Dr. Dobb's Journal

February 2014

NIMROD

A new systems programming language that rivals C in performance and compiles to C, C++, Objective-C, and JavaScript >>

Next

ALSO INSIDE

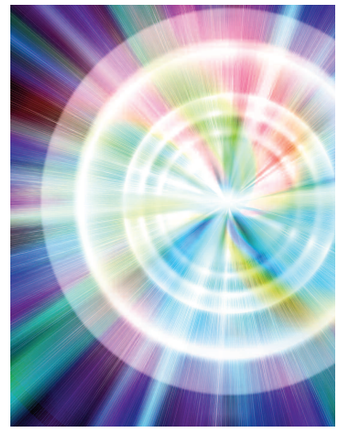
UBM Tech

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

February 2014



FEATURES

GUEST EDITORIAL

More on DrDobbs.com

The Rise And Fall of Languages in 2013
Much happened in languages during a year that appeared static.
http://www.drdobbs.com/240165192

Engineering Managers Should Code 30% of Their Time
Lose contact with the code, and you lose the connection to your team and the project. How then to make the time to manage and code? The cofounder of MongoDB explains his approach.
http://www.drdobbs.com/240165174

Scrum's Flawed Notion of Product Owner
Agile processes replace up-front interviews with ongoing interviews. Without an on-site customer, there are no interviews, and you end up with no design.
http://www.drdobbs.com/240165148

Hackathons: Proof That Ideas Are Cheap and Implementation Is Expensive
Hackathons, those 48-hour fests fueled by energy drinks and pizza, are a rapidly growing phenomenon, but what's in it for developers?
http://www.drdobbs.com/240164937

Social Processes and the Design of Programming Languages
"A proof does not in itself significantly raise our confidence in the probable truth of the theorem it purports to prove."
http://www.drdobbs.com/240165221

How I Came to Write D

The path that led Walter Bright to write a language, now among the top 20 most used, began with curiosity — and an insult.

By Walter Bright

I've liked figuring out how things worked and designing them from a very early age (I loved the Tom Swift books). In an earlier age, I could see myself designing locomotives or Schneider Trophy racers. My degree was in mechanical engineering. But mechanical engineering was frustrating because of the large expense involved with building anything, and my rather poor fabrication skills.

With programming, on the other hand, I could build the most intricate machines imaginable at no expense, needing only access to a computer. Like many programmers, I started with writing games. The appeal of writing games is sort of a God complex — you get to create a whole world that slavishly follows your rules. With the game Empire (http://www.classicempire.com/), not only did I create such a world, but I spent endless hours figuring out how to make the computer best operate the enemy armies. Even now, I still think of ways to improve it.

The trouble, though, was that the computer strategy's thirst for compute power far exceeded the abilities of the machines available, and so I became interested in how the compiler optimized the code. Compilers seemed like utterly magical devices that transformed source into machine code. How did this sorcery actually work? This was one of the Great Mysteries Of The Universe until *BYTE* magazine published the source code to Tiny Pascal. I devoured every line of that program; and once mastered, I felt I'd been given the password to the inner sanctum.

Some years went by. In the early '80s, I found myself part of a programming team developing software for MS-DOS. We were using C because it was the only high-level language we could find that actually worked on the PC. The other language implementations were unbelievably awful. Even the C compilers were execrable, but at least they were usable. The code they generated was terrible, the optimization nonexistent. I had the idea I could write a better C compiler myself.

I confided this to a colleague, and he suggested a lunch with him and the local C guru, who'd give me some advice on how to proceed. We went and my friend explained my ambition to the guru. His contemptuous response is still etched on my brain: "Who the hell do you think you are thinking you can write a C compiler?"

"I invented the Empire game because it was the game I wanted to play. I developed a C compiler because I needed a better compiler to develop Empire (and other projects) with."

I have to retroactively thank him for that because I found the desire to show him up to be powerfully motivating. I went on to implement the C compiler, known as Datalight C. True to my interest in optimization, it was the first on the PC to have a data-flow optimizing compiler. Such a concept was new enough that the compiler got into trouble in the computer magazine benchmarks because the optimizer figured out that the benchmarks did nothing and so deleted all that dead code — the journalist assumed my compiler was broken or cheating and Datalight C got a bad review. (This, of course, made me pretty mad, but there was no ubiquitous Internet in those days where I could post a riposte.)

It was later reincarnated as Zortech C. By then, there were many other C compilers on the PC (I think I counted 30 at one point). I was looking around for a competitive edge, and found Bjarne Stroustrup's book on C++ in the bookstore. I thought, "Heh, add a couple of new keywords, and in a couple of months, I'll have a C++ compiler!" Probably the understatement of the programming century. If I'd known what I was getting into, perhaps I would have believed the people who told me I couldn't do it.

Anyhow, the late 1980s was a time when lots of people were working on successors to C.

You can even find references to one such project called "D" on usenet of the time. For various reasons, including the fact that Zortech had an inexpensive C++ compiler ready to go on the most popular platform of the day, C++ buried those other languages and dominated the 1990s of programming; and I was carried along with that boom.

Along the way (we're now in the 1990s), I also wrote a Java compiler that generated native code, and a JavaScript compiler/interpreter. These products were unsuccessful. I'll note here that working on stuff that I needed has fared quite a bit better than working on stuff that I was told others need.

For example, I was out jogging one day with a programmer friend who said, "You know, what the world is desperate for is a Java compiler that generates native code. You'll make a mint off of that! I use Java and this is really needed." I told him that, coincidentally, I had written one and he could start using it right away. Of course, he never did.

I invented the Empire game because it was the game I wanted to play. I developed a C compiler because I needed a better compiler to develop Empire (and other projects) with. The Java and JavaScript compilers were other peoples' ideas. But back to where D came from.

Nobody could possibly work on compilers for 15+ years and not come up with ideas for language improvements. I tried out many language improvements in the C and C++ compilers, and they all fell flat.

Nobody was interested in language extensions — they wanted standard compliant languages, and they were right. Unfortunately, getting ideas adopted by standards committees is an arduous, multiyear bureaucratic process — a process I am not very patient with.

