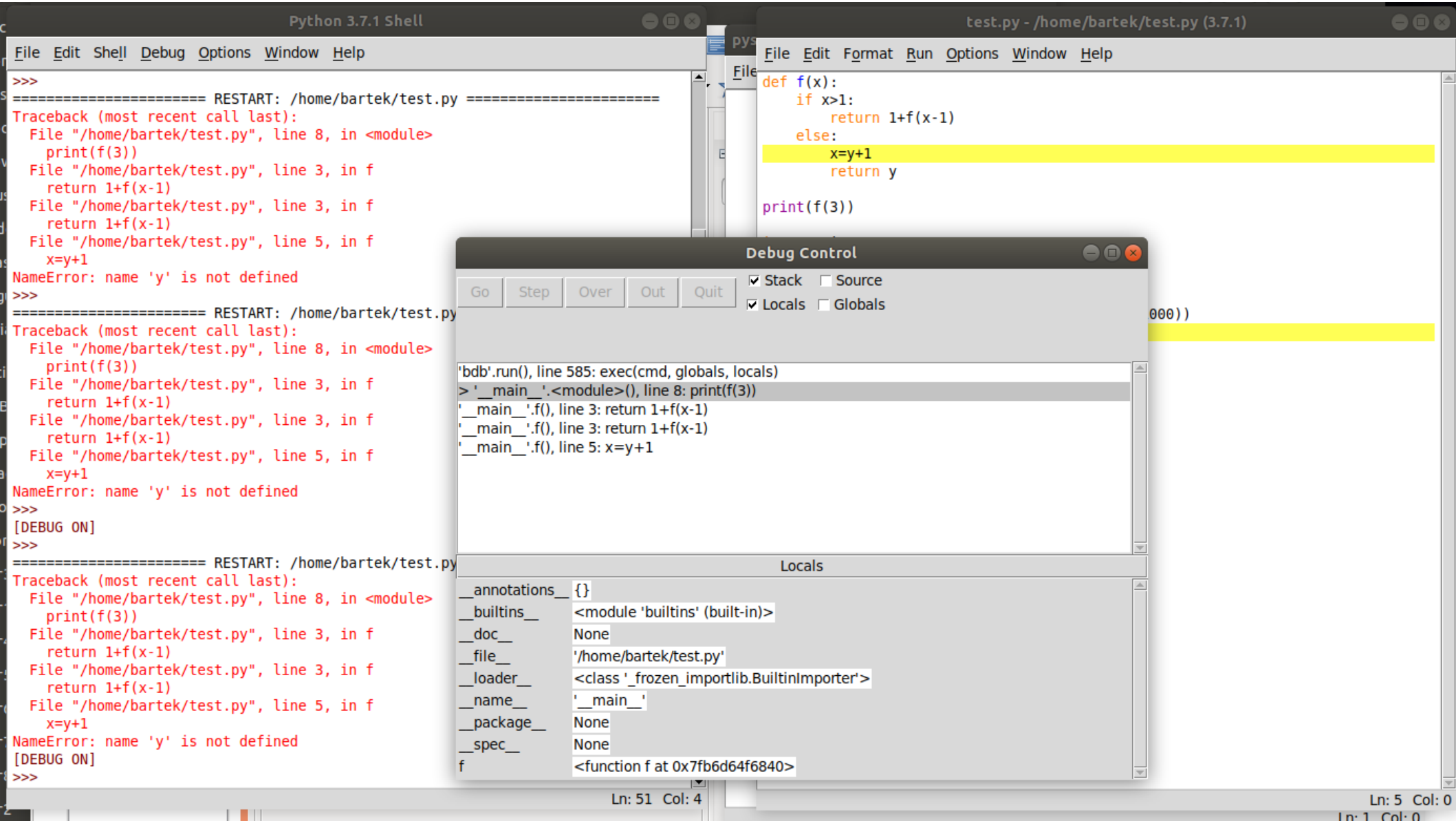
Plan na dziś

- Debugery: pdb, idle, inne graficzne interfejsy
- Profilery: profile,cProfile, kernprofiler, runsnakerun
- Benchmarking: timeit, pystone, etc.
- Moduł inspect – co jest pod maską autouzupełniania
- Logowanie/raportowania błędów: Moduły logging, warnings, loguru

