Death by a Thousand Leaks

What statically-analysing 370 Python extensions looks like

Presented by
David Malcolm
<dmalcolm@redhat.com>

What is static analysis?

- Discovering properties of a program without running it
- Programs that analyze other programs
- Treating programs as data, rather than code
- In particular, automatically finding bugs in code

What kind of code will be analyzed?

For this talk:

The C code of Python extension modules

Prerequisites

- I’m going to assume basic familiarity with Python, and with either C or C++
- Hopefully you’ve used, debugged, or written a Python extension module in C (perhaps via SWIG or Cython)

Outline

- Intro to "cpychecker"
- How to run the tool on your own code
- How I ran the tool on lots of code
- What bugs came up frequently
- Recommendations on dealing with C and C++ from Python
- Q & A

cpychecker

**git clone **

git://git.fedorahosted.org/gcc-python-plugin.git

Docs: http://tinyurl.com/cpychecker

Part of my Python plugin for GCC

- 6500 lines of Python code implementing a static checker for C extension modules
- See also my PyCon US 2012 talk: Static analysis of Python extension modules using GCC https://us.pycon.org/2012/schedule/presentation/78/

fedora

Reference counting

- For every object:
  - "what do I think my reference count is?" aka "ob_refcnt" (the object's view of how many pointers point to it) *versus*
  - the reality of how many pointers point to it
- As a C extension module author you must manually keep these in sync using Py_INCREF and Py_DECREF.

Reference counting

The two kinds of bugs:

- **ob_refcnt too high**
  - memory leaks (hence the title of this talk)
- **ob_refcnt too low**
  - **BOOM!!**

Checking reference counts

- For each path through the function and PyObject*, it determines:
  - what the reference count ought to be at the end of the function (based on how many pointers point to the object)
  - what the reference count *is*
- It will issues warnings for any that are incorrect.

File: **input.c**

Function: **test**

Error: **ob_refcnt of '*list' is 1 too high**

```
22 PyObject *
23 test(PyObject *self, PyObject *args)
24 {
25     PyObject *list;
26     PyObject *item;
27     list = PyList_New(1);
```

when PyList_New() succeeds

PyListObject allocated at: ***list = PyList_New(1);***

ob_refcnt is now refs: 1 + N where N >= 0

```
28     if (!list)
```

taking False path

```
29         return NULL;
30     item = PyLong_FromLong(42);
```

when PyLong_FromLong() fails

```
31     /* This error handling is incorrect: it's missing an
32        invocation of Py_DECREF(list): */
33     if (!item)
```

taking True path

```
34         return NULL;
35     /* This steals a reference to item; item is not leaked when we get here: */
36     PyList_SetItem(list, 0, item);
37     return list;
38 }
```

ob_refcnt of '*list' is 1 too high

was expecting final ob_refcnt to be N + 0 (for some unknown N)

but final ob_refcnt is N + 1

Limitations of the refcount checking

- purely intraprocedural
  - assumes every function returning a PyObject* returns a new reference, rather than a borrowed reference
    - (...although you can manually mark functions with non-standard behavior)
  - it knows about most of the CPython API and its rules

Limitations of the refcount checking (2)

- only tracks 0 and 1 times through any loop, to ensure that the analysis doesn't go on forever
- can be defeated by relatively simple code (turn up --maxtrans argument)

What it checks for (2)

It checks for the following along all of those code paths:

- Dereferencing a NULL pointer (e.g. using result of an allocator without checking the result is non-NULL)
- Passing NULL to CPython APIs that will crash on NULL

What it checks for (3)

- Usage of uninitialized local variables
- Dereferencing a pointer to freed memory
- Returning a pointer to freed memory
- Returning NULL without setting an exception

What it checks for (4)

It also does some simpler checking:

- type in calls to PyArg_ParseTuple et al
- types and NULL termination of PyMethodDef tables
- types and NULL termination of PyObject_Call{Function|Method}ObjArgs

What it doesn't check for

(patches welcome!)

- tp_traverse errors (which can mess up the garbage collector); missing it altogether, or omitting fields
- errors in GIL handling
- lock/release mismatches
- missed opportunities to release the GIL (e.g. compute-intensive functions; functions that wait on IO/syscalls)

What it can't check for

Does the code
"do the right thing"?