"As it stands, D wouldn't exist without the Internet."

In 1999, I decided I was retired and spent about six weeks watching TV until I thought I'd go mad.

It was time to get back to living. Whining about perceived problems with existing languages had gone on long enough; I decided to power up the machine shop. When tackling a problem like this, I am always reminded of Gimli the dwarf http://is.gd/31jCja0: "Certainty of death. Small chance of success. What are we waiting for?" Why not? At least I'll go down sword in hand fighting the glorious fight.

So D started out, stumbling along as a solo project. I've never been comfortable programming in a language I hadn't written a compiler for (I know, another oddity), but I'd had enough experience implementing various languages that I was pretty confident I could get D working. I also by now had the huge advantage of the existing C++ compiler ecosystem to work with.

A couple years later, D first appeared on Slashdot and it rapidly started attracting users and collaborators. Turns out, I am hardly a unique person in what I want from a language! D grew dramatically in ambition, with collaborators from all over the world. It wasn't until a few months ago at Dconf2013 that we even knew what each other looked like. (This is one of the greatest aspects of the Internet revolution: You can work successfully with others while knowing nothing about their sex, age, looks, race, religion, language, culture, disabilities, histories, etc. It's as pure a meritocracy as it gets. Only your ideas, contributions, and how you present yourself matter.)

As it stands, D wouldn't exist without the Internet. How else could disparate enthusiasts have ever gotten together? The rise of collaborative tools like github and bugzilla have been absolutely critical to D's development (along with the open source model, of course!).

But I've had to learn things, too, often the hard way:

- My natural tendency is to work solo on things. My work performance reviews at the various jobs I've had usually included comments like "Walter needs to learn to work better with others." My desire to control everything nearly wrecked the D community at one point, and I've had to change.
- I've had to learn how to manage a project where people are all volunteers. Since I don't pay anybody anything, I can't tell anybody to do anything. I have to find other ways. I rank pretty close to "nerd" on personality tests,so motivating people is not natural for me. This is actually a fascinating challenge.
- In the D forums, proposals for new language features come every day. I have to say "no" a lot, which is hard to do, and harder for the proposers to hear.
- Management is a hard job, and I'm not well-suited for it. I didn't do a very good job of it at Zortech; and at various corporate jobs, I was (wisely) never promoted into management. But with D, it

couldn't be avoided — I have had to learn how to do it, and do it reasonably well. I'm pretty motivated to try and be a better manager, because I want D to succeed.

- As my friends know, I love to debate. I'm a participant in many Internet flame wars. But I can't do that with D because nobody ever changes their mind as a result of a flame war, no matter how wrong they are :-). All this experience actually lends a curious advantage — I can tell when a conversation thread is starting to slip into unproductive argument, and take steps to lower the voltage and defuse it.

  I still have to fight the urge to go "Shields up! Phasers on vaporize!" Of course, as anyone on the D forums will agree, this is a work in progress for me. D has been and continues to be an amazing experience — it's borrowed the best ideas from many other languages, and is the summation of enormous and selfless effort by many, many collaborators from all over the world, who can all be justly proud of what they've accomplished.

There has been one result of all this that I especially treasure — I love programming in D. It's the language I wish I'd always had available. I'm glad I didn't listen to the naysayers, the Debbie Downers, and of course that nameless C guru from long ago, whom I owe thanks to.

So, you want to write your own language? All I can say is: Certainty of death. Small chance of success. What are you waiting for?

Acknowledgment

Thanks to Andrei Alexandrescu for his helpful suggestions on this article.

— Walter Bright is the designer of the D language. As mentioned here, he previously wrote the Datalight and Zortech C++ compilers, as well as several games. He regularly blogs for **Dr. Dobb's.**

Comment

News Briefs

By Adrian Bridgwater

One API For Box, Dropbox, Google Drive, Sky Drive, and SharePoint

Cloud Integration Platform-as-a-Service company Cloud Elements has announced Documents Hub, a "one-to-many" integration platform service. This developer tool provides a single API to integrate Box, Dropbox, Google Drive, Sky Drive, and Microsoft SharePoint — all through a uniform set of REST API calls. The suggestion here is that developers can now connect to an entire category of services such as documents, CRM, marketing and accounting, etc. There is a single console to provision, integrate, monitor, and maintain these services. Additional document and file services will regularly be added to the Documents Hub, including support early next year for Amazon S3 and Rackspace Files.
http://www.drdobbs.com/240165298

Coverity Covers C/C++, Java, and C#

The Coverity Development Testing Platform 7.0 arrives this week with claims of being the industry's first enterprise-scale solution that combines code analysis, change-aware unit test analysis, and policy management across C/C++, Java, and C#. Twenty-one new (or enhanced) C# analysis algorithms further for defect detection in C# codebases will provide developers (in theory) with the information they need to fix quality defects, such as resource leaks, concurrency issues, and null references. There is also expanded Java Web application security coverage in the form of expanded coverage for the Open Web Application Security Project (OWASP) Top 10 and Common Weakness Enumerations (CWE) security vulnerabilities in Java applications. Along with 17 new analysis algorithms for Java and C/C++ codebases, the platform allows the import of critical defects into SonarQube, a popular open source quality management platform used in Java development, to view and manage a broader range of defects in Java applications within a single workflow.
http://www.drdobbs.com/240165223

Dart Language Gets New Technical Committee

Ecma has created a new technical committee named TC52 to publish a standard specification of the Dart complex web application language. Dart is intended to help developers build HTML5 apps for the Web, fast, hence the name. Dart has been described as a "batteries included" developer platform that integrates the language, libraries, an editor, a virtual machine, and a compiler with JavaScript output, too.
http://www.drdobbs.com/240165219

Microsoft M#, An Extension Of C#

Microsoft has quietly blogged references to what may be M#. This new systems-level programming language is hoped to be eventually open sourced and will exist as a language that gives developers type safety and productivity plus performance; i.e. at the same time rather than one or other as a trade-off. Known in full as M Sharp, this general-purpose programming language is being positioned as safe, productive, and performant. Playing down the development in general, Microsoft architect and developer Joe Duffy says that M# is officially a Microsoft research project, but that he "welcomes technical dialogue" right now.
http://www.drdobbs.com/240165199

Nimrod: A New Systems Programming Language

A language with extensive metaprogramming support, generics and exception tracking built in, optional garbage collection, and rivals C in performance. And it can compile to C, C++, Objective-C, and JavaScript.