Debugowanie kodu

- Potrzebujemy czasem wykonać część naszego kodu, żeby zobaczyć co się dzieje nie tak podczas wykonania
- Najprostszy jest oczywiście tzw.
  „print debugger”
- Python ma wbudowany moduł pdb – python debugger
- Obsługa pdb jest wbudowana w środowisko IDLE
- Wiele innych środowisk (np. popularny pyCharm) też obsługuje pdb

Graficzny debugger



Tekstowy debugger

- Możemy uruchomić nasz program poprzez `python -m pdb program.py`
- Jednoliterowe komendy typu (s)tep, (c)ontinue, (u)p, etc.
- Możemy też wpisać do kodu komendy typu

```
import pdb

pdb.set_trace()
```

Python Debugger Cheatsheet

Getting started

import pdb;pdb.set_trace()	start pdb from within a script
python -m pdb <file.py>	start pdb from the commandline

Basics

h(elp)	print available commands
h(elp) ***command***	print help about *command*
q(uit)	quit debugger

Examine

p(rint) ***expr***	print the value of *expr*
pp ***expr***	pretty-print the value of *expr*
w(here)	print current position (including stack trace)
l(ist)	list 11 lines of code around the current line
l(ist) ***first, last***	list from *first* to *last* line number
a(rgs)	print the args of the current function

Miscellaneous

!*stmt*	treat *stmt* as a Python statement instead of a pdb command
alias ***map stmt***	map Python statement as a map command
alias ***map <arg1 ...> stmt***	pass arguments to Python statement. *stmt* includes %1, %2, ... literals.

Save pdb commands to local <./.pdbrc> file for repetitive access.

Movement

<ENTER>	repeat the last command
n(ext)	execute the current statement (step over)
s(tep)	execute and step into function
r(eturn)	continue execution until the current function returns
c(ontinue)	continue execution until a breakpoint is encountered
u(p)	move one level up in the stack trace
d(own)	move one level down in the stack trace
until	continue execution until the end of a loop or until the set line
j(ump)	set the next line that will be executed (local frame only)

Breakpoints

b(reak)	show all breakpoints with its *number*
b(reak) ***lineno***	set a breakpoint at *lineno*
b(reak) ***lineno, cond***	stop at breakpoint *lineno* if Python condition *cond* holds, e.g. i==42
b(reak) ***file:lineno***	set a breakpoint in *file* at *lineno*
b(reak) ***func***	set a breakpoint at the first line of a *func*
tbreak ***lineno***	set a temporary breakpoint at *lineno*, i.e. is removed when first hit
disable ***number***	disable breakpoint *number*
enable ***number***	enable breakpoint *number*
clear ***number***	delete breakpoint *number*

Author: Florian Preinstorfer (nblock@archlinux.us) — version 1.2 —
See `https://github.com/nblock/pdb-cheatsheet` for more information.

Co jest pod maską?

- Moduł FaultHandler – pythonowy moduł zapewniający co się stanie, gdy program napotka błąd (przerwanie, nieobsłużony wyjątek, etc.)
- Możemy to kontrolować, np. żeby przejąć ctrl-c
- Jest też moduł inspect – pozwalający na przeglądanie własności obiektów w interpreterze pythona, który pozwala np. na autouzupełnianie

Dwa cytaty nt. optymalizacji kodu

- First make it work,
  Then make it right,
  Then make it fast
  (Kent Beck)

- Premature optimization is
  the root of all evil in programming.
  (Donald Knuth, attributing to Hoare)

Profilowanie vs. Pomiar czasu wykonania

- Możemy zastanawiać się jak zmienić kod, żeby działał szybciej
- Jeżeli chcemy wiedzieć jak szybko działają różne wersje naszego kodu, albo na różnych komputerach – mówimy o pomiarze czasu wykonania (benchmarking)
- Jeśli chcemy zobaczyć, które fragmenty kodu zajmują nam najwięcej czasu, możemy użyć narzędzi do profilowania (profiling)