How to run it on your own code

```
git clone \
    git://git.fedorahosted.org/gcc-python-plugin.git
```

Dependencies

(on Fedora)

```
sudo yum install \
    gcc-plugin-devel \
    python-devel \
    python-six \
    python-pygments \
    graphviz
```

Building the checker

Building the checker:

```
make plugin
```

Checking that it's working:

```
make demo
```

```
demo.c: In function ‘make_a_list_of_random_ints_badly’:
demo.c:90:26: warning: Mismatching type in call to PyArg_ParseTuple with
  argument 3 ("&count") had type
    "long int *" (pointing to 64 bits)
  but was expecting
    "int *" (pointing to 32 bits)
  for format code "i"
demo.c:102:1: warning: ob_refcnt of '*item' is 1 too high [enabled by de
demo.c:102:1: note: was expecting final ob_refcnt to be N + 1 (for some
demo.c:102:1: note: due to object being referenced by: PyListObject.ob_i
demo.c:102:1: note: but final ob_refcnt is N + 2
demo.c:97:14: note: PyLongObject allocated at:          item = PyLong_Fro
demo.c:90:26: note: when PyArg_ParseTuple() succeeds at:     if (!PyArg_
demo.c:90:8: note: taking False path at:     if (!PyArg_ParseTuple(args,
demo.c:94:10: note: reaching:     list = PyList_New(0);
demo.c:94:10: note: when PyList_New() succeeds at:     list = PyList_New
demo.c:96:5: note: when considering range: 1 <= count.0 <= 0x7fffffff at
demo.c:96:5: note: taking True path at:     for (i = 0; i < count; i++)
demo.c:97:31: note: reaching:          item = PyLong_FromLong(random());
demo.c:97:14: note: when PyLong_FromLong() succeeds at:          item = P
demo.c:97:14: note: ob_refcnt is now refs: 1 + N where N >= 0
demo.c:98:22: note: when PyList_Append() succeeds at:          PyList_App
demo.c:98:22: note: ob_refcnt is now refs: 2 + N where N >= 0
demo.c:98:22: note: '*item' is now referenced by 1 non-stack value(s): P
demo.c:96:5: note: when considering count.0 == (int)1 from demo.c:90 at:
demo.c:96:5: note: taking False path at:     for (i = 0; i < count; i++)
demo.c:101:5: note: reaching:     return list;
```

File: **input.c**

Function: **test**

Error: **ob_refcnt of '*list' is 1 too high**

```
22 PyObject *
23 test(PyObject *self, PyObject *args)
24 {
25     PyObject *list;
26     PyObject *item;
27     list = PyList_New(1);
```

when PyList_New() succeeds

PyListObject allocated at: ***list = PyList_New(1);***

ob_refcnt is now refs: 1 + N where N >= 0

```
28     if (!list)
```

taking False path

```
29         return NULL;
30     item = PyLong_FromLong(42);
```

when PyLong_FromLong() fails

```
31     /* This error handling is incorrect: it's missing an
32        invocation of Py_DECREF(list): */
33     if (!item)
```

taking True path

```
34         return NULL;
35     /* This steals a reference to item; item is not leaked when we get here: */
36     PyList_SetItem(list, 0, item);
37     return list;
38 }
```

ob_refcnt of '*list' is 1 too high

was expecting final ob_refcnt to be N + 0 (for some unknown N)

but final ob_refcnt is N + 1

Building with it

Distutils
CC=/path/to/built/plugin/gcc-with-cpychecker \
  python setup.py build
to set the environment variable
Makefiles
make CC=/path/to/built/plugin/gcc-with-cpychecker
to override the Makefile variable CC.

Let us know how you get on!

Mailing list:

- gcc-python-plugin@lists.fedorahosted.org
- See: https://fedorahosted.org/mailman/listinfo/gcc-python-plugin

Analyze all the things!

- The goal: analyze all of the C Python extensions in a recent Linux distribution
  - Specifically: all of the Python 2 C code in Fedora 17
    - Every source rpm that builds something that links against libpython2.7
    - 370(ish) packages
- The reality:
  - Some unevenness in the data coverage, so take my numbers with a pinch of salt
  - Lots of bugfixing as I went...

Running cpychecker a lot

Scaling up to hundreds of projects:

- building via RPM
  - hides the distutils vs Makefile vs CMake etc
- "mock" builds
  - every build gets its own freshly-provisioned chroot
- Use this to reliably inject static analysis...