By Andreas Rumpf

Nimrod is a statically typed, imperative programming language that tries to give the programmer ultimate power without compromising runtime efficiency. This means it focuses on compile-time mechanisms in all their various forms.

It uses a syntax that is a reminiscent of both Python and Pascal, has an AST-based clean macro system that is ideal for metaprogramming. It supports soft realtime GC on thread-local heaps and uses asynchronous message passing between threads, so no "stop the world" mechanism is necessary. An unsafe shared memory heap is also provided for the increased efficiency that results from that model.

It compiles to commented C code, which attains consistently outstanding results on benchmarks. It can also be made to output JavaScript. The entire Nimrod toolchain (compiler, library, build tool, and so on) is written in Nimrod.

This article gives a quick overview of Nimrod's many features. After an introduction to Nimrod's syntax, I show how the language allows for common procedural, functional, OO, and metaprogramming techniques while remaining simple and efficient.

Introduction to Nimrod's Syntax

Nimrod uses a conventional infix-based syntax. Like Python or Haskell, it uses indentation rather than braces to group statements. The usual control flow statements such as `if`, `case`, `while`, and `for` are provided.

Slightly unusual are the many ways in which function applications can be written: There is the traditional prefix notation `f(x, y)`. If the

call is a statement, the parentheses may be omitted: `f x, y`. This is called the *command notation* and this version of "hello world" makes use of it:

```
echo "hello world!"
```

`echo "abc"` is an alias for `write stdout, "abc"` with the notable difference that it abstracts away the output stream, which makes it easier to emulate for the JavaScript target or to make the compiler evaluate it at compile time.

"Nimrod features the concept of a *routine abstraction*. A routine in Nimrod can be a procedure, method, template, macro, iterator, or a converter."

Another notation for invoking functions is the so-called *method invocation* syntax: `x.f(y, z)`. If there is only one argument, the `()` can be left out: `x.f`. So you can write `x.len` instead of `x.len()` or `len(x)`. This way, there is no need for special getters or read-only properties.

The language clearly distinguishes between `f` and `f()` because functions are first-class citizens and can be passed around like in functional programming languages.

Finally, there are "generalized string literals" that introduce yet another piece of syntactical sugar: Instead of `f("abc")`, you can write `f"abc"` and then backslashed escape sequences like `\n` are not interpreted. This is designed for easy embedding of mini-languages like regular expressions: `re"\w+"` is much easier to write and read than `re("\\w+")`.

Functions are called *procedures* in Nimrod and are declared with the `proc` keyword. Unlike in C and C++, parameters are read-only unless they are declared as `var`, in which case "pass by reference" is used (pass by reference is implemented with a hidden pointer).

Similar to Haskell, operators in Nimrod are simply sugar for functions. The following example declares a procedure named `++` that can take 1, 2, or 3 arguments. The value of `y` defaults to 1, and `z` defaults to 0. `++` modifies `x` and adds `y` and `z` to it.

```
proc `++`(x: var int; y: int = 1; z: int = 0) =
  x = x + y + z

var g = 70
# ++ can then be used like this:
++g
g ++ 7
g.`++`(10, 20)
```

Nimrod features the concept of a *routine abstraction*. A routine in Nimrod can be a procedure, method, template, macro, iterator, or a converter. All routines are invoked with the same syntax; thus, you cannot tell from the invocation which kind of routine it is. Similar to Lisp, Nimrod consciously decouples the syntax from the semantics to allow for powerful metaprogramming:

```
template `!=`(x, y: expr): expr = not (x == y)

# invocation is as if '!=' were a proc:
echo 34 != 33
```

A template is a simple form of a macro; the example shows how the unequals operator is defined in Nimrod. For metaprogramming, the

type system is weakened and very general types like `expr` (expression), `stmt` (statement), or `typedesc` (type descriptor) are available. Note how the template is invoked like an operator.

Functional Programming with Nimrod

As I mentioned earlier, parameters that are not `var` are read-only, so Nimrod has a notion of immutability. Immutability is not deep, however: As soon as any kind of pointer is involved, the location that the pointer points to can be modified:

```
proc modify(n: ref Node) =
  n.value = 45
```

There are two kinds of pointers in Nimrod: `ref` and `ptr`. A `ref` is a pointer that is considered by the garbage collector (traced), while a `ptr` is not (untraced). In general, `ptr` is used for interfacing with C/C++ or to implement weak references or simply for manual memory management. Nimrod is, after all, a systems programming language.

Similar to parameters are `let` variables. A `let` can be assigned only once. Of course, Nimrod also has variables, which use the `var` keyword:

```
let lv = stdin.readline
var vv = stdin.readline
vv = "abc" # valid, reassignment allowed
lv = "abc" # fails at compile time
```

`let` has been designed to emulate parameter passing semantics so that `proc square(x: int): int = x*x` can be emulated with:

```
template square(x: int): int =
  # ensure 'x' is only evaluated once:
  let y = x
  y * y
```

Finally, there is also `const`, which declares true constants. Constants can't be assigned at all, not even once. Instead, their value has to be known at compile time. Nimrod has a sophisticated compile-time evaluation engine, so the following works:

```
proc mostSignificantBit(n: int): int =
  # naive algorithm:
  var m = n
  while m != 0:
    m = m shr 1  # 'shr' means "shift right"
    result += 1
  result -= 1

const msb3999 = mostSignificantBit(3999)
```

`Procs` that return a value have an implicitly declared result variable that represents the return value, so there is no need to write `return result`. `result` is Nimrod's way to guarantee what is called *return value optimization* in C++.

Most variables are initialized implicitly in Nimrod and the initial value is a binary 0. Hence, `result` starts with 0, which is the natural start for counting. `shr` is Nimrod's shift-right operator, a keyword has been chosen to avoid confusion, as >> has the same precedence as `>`. The reason for this is that Nimrod supports user-defined operators and thus needs a simple rule of how operator precedence should be handled. The (simplified) rule is that the first character of the operator determines the precedence.