Mouł timeit

- Istnieje funkcja timeit() w module timeit, która pozwala na prosty benchmarking wyrażeń w pythonie

```
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

```
def test():
    """Stupid test function"""
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

Moduł time do benchmarkingu

- Możemy używać modułu time

```
import time

start_time = int(round(time.time() * 1000))
test_function()
time_diff = time.time() - start_time
print({'total time': time_diff})
```

- Albo zrobić dekorator

```
import time

def time_function(f):
    def timed(*args,**kwds):
        start_time = int(round(time.time() * 1000))
        result = f(*args,**kwds)
        time_diff = time.time() - start_time
        print({'total time': time_diff})
        return result
    return timed

@time_function
def test_function():
    pass
```

Benchmarki sprzętu

- Warto wiedzieć jak szybki jest komputer, może nam pomóc moduł test.pystone

```
bartek@centromere:~$ python -m test.pystone
Pystone(1.1) time for 50000 passes = 0.761473
This machine benchmarks at 65662.2 pystones/second


bartek@nucleus:~$ python -m test.pystone
Pystone(1.1) time for 50000 passes = 0.423339
This machine benchmarks at 118109 pystones/second
```

```
bartek@looper:~$ python -m test.pystone
Pystone(1.1) time for 50000 passes = 0.176478
This machine benchmarks at 283321 pystones/second
```

Moduły cProfile i profile

- W zasadzie identyczne co do działania i interfejsu
- Oba moduły pozwalają na policzenie czasu działania różnych fragmentów kodu
- Moduł **profile jest dużo wolniejszy**, ale napisany w pythonie, więc można go rozszerzać, **w praktyce stosujemy cProfile**
- Wywołujemy je przez

```
python -m cProfile plik.py -o plik.prof
```