"mock-with-analysis"

Running checkers:

- cpychecker
- cppcheck
- clang-analyzer
- gcc warnings

https://github.com/fedora-static-analysis/mock-with-analysis

fedora

Scaling up (continued)

- separation of model from presentation
  - "Firehose" XML format:
  - https://github.com/fedora-static-analysis/firehose
- detect analyzers that fail or exceed 1 minute to run
- store the result in a database
- capture any sources mentioned in a report
- can also capture arbitrary data e.g. code metrics

```
382             free(expr); expr = NULL;
383     }
384     goto cleanup;
385
386 err:
387     error = errno;
388     py_decref(dict);
389     py_decref(tuple);
390     PyErr_SetString(PyExc_RuntimeError,strerror(error));
391 cleanup:
392     free(tmp);
393     free(rule_str);
394     free(expr);
395     errno = error;
396     return output;
397 }
```

ob_refcnt of return value is 1 too low

(emitted by cpychecker)

TODO: a detailed trace is available in the data model (not yet rendered in this report)

```
398
399 static int perform_ft_query(const apol_policy_t * policy, const options_t * opt, apol_vector_
400 {
401     apol_filename_trans_query_t *ftq = NULL;
402     size_t i;
403     int error = 0;
404
405     if (!policy || !opt || !v) {
406             PyErr_SetString(PyExc_RuntimeError,strerror(EINVAL));
407             errno = EINVAL;
```

```
193 static PyObject *get_ipaddress(PyObject *self __unused, PyObject *args)
194 {
195     struct ifreq ifr;
196     int fd, err;
197     char *devname;
198     char ipaddr[20];
199
200     if (!PyArg_ParseTuple(args, "s", &devname))
```

Mismatching type in call to PyArg_ParseTuple with format code "s"

*argument 3 ("&devname") had type "char * *" but was expecting "const char * *" for format code "s"*

(emitted by cpychecker)

```
201             return NULL;
202
203     /* Setup our request structure. */
204     memset(&ifr, 0, sizeof(ifr));
205     strncpy(&ifr.ifr_name[0], devname, IFNAMSIZ);
206     ifr.ifr_name[IFNAMSIZ - 1] = 0;
```

```
193 static PyObject *get_ipaddress(PyObject *self __unused, PyObject *args)
194 {
195     struct ifreq ifr;
196     int fd, err;
197     const char *devname;
198     char ipaddr[20];
199
200     if (!PyArg_ParseTuple(args, "s", &devname))
201             return NULL;
202
203     /* Setup our request structure. */
204     memset(&ifr, 0, sizeof(ifr));
205     strncpy(&ifr.ifr_name[0], devname, IFNAMSIZ);
206     ifr.ifr_name[IFNAMSIZ - 1] = 0;
```

fedora™

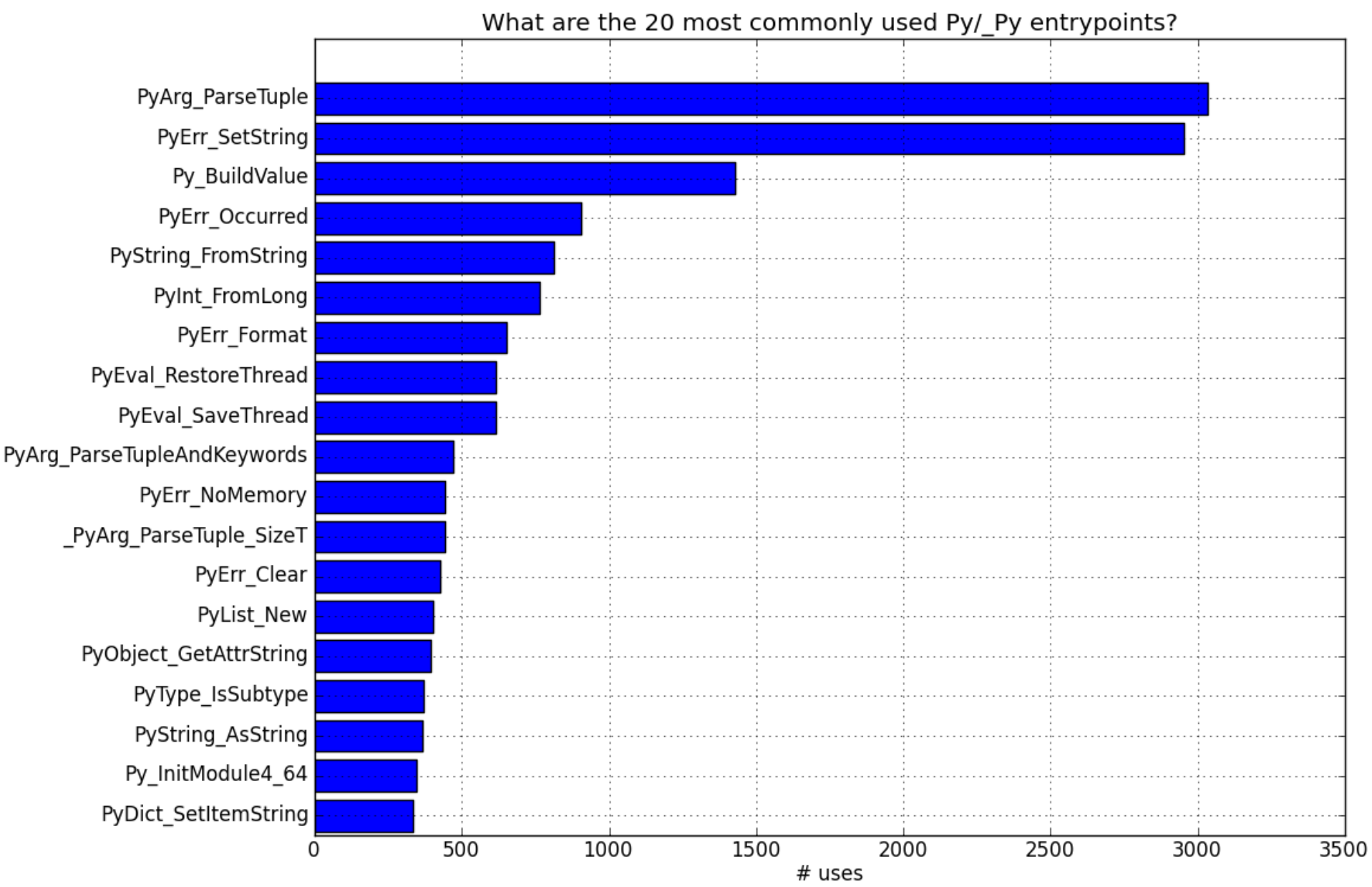
```
    TracerType.tp_new = PyType_GenericNew;
    if (PyType_Ready(&TracerType) < 0) {
        return;
    }

    Py_INCREF(&TracerType);
    PyModule_AddObject(mod, "Tracer", (PyObject *)&TracerType);
```