Now let's get back to functional programming. Since `procs` are first-class citizens, defining `map` and `filter` is straightforward:

```
proc filter[T](a: openarray[T]; predicate:
      proc (x: T): bool): seq[T] =
  result = @[] # @[] constructs the empty seq
  for x in a:
    if predicate(x): result.add(x)
```

```
proc map[T, S](a: openarray[T]; fn: proc (x: T): S):
      seq[S] =
  newSeq(result, a.len)
  for i in 0 .. >a.len: result[i] = fn(a[i])
```

openarray is a special type that is only valid for parameters, it is compatible with arrays and sequences. A sequence (seq) is a growable array. openarray is implemented as a pointer to the first element and a length. Lists instead of arrays are, of course, possible, too, but relatively uncommon. (In this, Nimrod shows its imperative roots.)

Pattern Matching and Metaprogramming

Nimrod supports `product` and `sum` types with some twists: A `sum` type (also called an *algebraic data type*) is supported by Nimrod with a classical enum plus a so-called *object variant*. Let's say we want to create a library for working with mathematical expressions such as `x^2 + 5*x`. It's natural to define our data types like this:

```
type
  FormulaKind = enum
    fkVar,        ## element is a variable like 'X'
    fkLit,        ## element is a literal like 0.1
    fkAdd,        ## element is an addition operation
    fkMul,        ## element is a multiplication operation
    fkExp         ## element is an exponentiation operation

type
  Formula = ref object
    case kind: FormulaKind
    of fkVar: name: string
    of fkLit: value: float
    of fkAdd, fkMul, fkExp: left, right: Formula
```

Nimrod's enum is an old-school typesafe enum, as in Ada, without any fields. To avoid name clashes, it's common to prefix the enum values with a two-letter abbreviation. The case part in the object declaration introduces a checked union. So the access of `f.name` will raise an exception `if f.kind != fkVar`. Everything in Nimrod, including `object`, is a value type, but I prefer reference semantics here for easier manipulation of formulas.

The obvious thing to do with a formula is to evaluate it. The following `proc` does just that. It requires a mapping `varToVal` from the variable name to its value:

```
from math import pow

proc evaluate(n: Formula; varToVal: proc (name: string):
float): float =
  case n.kind
  of fkVar: varToVal(n.name)
  of fkLit: n.value
  of fkAdd: evaluate(n.left, varToVal) +
    evaluate(n.right, varToVal)
  of fkMul: evaluate(n.left, varToVal) *
    evaluate(n.right, varToVal)
  of fkExp: pow(evaluate(n.left, varToVal),
    evaluate(n.right, varToVal))
```

Now, to check whether a formula is a polynomial (to see if we can easily differentiate it, for instance), we can use the following code:

```
proc isPolyTerm(n: Formula): bool =
  n.kind == fkMul and n.left.kind == fkLit and
    (let e = n.right;
    e.kind == fkExp and e.left.kind ==
  fkVar and e.right.kind == fkLit)
```

```
proc isPolynomial(n: Formula): bool =
  isPolyTerm(n) or
    (n.kind == fkAdd and isPolynomial(n.left) and
      isPolynomial(n.right))
```

`isPolyTerm` is quite ugly. Pattern matching would be much nicer. While Nimrod does not support elaborate pattern matching beyond case out-of-the-box, it's quite easy to implement it thanks to the sophisticated macro system: For pattern matching, we define a macro `=~` that constructs the and expression at compile time. Then the code can look like this:

"The point of Nimrod's macros is that they enable DSLs to make use of Nimrod's lovely infix syntax."

```
proc isPolyTerm(n: Formula): bool = n =~ fkMul(fkLit,
  fkExp(fkVar, fkLit))
```

But this is still not as nice as it could be: The point of Nimrod's macros is that they enable DSLs that make use of Nimrod's lovely infix syntax. In fact, that's a conscious design decision: Macros do not affect Nimrod's syntax — they can only affect the semantics. This helps readability.

So here is what we really want to support:

```
proc isPolyTerm(n: Formula): bool = n =~ c * x^c
```

Where `c` matches any literal , `x` matches any variable, and the operators their corresponding formula kinds:

```
proc pat2kind(pattern: string): FormulaKind =
  case pattern
  of "^": fkExp
  of "*": fkMul
  of "+": fkAdd
  of "x": fkVar
  of "c": fkLit
  else:   fkVar # no error reporting for reasons of simplicity
```

Note that for reasons of simplicity, we don't implement any kind of variable binding, so `1 * x^2` matches `c * x^c` as `c` is not bound to the literal `1` in any way. This form of variable binding is called *unification*. Without unification, the pattern matching support is still quite primitive. However, unification requires a notion of equality, and since many useful but different equality relations exist, pattern matching is not baked into the language.

So here's the implementation of the `=~` macro in all its glory:

```
import macros

proc matchAgainst(n, pattern: PNimrodNode):
    PNimrodNode {.compileTime.} =
  template `@`(current, field: expr): expr =
   newDotExpr(current, newIdentNode(astToStr(field)))

 template `==@`(n, pattern: expr): expr =
   newCall("==", n@kind, newIdentNode
    ($pat2kind($pattern.ident)))

 case pattern.kind
 of CallNodes:
   result = newCall("and",
     n ==@ pattern[0],
     matchAgainst(n@left, pattern[1]))
   if pattern.len == 3:
     result = newCall("and", result.copy,
       matchAgainst(n@right, pattern[2]))
```

```
 of nnkIdent:
   result = n ==@ pattern
 of nnkPar:
   result = matchAgainst(n, pattern[0])
 else:
   error "invalid pattern"

macro `=~` (n: Formula; pattern: expr): bool =
 result = matchAgainst(n, pattern)
```

In Nimrod, a template is a declarative form of a macro, while a macro is imperative. It constructs the AST with the help of an API that can be found in the macros module, so that's what line 1 imports. The final macro definition is in line 25 and it follows a fairly common approach: It delegates all of its work to a helper `proc` called `matchAgainst`, which constructs the resulting AST recursively. `PNimrodNode` is the type the Nimrod AST consists of. The Nimrod AST is structured quite similar to how we implemented `Formula`, except that every node can have a variable number of children. `n[i]` is the ith child.