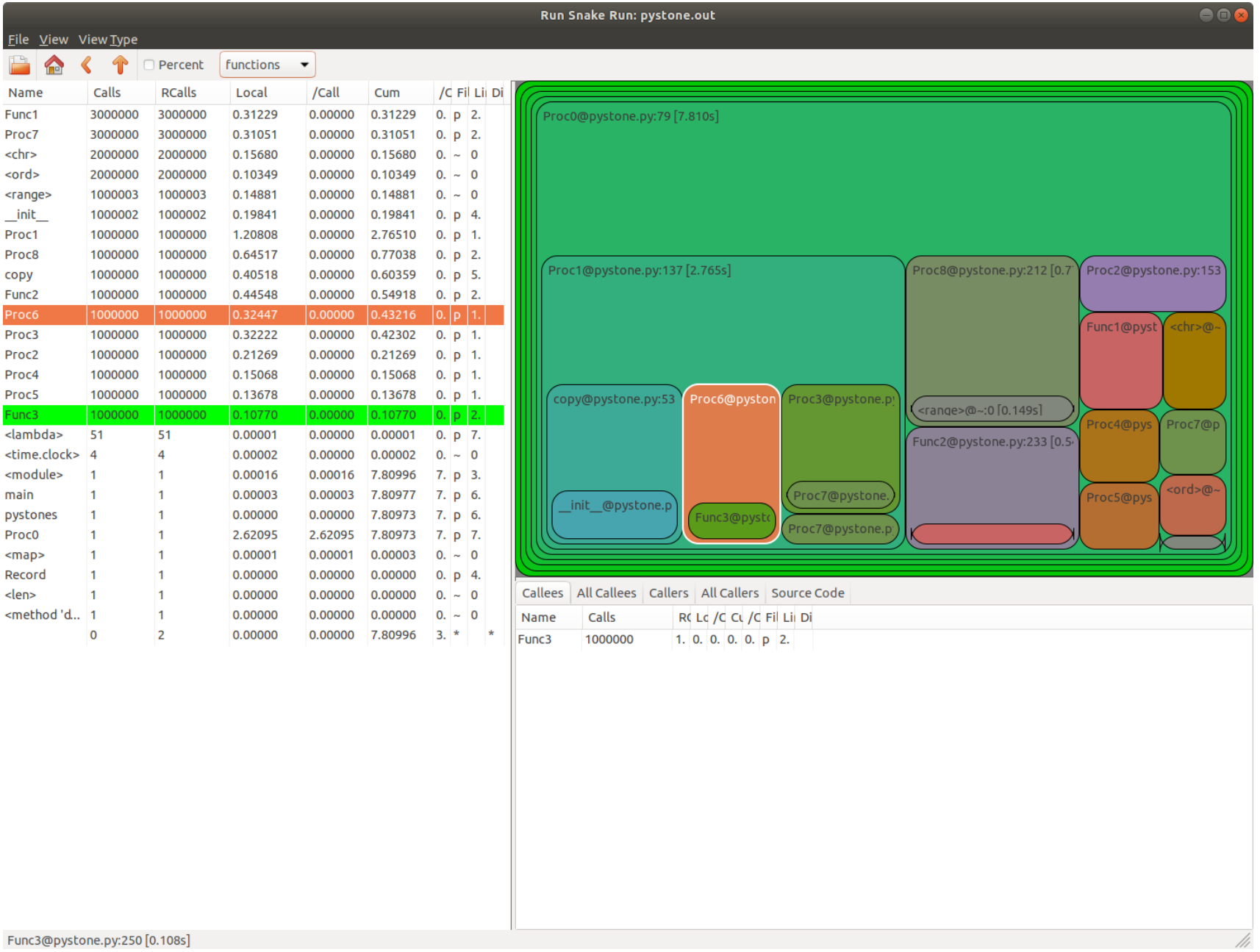
Wynik cprofile

```
bartek@Notch:~/Dropbox/Slides/Teaching/ASB/2019/profiling/py2$ python -m cProfile pystone.py
Pystone(1.1) time for 50000 passes = 0.386489
This machine benchmarks at 129370 pystones/second
         1100068 function calls in 0.393 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    50000    0.061    0.000    0.142    0.000 pystone.py:137(Proc1)
    50000    0.010    0.000    0.010    0.000 pystone.py:153(Proc2)
    50000    0.017    0.000    0.023    0.000 pystone.py:164(Proc3)
    50000    0.007    0.000    0.007    0.000 pystone.py:174(Proc4)
    50000    0.006    0.000    0.006    0.000 pystone.py:181(Proc5)
    50000    0.016    0.000    0.022    0.000 pystone.py:188(Proc6)
   150000    0.016    0.000    0.016    0.000 pystone.py:207(Proc7)
    50000    0.032    0.000    0.040    0.000 pystone.py:212(Proc8)
   150000    0.015    0.000    0.015    0.000 pystone.py:225(Func1)
    50000    0.022    0.000    0.027    0.000 pystone.py:233(Func2)
    50000    0.006    0.000    0.006    0.000 pystone.py:250(Func3)
        1    0.000    0.000    0.393    0.393 pystone.py:33(<module>)
        1    0.000    0.000    0.000    0.000 pystone.py:43(Record)
    50002    0.011    0.000    0.011    0.000 pystone.py:45(__init__)
    50000    0.021    0.000    0.031    0.000 pystone.py:53(copy)
        1    0.000    0.000    0.392    0.392 pystone.py:60(main)
        1    0.000    0.000    0.392    0.392 pystone.py:67(pystones)
       51    0.000    0.000    0.000    0.000 pystone.py:75(<lambda>)
        1    0.129    0.129    0.392    0.392 pystone.py:79(Proc0)
   100000    0.008    0.000    0.008    0.000 {chr}
        1    0.000    0.000    0.000    0.000 {len}
        1    0.000    0.000    0.000    0.000 {map}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
   100000    0.005    0.000    0.005    0.000 {ord}
    50003    0.009    0.000    0.009    0.000 {range}
        4    0.000    0.000    0.000    0.000 {time.clock}
```

RunSnakeRun do wizualizacji



Kernprof i line_profile

- Jeśli chcemy znać w danej funkcji czasy wykonania poszczególnych linii kodu, możemy użyć pakietu line_profiler
- Założywszy, że w naszym skrypcie skrypt.py umieściliśmy taki kod z dekoratorem:

```
@profile
def funkcja_do profilowania(a, b, c):
    ...
```