Failure runninng cpychecker (python-exception)

```
traceback: Traceback (most recent call last):
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/__init__.py", line 131, in _check_refcounts
    self.options)
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/refcounts.py", line 4355, in check_refcounts
    rep = impl_check_refcounts(ctxt, fun, options)
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/refcounts.py", line 4213, in impl_check_refcounts
    limits=limits)
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/absinterp.py", line 3089, in iter_traces
    depth + 1):
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/absinterp.py", line 3089, in iter_traces
    depth + 1):
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/absinterp.py", line 3089, in iter_traces
    depth + 1):
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/absinterp.py", line 3089, in iter_traces
    depth + 1):
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/absinterp.py", line 3089, in iter_traces
    depth + 1):
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/absinterp.py", line 3089, in iter_traces
    depth + 1):
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/absinterp.py", line 3089, in iter_traces
    depth + 1):
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/absinterp.py", line 3089, in iter_traces
    depth + 1):
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/absinterp.py", line 3089, in iter_traces
    depth + 1):
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/absinterp.py", line 3089, in iter_traces
    depth + 1):
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/absinterp.py", line 3089, in iter_traces
    depth + 1):
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/absinterp.py", line 3089, in iter_traces
    depth + 1):
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/absinterp.py", line 3053, in iter_traces
    transitions = curstate.get_transitions()
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/absinterp.py", line 2105, in get_transitions
    return self._get_transitions_for_stmt(stmt)
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/absinterp.py", line 2121, in _get_transitions_for_stmt
    return self._get_transitions_for_GimpleCall(stmt)
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/absinterp.py", line 2292, in _get_transitions_for_GimpleCall
    return meth(stmt, *args)
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/refcounts.py", line 2544, in impl_PyModule_AddObject
    s_success.cpython.steal_reference(v_value, stmt.loc)
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/refcounts.py", line 741, in steal_reference
    _steal_ref)
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/libcpychecker/refcounts.py", line 551, in change_refcount
    check_isinstance(oldvalue, RefcountValue)
  File "/usr/lib/gcc/x86_64-redhat-linux/4.7.2/plugin/python2/gccutils.py", line 642, in check_isinstance
    raise TypeError('%s / %r is not an instance of %s' % (obj, obj, types))
TypeError: (long int)val [-0x7fffffffffffffff <= val <= 0x7fffffffffffffff] from coverage/tracer.c:714 / WithinRange(gcctype='long int', loc=gcc.l
```