The various function application syntaxes (prefix, infix, command) all map to the same AST structure `kind(callee, arg1, arg2, ...)`, where `kind` describes the particular syntax. In `matchAgainst`, we treat every call syntax the same with the help of `macros.CallNodes`. We allow for `a(b)` and `a(b, c)` (line 15) call syntaxes and construct the AST representing an and expression with the help of two `@` and `==@` templates.

`n@field` constructs the AST that corresponds to `n.field` and `a ==@ b` constructs `a.kind == pat2kind(b)`. Line 18 deals with the case when the pattern only consists of a single identifier (`nnkIdent`), and line 20 supports `()` (`nnkPar`) so that grouping in a pattern is allowed.

As this example shows, metaprogramming is a good way to transform two lines of long ugly code to a short beautiful one-liner at the cost of 30 lines of ugly code dealing with AST transformations. However, the DSL we created here pays off as soon as there are more patterns to match against. It's also reasonably easy to abstract the `=~` pattern-match operator so that it operates on more than just the Formula data type. In fact, a library solution that also supports unification is in development.

Conclusion

Nimrod (http://nimrod-lang.org/index.html) is open source software that runs on Windows, Linux, Mac OS, and BSD. In addition to generating C and JavaScript, it can generate C++ or Objective-C. The compiler can optionally enforce all kinds of error checking (bounds checking, overflow, etc.) and it can perform extensive optimizations.

It has an extensive standard library and many ported libraries. In addition, it has wrappers for most of the premier C libraries (including OLE, X, Glib/GTK, SQLite, etc.) and C-based languages (Lua and Tcl). If you're searching for a systems programming language that provides higher-level constructs and extensive metaprogramming, but boils down to C, Nimrod might well be what you're looking for.

— Andreas Rumpf is the creator of Nimrod.

Comment

Previous Next
Table Of Contents

Unit Testing with Python

Python has substantial resources to enable unit testing

By José R.C. Cruz

Unit testing is considered an essential part of software development. Through unit testing, we can evaluate each code component, find out how well it performs, and determine how well it reacts to valid or invalid input. A regression suite of unit tests is also an excellent way of detecting unexpected changes in a code base caused by refactoring or writing new code.

In this article, I examine the mechanisms of unit testing in Python, starting with the unittest module and its key classes. I examine tests individually and in suites, and I discuss how to facilitate their construction and use. Readers should have a working knowledge of Python. The sample test code requires Python 2.5 or later.

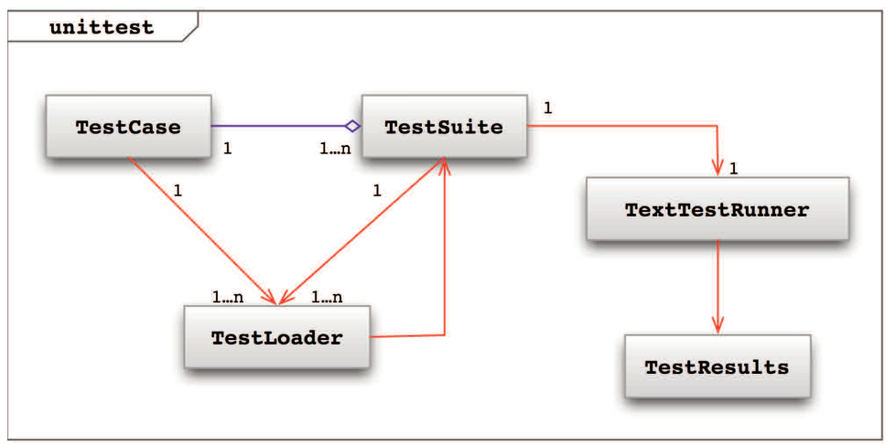


Figure 1: Core classes in unittest.

The unittest Module

The unittest module started life as the third-party module PyUnit. PyUnit was a Python port of JUnit, the Java unit testing framework. Designed by Steve Purcell, PyUnit became an official Python module starting with version 2.5.

As Figure 1 shows, there are five key classes in the unittest module. The `TestCase` class holds the test routines and provides hooks for preparing each routine and for cleaning up after. The `TestSuite` class serves as a collection container. It can hold multiple `TestCase` objects and multiple `TestSuite` objects.

The `TestLoader` class loads test cases and suites defined locally or from an external file. It emits a `TestSuite` object that holds those cases and suites. The `TextTestRunner` class provides a standard platform to run the tests. The `TestResults` class provides a standard container for the test results.

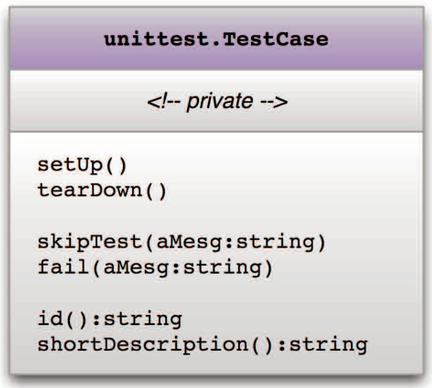


Figure 2: The structure of a TestCase class.

Out of these five classes, only `TestCase` must be subclassed. The other four classes can also be subclassed, but they are generally used as is.

Preparing a Test Case

Figure 2 shows the structure of the `TestCase` class. In it are three sets of methods that are used most often in designing the tests. In the first set are the pre- and post-test hooks. The `setUp()` method fires before each test routine, the `tearDown()` after the routine. Override these methods when you create a custom test case.

The second pair of methods control test execution. Both methods take a message string as input, and both abort an ongoing test. But the `skipTest()` method cancels the current test, while the `fail()` method fails it explicitly.

The third set of methods help identify the test. The method `id()` returns a string containing the name of the `TestCase` object and of the test routine. And the method `shortDescription()` returns the `docstr` comment at the start of each test routine. If the routine has no such comment, `shortDescription()` returns a `None`.