- To wykonanie następującego kodu pozwoli nam na zobaczenie profilu naszej funkcji:

```
$ pip install line_profiler
$ kernprof -l skrypt.py
$ python -m line_profiler script_to_profile.py.lprof
```

Memory profiler

- Podobne narzędzie do line_profiler’a to memory profiler
- Tu z kolei możemy zobaczyć zużycie pamięci z podziałem na linijki kodu.
- Dla każdej linijki mamy podane obecne zużycie oraz jego wzrost
- Wynik dostajemy tylko dla funkcji dekorowanych przez @profile

Moduły do obsługi komunikatów

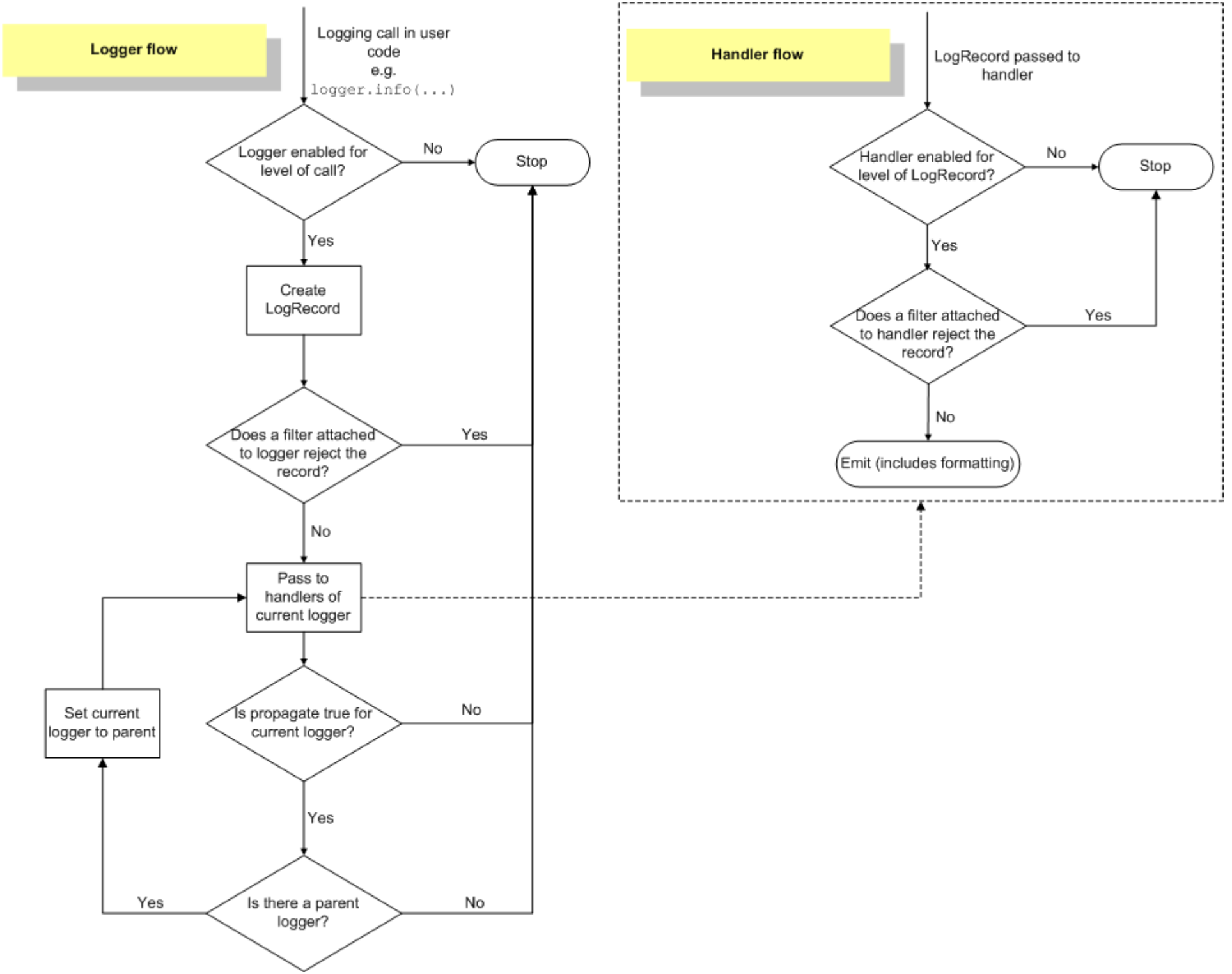
- Często mamy potrzebę wyświetlania, czy w inny sposób zapisywania, co się dzieje z długo działającym programem, np. serwerem
- Czasem są to po prostu informacje, czasem różnego rodzaju ostrzeżenia lub błędy, które nasz program może obsłużyć
- Typowo używamy w tym celu modułów logging i warnings