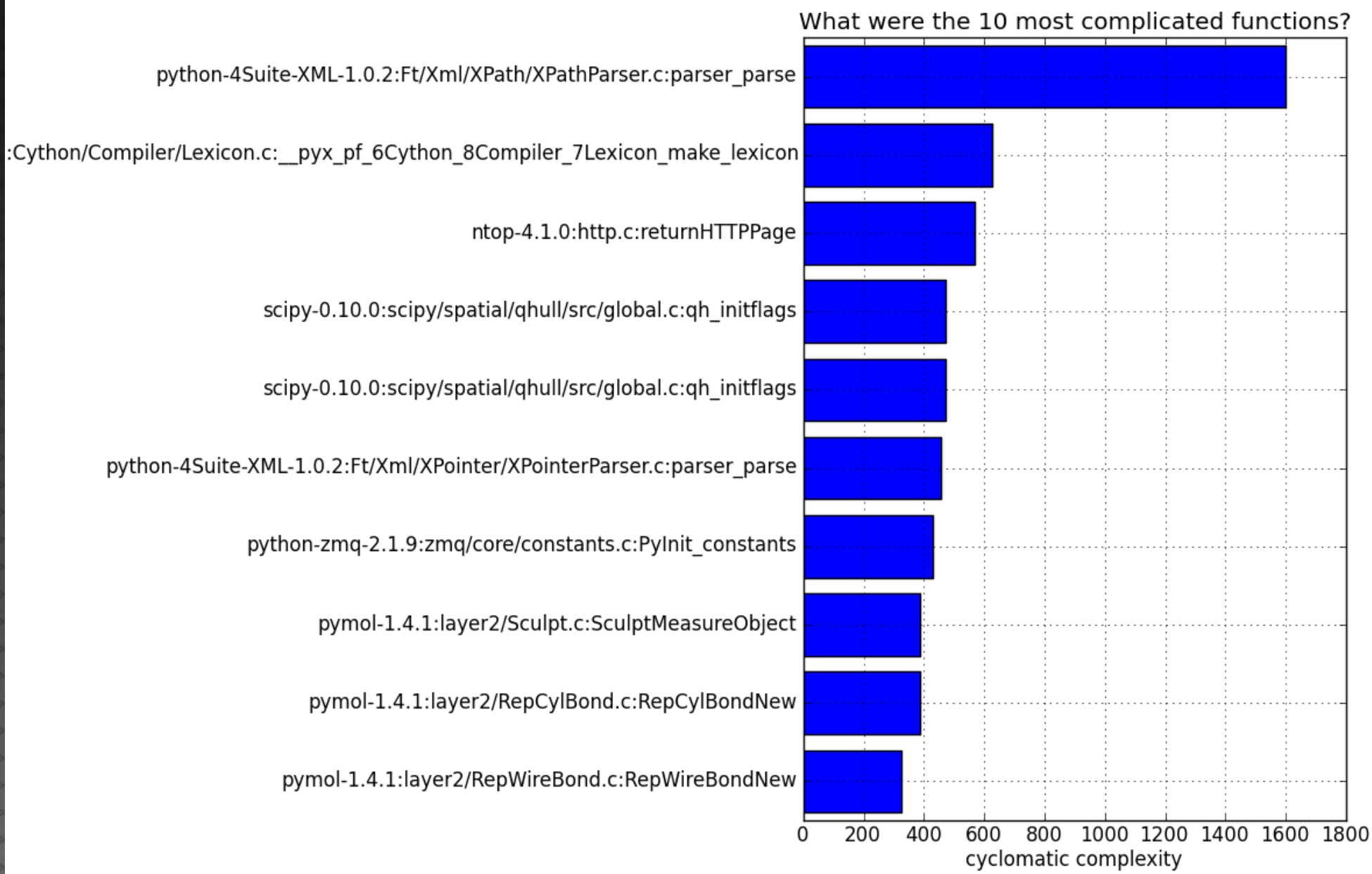
```
}
```


What are the 20 most commonly used Py/_Py entrypoints?
PyArg_ParseTuple
PyErr_SetString
Py_BuildValue
PyErr_Occurred
PyString_FromString
PyInt_FromLong
PyErr_Format
PyEval_RestoreThread
PyEval_SaveThread
PyArg_ParseTupleAndKeywords
PyErr_NoMemory
_PyArg_ParseTuple_SizeT
PyErr_Clear
PyList_New
PyObject_GetAttrString
PyType_IsSubtype
PyString_AsString
Py_InitModule4_64
PyDict_SetItemString
0
500
1000
1500
2000
2500
3000
3500
uses