Listing One shows the sample bare bones test case `FooTest`. `FooTest` has two test routines: `testA()` and `testB()`. Both routines get the required argument of self. Both have a `docstr` comment for a first line.

Listing One: Code to show the sequence of unit test execution.

```
#!/usr/bin/python

import unittest

class FooTest(unittest.TestCase):
    """Sample test case"""

    # preparing to test
    def setUp(self):
        """ Setting up for the test """
        print "FooTest:setUp_:begin"
        ## do something...
        print "FooTest:setUp_:end"

    # ending the test
    def tearDown(self):
        """Cleaning up after the test"""
        print "FooTest:tearDown_:begin"
        ## do something...
        print "FooTest:tearDown_:end"

    # test routine A
    def testA(self):
        """Test routine A"""
        print "FooTest:testA"

    # test routine B
    def testB(self):
        """Test routine B"""
        print "FooTest:testB"
```

Figure 3 shows how `FooTest` behaves when executed. Note the same `setUp()` and `tearDown()` methods run before and after each

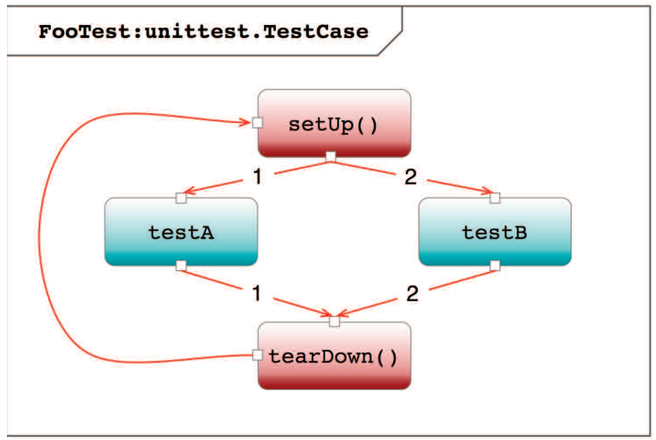


Figure 3: FooTest behavior.

test routine. So how do you let `setUp()` and `tearDown()` know which routine is being run? You must first identify the routine by calling `shortDescription()` or `id()` (See Listing Two). Then use an `if–else` block to route to the appropriate code. In the sample snippet, FooTest calls `shortDescription()` to get the routine's docstr comment, then runs the prep and clean-up code for that routine.

Listing Two: Using test descriptions.

```
import unittest

class FooTest(unittest.TestCase):
    """Sample test case"""

    # preparing to test
    def setUp(self):
        """ Setting up for the test """
        print "FooTest:setUp_:begin"
        testName = self.shortDescription()
        if (testName == "Test routine A"):
            print "setting up for test A"

        elif (testName == "Test routine B"):
            print "setting up for test B"

        else:
            print "UNKNOWN TEST ROUTINE"

        print "FooTest:setUp_:end"

    # ending the test
    def tearDown(self):
        """Cleaning up after the test"""
        print "FooTest:tearDown_:begin"

        testName = self.shortDescription()
        if (testName == "Test routine A"):
            print "cleaning up after test A"

        elif (testName == "Test routine B"):
            print "cleaning up after test B"

        else:
            print "UNKNOWN TEST ROUTINE"

        print "FooTest:tearDown_:end"

    # see Listing One...
```

Designing a Test Routine

Each test routine must have the prefix "test" in its name. Without that prefix, the routine will not run. To perform a test, the test routine should use an assert method. An assert method gets one or more test arguments and an optional `assert` message. When a test fails, the `assert` halts the routine and sends the error message to `stdout`.

Assert	Complement Assert	Operation
assertTrue(a, M)	assertFalse(a, M)	a = True; a = False
assertEqual(a, b, M)	assertNotEqual(a, b, M)	a = b; a ≠ b
assertIs(a, b, M)	assertIsNot(a, b, M)	a is b; a is not b
assertIsNone(a, M)	assertIsNotNone(a, M)	a = nil; a ≠ nil
AssertIsInstance(a, b, M)	AssertIsNotInstance(a, b, M)	isinstance(a,b); not isinstance(a,b)

Table 1: Basic asserts in unittest.

There are three sets of `assert` methods. In the first set (Table 1) are the basic Boolean `asserts`, which fire on a `True` or `False` result.

To check for just a `True` or `False`, use `assertTrue()` or `assertFalse()`, as in Listing Three:

Listing Three: Checking for True or False.

```
self.assertTrue(argState, "foobar() gave back a False")
# -- fires when the instance method foobar() returns a True

self.assertFalse(argState)
# -- fires when foobar() returns a False
# Notice this one does not supply an assert message
```

To check whether two arguments are the same, use `assertEqual()` and `assertNotEqual()` as in Listing Four. These last two `asserts` check the arguments' values, as well as their data types.

Listing Four: Checking arguments.

```
argFoo = "narf"
argBar = "zort"

self.assertEqual(argFoo, argBar, "These are not the same")
# -- this assert will fail

self.assertNotEqual(argFoo, argBar, "These are the same")
# -- this assert will succeed

argFoo = 123
argBar = "123"

self.assertEqual(argFoo, argBar, "These are not the same")
# -- this assert will fail
```

To check if the arguments are the same objects, use `assertIs()` and `assertIsNot()`. Like `assertEqual()` and `assertNotEqual()`, these two asserts examine both argument values and type. To check if an argument is an instance of a specific class, use `assertIsInstance()` and `assertIsNotInstance()` as in Listing Five.

Listing Five: Checking if an argument is an instance of a specific class.

```
argFoo = Bar()

# checking against class Bar
self.assertIsInstance(argFoo, Bar, "The object is not an instance of class Bar")
# -- this assert will succeed


# checking against class Foo
self.assertIsNotInstance(argFoo, Foo, "The object is an instance of class Foo")
# -- this assert will fail
```

Assert	Complement Assert	Operation
assertGreater(a, b, M)	assertLess(a, b, M)	a > b; a < b
assertGreaterEqual(a, b, M)	assertLessEqual(a, b, M)	a ≥ b; a ≤ b
assertAlmostEqual(a, b, n, M)	assertNotAlmostEqual(a, b, n, M)	round(a-b, n) = 0 round)a-b, n) ≠ 0
assertItemsEqual(a, b, M)	*none*	sort(a) = sort(b); sort(a) ≠ sort(b)

Table 2: The comparative assertions.