Co i jak wyświetlać

Task you want to perform	The best tool for the task
Display console output for ordinary usage of a command line script or program	`print()`
Report events that occur during normal operation of a program (e.g. for status monitoring or fault investigation)	`logging.info()` (or `logging.debug()` for very detailed output for diagnostic purposes)
Issue a warning regarding a particular runtime event	`warnings.warn()` in library code if the issue is avoidable and the client application should be modified to eliminate the warning `logging.warning()` if there is nothing the client application can do about the situation, but the event should still be noted
Report an error regarding a particular runtime event	Raise an exception
Report suppression of an error without raising an exception (e.g. error handler in a long-running server process)	`logging.error()`, `logging.exception()` or `logging.critical()` as appropriate for the specific error and application domain

Architektura modułu logging

- Instancje klasy Logger, zwykle dla każdego modułu z osobna, na których wywołujemy metody (debug(), info(), warn(), error(), critical())
- Handlers – czyli obsługa wiadomości – zwykle na konsolę lub do pliku
- Filtry, które pozwalają na wybieranie co dzieje się z wiadomościami
- Formattery – które obsługują co oprócz samej wiadomości jest logowane


Logger flow
Logging call in user code e.g. logger.info(...)
Logger enabled for level of call?
No
Stop
Yes
Create LogRecord
Does a filter attached to logger reject the record?
Yes
No
Pass to handlers of current logger
Is propagate true for current logger?
No
Yes
Set current logger to parent
Is there a parent logger?
Yes
No
Handler flow
LogRecord passed to handler
Handler enabled for level of LogRecord?
No
Stop
Yes
Does a filter attached to handler reject the record?
Yes
No
Emit (includes formatting)

Użyteczne Handlery wiadomości

1. `StreamHandler` instances send messages to streams (file-like objects).
2. `FileHandler` instances send messages to disk files.
3. `BaseRotatingHandler` is the base class for handlers that rotate log files at a certain point. It is not meant to be instantiated directly. Instead, use `RotatingFileHandler` or `TimedRotatingFileHandler`.
4. `RotatingFileHandler` instances send messages to disk files, with support for maximum log file sizes and log file rotation.
5. `TimedRotatingFileHandler` instances send messages to disk files, rotating the log file at certain timed intervals.
6. `SocketHandler` instances send messages to TCP/IP sockets. Since 3.4, Unix domain sockets are also supported.
7. `DatagramHandler` instances send messages to UDP sockets. Since 3.4, Unix domain sockets are also supported.
8. `SMTPHandler` instances send messages to a designated email address.
9. `SysLogHandler` instances send messages to a Unix syslog daemon, possibly on a remote machine.
10. `NTEventLogHandler` instances send messages to a Windows NT/2000/XP event log.
11. `MemoryHandler` instances send messages to a buffer in memory, which is flushed whenever specific criteria are met.
12. `HTTPHandler` instances send messages to an HTTP server using either `GET` or `POST` semantics.
13. `WatchedFileHandler` instances watch the file they are logging to. If the file changes, it is closed and reopened using the file name. This handler is only useful on Unix-like systems; Windows does not support the underlying mechanism used.
14. `QueueHandler` instances send messages to a queue, such as those implemented in the `queue` or `multiprocessing` modules.
15. `NullHandler` instances do nothing with error messages. They are used by library developers who want to use logging, but want to avoid the 'No handlers could be found for logger XXX' message which can be displayed if the library user has not configured logging. See Configuring Logging for a Library for more information.