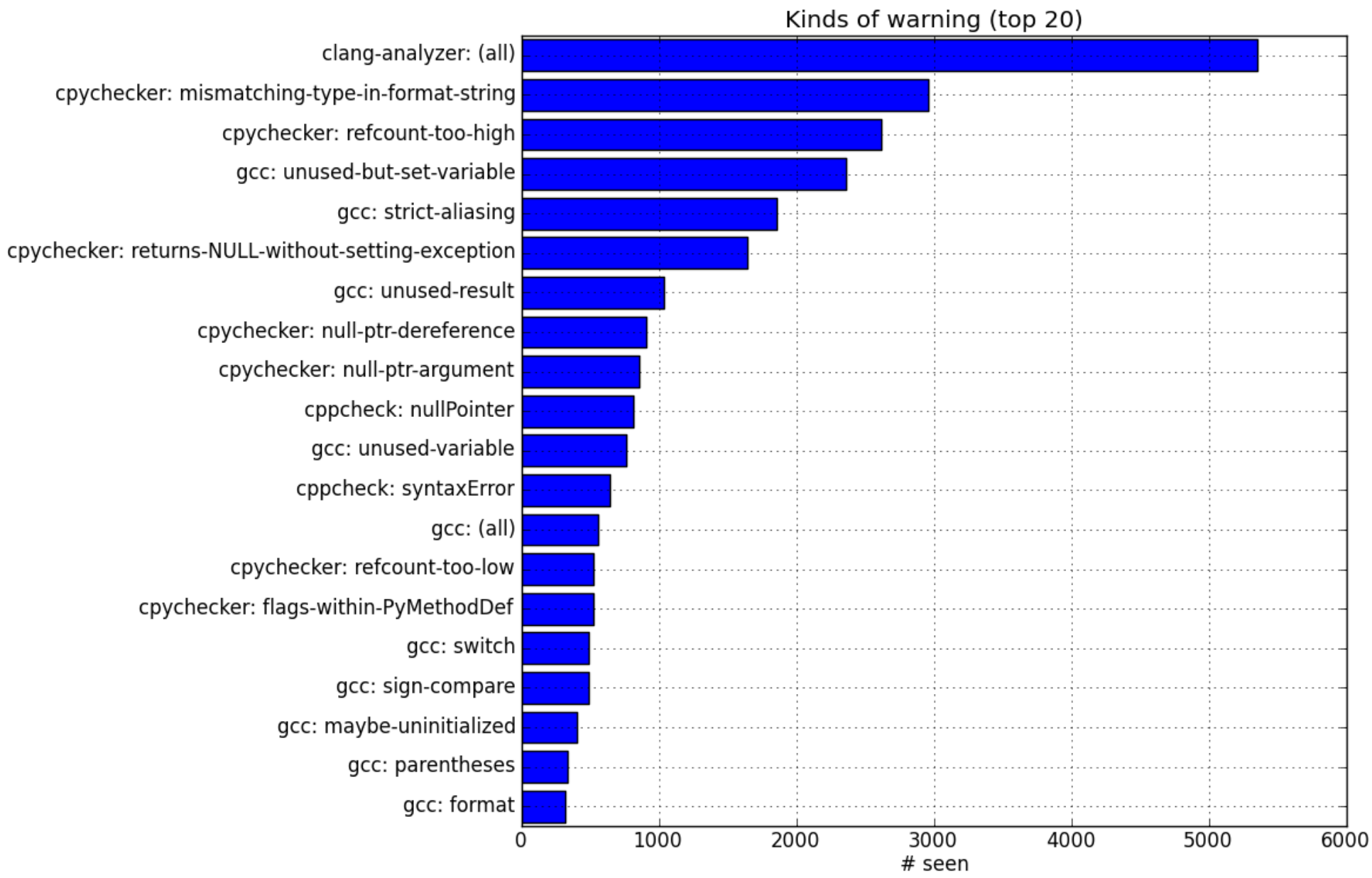
What are the least commonly used Py/_Py entrypoints?