Both `asserts` get a class name as a second argument. Both behave similarly to the library function `isInstance()`. Finally, to check for a `nil`, use `assertIsNone()` and `assertIsNotNone()`.

The second set of `asserts` are comparative (see Table 2).To check whether one argument is greater or less than another, use `assertGreater()` and `assertLess()` as in Listing Six.

Listing Six: Greater or Less.

```
argFoo = 123
argBar = 452

self.assertGreater(argFoo, argBar, “Foo is less than Bar”)
# -- this assert will fail

self.assertLess(argFoo, argBar, “Foo is greater than Bar”)
# -- this assert will succeed
```

To check whether one argument is greater, less than or equal to the other, use `assertGreaterEqual()` and `assertLessEqual()`. The arguments in these four `asserts` can be a primitive (`integer`, `float`, `character`), a sequence, or a collection. Both arguments, however, must have the same data type.

To do a tolerance check, use `assertAlmostEqual()` and `assertAlmostNotEqual()`. These two `asserts` round off the arguments to a fixed number of decimal places before comparing their values. The number of decimal places is 7 by default. To change it, pass the new number with a `places` label.

To compare two sequences, use `assertItemsEqual()`. Its two arguments must be the same sequence type (list, tuple, set, and so on). Note that this `assert` sorts the sequence items prior to comparison.

Third set of `asserts` (Table 3) work with collection objects such as dictionaries, lists, sets, and tuples.

Assert	Complement Assert	Operation
assertIn(a, b, M)	assertNotIn(a, b, M)	a in b; a not in b
assertDictContainsSubset(a, b, M)	*none*	a has b
assertDictEqual(a, b, M)	*none*	a = b
assertListEqual(a, b, M)	*none*	a = b
assertSetEqual(a, b, M)	*none*	a = b
assertSequenceEqual(a, b, M)	*none*	a = b
assertTupleEqual(a, b, M)	*none*	a = b
assertMultilineEqual(a, b, M)	*none*	a = b

Table 3: Assertions for collections.

Arguments must be a collection type. Some assertions need not use arguments of the same type. All but one `assert` in this set have no complements.

To check whether one dictionary has some of the key/value pairs as the other, use `assertDictContainsSubset()` as shown in Listing Seven.

Listing Seven.

```
argBar = {'narf':456, 'poink':789}

self.assertDictContainsSubset(argFoo, argBar, "Foo does not
have Bar")
# -- this assert will succeed

self.assertDictContainsSubset(argBar, argFoo, "Foo does not
have Bar")
# -- this assert will fail

argBar = {'narf':456, 'egad':789}
self.assertDictContainsSubset(argFoo, argBar, "Foo does not
have Bar")
# -- this assert will also fail
```

The first argument serves as reference; the second holds the pairs in question. To check if both dictionaries have the same key/value pairs, use `assertDictEqual()`. Each pair must have the same key labels and data values. How the pairs are arranged is irrelevant.

To check two sequence objects, use `assertSequenceEqual()`. Sequence types include lists, sets, tuples, even strings. For sequence objects to be same, they must have the same number of data items. The items must have the same value and they must be arranged the same. The sequence type must also be the same.

To check if two list objects are the same, use `assertListEqual()`. Both objects must have the same number of items. Those items must have the same values and the same order. To check two set objects, use `assertSetEqual()`. As with lists, both set objects must have the same number of items and item values. But item order is irrelevant, because set objects arrange their items internally.

Assert	Complement Assert	Operation
`assertIn(a, b, M)`	`assertNotIn(a, b, M)`	`a in b; a not in b`
`assertDictContainsSubset(a, b, M)`	*none*	`a has b`
`assertDictEqual(a, b, M)`	*none*	`a = b`
`assertListEqual(a, b, M)`	*none*	`a = b`
`assertSetEqual(a, b, M)`	*none*	`a = b`
`assertSequenceEqual(a, b, M)`	*none*	`a = b`
`assertTupleEqual(a, b, M)`	*none*	`a = b`
`assertMultilineEqual(a, b, M)`	*none*	`a = b`

Table 3: Assertions for collections.

Finally, to check if two tuples are the same, use `assertTuplesEqual()`. To check if two strings are the same, use `assertMultilineEqual()`. And to find out if one string is or is not inside another string, use `assertIn()` and `assertNotIn()`.

This third set of `asserts` has one interesting behavior. If the collection objects are not equal, the `assert` will report the differences between the objects. It also adds this `diff` result to the `assert` message, if one is available.

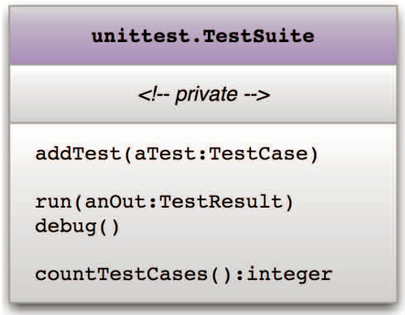


Figure 4: The TestSuite class.

Preparing a Test Suite

Usually, a few test cases are enough for your testing needs. But what if you have a dozen or more tests cases on hand — some you wrote yourself, some written by others? What if you want only a subset of test routines to run from the same test case? What if you want to refactor your test routines for easier cataloging and distribution? For these situations, you might need a test suite.

Figure 4 shows the basic structure of the `TestSuite` class.

There are three sets of instance methods. The first set lets us add test cases to the suite. To add a single test case, use the `addTest()` method as shown in Listing Eight.