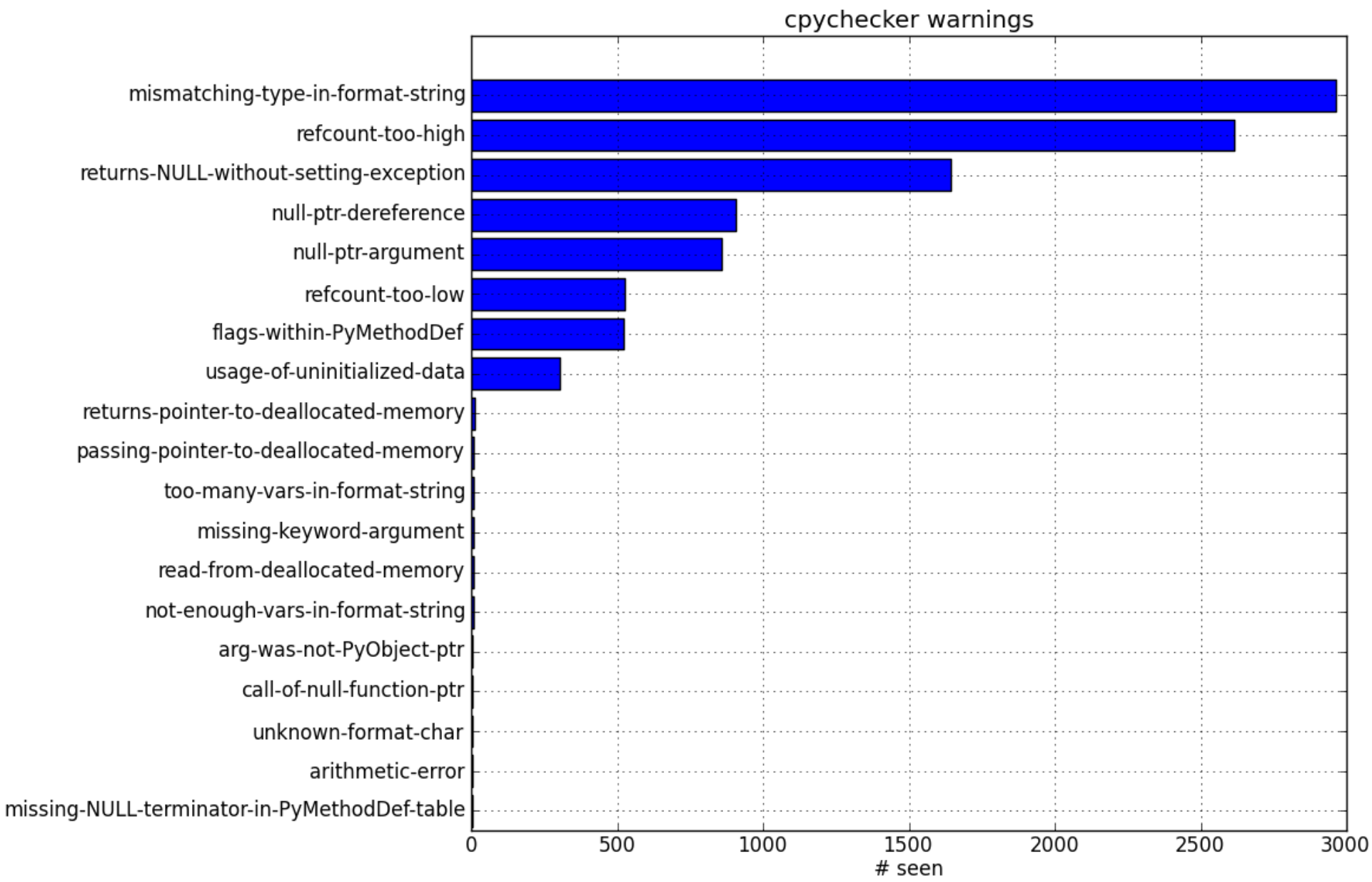
- There are many with just 1 user, but most of these are false positives:
- about 50 actual CPython API entrypoints with just one user
- about 100 "entrypoints" due to other projects reusing the prefix

(see source code of this talk if you're interested in the data: https://github.com/davidmalcolm/PyCon-US-2013-Talk


What were the 10 most complicated functions?
python-4Suite-XML-1.0.2:Ft/Xml/XPath/XPathParser.c:parser_parse
:Cython/Compiler/Lexicon.c:__pyx_pf_6Cython_8Compiler_7Lexicon_make_lexicon
ntop-4.1.0:http.c:returnHTTPPage
scipy-0.10.0:scipy/spatial/qhull/src/global.c:qh_initflags
scipy-0.10.0:scipy/spatial/qhull/src/global.c:qh_initflags
python-4Suite-XML-1.0.2:Ft/Xml/XPointer/XPointerParser.c:parser_parse
python-zmq-2.1.9:zmq/core/constants.c:PyInit_constants
pymol-1.4.1:layer2/Sculpt.c:SculptMeasureObject
pymol-1.4.1:layer2/RepCylBond.c:RepCylBondNew
pymol-1.4.1:layer2/RepWireBond.c:RepWireBondNew
0
200
400
600
800
1000
1200
1400
1600
1800
cyclomatic complexity

What did the analyzers complain about?


Kinds of warning (top 20)
clang-analyzer: (all)
cpychecker: mismatching-type-in-format-string
cpychecker: refcount-too-high
gcc: unused-but-set-variable
gcc: strict-aliasing
cpychecker: returns-NULL-without-setting-exception
gcc: unused-result
cpychecker: null-ptr-dereference
cpychecker: null-ptr-argument
cppcheck: nullPointer
gcc: unused-variable
cppcheck: syntaxError
gcc: (all)
cpychecker: refcount-too-low
cpychecker: flags-within-PyMethodDef
gcc: switch
gcc: sign-compare
gcc: maybe-uninitialized
gcc: parentheses
gcc: format
0
1000
2000
3000
4000
5000
6000
seen


cpychecker warnings
mismatching-type-in-format-string
refcount-too-high
returns-NULL-without-setting-exception
null-ptr-dereference
null-ptr-argument
refcount-too-low
flags-within-PyMethodDef
usage-of-uninitialized-data
returns-pointer-to-deallocated-memory
passing-pointer-to-deallocated-memory
too-many-vars-in-format-string
missing-keyword-argument
read-from-deallocated-memory
not-enough-vars-in-format-string
arg-was-not-PyObject-ptr
call-of-null-function-ptr
unknown-format-char
arithmetic-error
missing-NULL-terminator-in-PyMethodDef-table
0
500
1000
1500
2000
2500
3000
seen

Refcounting warnings

- refcount-too-high: 2614 times
- refcount-too-low: 524 times

fedora

Missing Py_INCREF() on Py_None

7% of the refcount-too-low warnings

(occurred 39 times (within 370 packages)

```
PyObject*
some_method(PyObject *self, PyObject *args)
{
    [...snip...]

    /* BUG: loses a reference to Py_None */
    return Py_None;
}
```

```
$ python script.py
Fatal error: deallocating None
```

fedora

Fixing Py_INCREF on Py_None

```
PyObject*
some_method(PyObject *self, PyObject *args)
{
    [...snip...]

    /* Fixed version of the above: */
    Py_RETURN_NONE;
}
```

Reference leak in Py_BuildValue with "O"

```
/* BUG: reference leak: */
return Py_BuildValue("Oi", some_object_we_own_a_ref_on, 42);
```

```
/* Fixed version of the above: */
return Py_BuildValue("Ni", some_object_we_own_a_ref_on, 42);
```

```
/* If it's just one object, why use Py_BuildValue? */
return some_object_we_own_a_ref_on;
```

1700+ places lacking error checking

- null-ptr-dereference: 907
- null-ptr-argument: 857

"goto" considered wonderful

```
{
    PyObject *local0 = NULL;
    PyObject *local1 = NULL;
    PyObject *local2 = NULL;
    /* etc */

    local0 = PyFoo_DoBar();
    if (!local0) goto error;

    /* etc */

    return result;

error:
    Py_XDECREF(local2);
    Py_XDECREF(local1);
    Py_XDECREF(local0);
    return NULL;
}
```

DO NOT DO THIS...

```
Py_XDECREF(PyObject_CallObject(callable, args));
```

How the compiler sees it...

```
do {
  if ((PyObject_CallObject(callable, args)) == ((void *)0))
    ;
  else
    do {
      if (--(PyObject_CallObject(callable, args)->ob_refcnt) != 0)
        ;
      else
        (*(PyObject_CallObject(callable, args)->ob_type)->tp_dealloc)
          PyObject_CallObject(callable, args);
    } while (0);
} while (0);
```

Filed as http://bugs.python.org/issue17206

fedora

How the compiler sees it...

```
/* Call it once */
if ((PyObject_CallObject(callable, args)) != NULL) {
    /*
       If it doesn't raise an exception, leak the reference (BUG 1),
       and call it again (BUG 2).

       Assume that the second call doesn't raise an exception,
       otherwise segfault the interpreter (BUG 3),
       and DECREF the result, but don't deallocate if the refcount
       is zero (BUGS 4 and 5)
     */
    if (--(PyObject_CallObject(callable, args)->ob_refcnt) == 0) {
      /*
        If the refcount is zero, call it again! (BUG 6)
        Assume the result is non-NULL (otherwise segfaulting, BUG 7)
        and deallocate whatever you got back (even if the refcount
        is non-zero, BUG 8)
       */
      (*(PyObject_CallObject(callable, args)->ob_type)->tp_dealloc)
        /* and for good measure, call it agains (BUG 9)
           and leak a reference to the result (BUG 10) */
        PyObject_CallObject(callable, args);
    }
}
```

Filed as http://bugs.python.org/issue17206

fedora

The correct way to discard the result

```
PyObject *result;
result = PyObject_CallObject(callable, args);
Py_XDECREF(result);

/* Presumably the caller will do something about any exception: */
return (result != NULL) ? 0 : -1;
```

Dealing with C and C++ from Python

- Do you **really** need C?
- Can you get away with pure Python code?
- Consider using Cython
- ctypes is good, but has its own issues
- cffi?
- If you must use C, run cpychecker on your code

In conclusion

- Intro to "cpychecker"
- How to run the tool on your own code
- How I ran the tool on lots of code
- What bugs came up frequently
- Recommendations on dealing with C and C++ from Python

Thanks for listening!

Q & A

git clone \

git://git.fedorahosted.org/gcc-python-plugin.git

- cpychecker's mailing list:
  https://fedorahosted.org/mailman/listinfo/gcc-python-plugin
- This talk:
  - https://github.com/davidmalcolm/PyCon-US-2013-Talk

fedora