Listing Eight.

```
class FooTest(unittest.TestCase):
    def testA(self):
        """Test routine A"""
        print "Running test A"

# creating a new test suite
newSuite = unittest.TestSuite()

# adding a test case
newSuite.addTest(unittest.makeSuite(FooTest))
```

Pass the test case (here being `FooTest`) to the instance method using the convenience function `makeSuite()`. You can also use `makeSuite()` to "convert" the test case into a test suite.

```
newSuite = unittest.makeSuite(FooTest)
```

To add a specific test routine, pass the test case object to the suite through the same `addTest()` method. Then pass the name of the test routine to the test case's constructor. Notice the routine name is passed as a string.

```
newSuite.addTest(FooTest("testA"))
```

You can also use the same method to add two or more test routines to the same suite:

```
newSuite.addTest(FooTest("testA"))
newSuite.addTest(FooTest("testB"))
#...
```

To add two or more test cases, gather the names of the test cases into a list as shown in Listing Nine.

Listing Nine.

```
testList = [FooTest, BarTest]
testLoad = unittest.TestLoader()

caseList = []
for testCase in testList:
    testSuite = testLoad.loadTestsFromTestCase(testCase)
    caseList.append(testSuite)

newSuite = unittest.TestSuite(caseList)
```

Parse the list with a `for` loop, then use a `TestLoader` object (`testLoad`) to read each case. Add the read cases to a second list (`caseList`). Then

create the `TestSuite` object (`newSuite`), passing to the class constructor the list object caseList.

Suppose you want to add another test suite to the suite. Simply pass the other suite to the `addTest()` method — no need to reuse the `makeSuite()` function to prepare the added suite.

```
fooSuite = unittest.TestSuite()
fooSuite.addTest(unittest.makeSuite(FooTest))
#...
barSuite = unittest.TestSuite()
barSuite.addTest(fooSuite)
```

The second set of methods run the tests in the test suite. The `run()` method takes a `TestResult` object as input, while `debug()` does not. But `debug()` does let an external debugger monitor the ongoing test.

Finally, the last set contains the method `countTestCases()`. This method returns the number of test cases held in the suite.

```
testCount = fooSuite.countTestCases()
```

Running the Tests

You have two ways to run your unit tests. If the test script is a single file with one or more test cases, add these lines after the last test case.

```
if __name__ == "__main__":
unittest.main()
```

The `if` block detects how the file is acted upon. If the file is imported into another file, the macro `__name__` is unchanged. If the file is executed directly, either from the text editor, from another script, or from the command-line, the `__name__` macro resolves into "`__main__`", and the class method `main()` gets called. This in turn invokes the `run()` methods of every test case defined or imported by the script file.

If the script file defines a test suite, first create an instance of `TextTestRunner`. Then pass the test suite object to the runner's `run()` method.

```
fooRunner = unittest.TextTestRunner()
fooRunner.run(fooSuite)
```

Regardless of approach, test cases and their routines run in alphanumeric order. `BarTest` runs before `FooTest`, `FooTest` before `Test123`, and `test_12()` before `test_A()`. Consider the test script in Listing Nine. In it, you have two test cases: `BarTest` and `FooTest`. `BarTest` has three test routines; `FooTest` has two.

Figure 5 shows how these two test cases run. `BarTest` runs first, `FooTest` second.

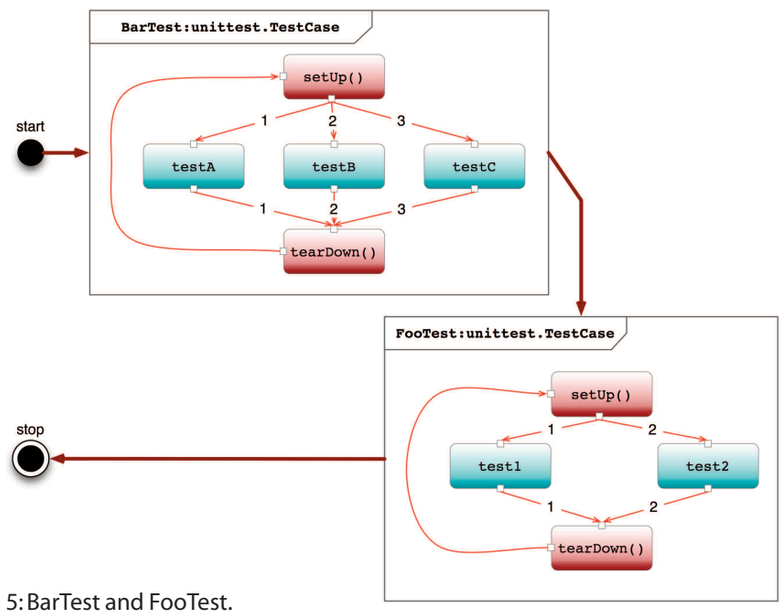


Figure 5: BarTest and FooTest.

The test routines in `BarTest` run in the order from A to C. Those in `FooTest` run from 1 to 2. The same `setUp()` and `tearDown()` methods run before and after each test routine. But the `BarTest` routines have their own `setUp()` and `tearDown()`. The same also holds for the `FooTest` routines.

Finally, you have the option to skip or fail some of the test routines. To skip a routine unconditionally, use the class method `unittest.skip()` as shown in Listing Ten.

Listing Ten: Skipping a routine.

```
@unittest.skip("Skip over the entire test routine")
def testB():
    """Test routine B"""
    # the following code will not run
    fooA = narf()
    self.assertNotEqual(fooA, 123)
```

This method gets one argument: a log message describing the reason for the skip. Place the method call before the test routine, and make sure to prefix the call with an `@` token. Alternatively, use the instance method `skipTest()`, which you can place inside the test routine as in Listing Eleven.

Listing Eleven.

```
"""Test routine B"""
self.skipTest("Skip over the rest of the routine")

# the following code will not run
fooA = narf()
self.assertNotEqual(fooA, 123)
```

To skip a test routine conditionally, use the class methods `unittest.skipIf()` and `unittest.skipUnless()` as in Listing Twelve:

Listing Twelve.

```
@unittest.skipIf(self.fooC > 456, "Skip over this routine")
def testB():
    """Test routine B"""
    # the following code will run only when fooC is less
    # than or equal to 456
    fooA = narf()
    self.assertNotEqual(fooA, 123)

@unittest.skipUnless(self.fooC > 456, "Skip over this routine")
def testC():
    """Test routine C"""
    # the following code will run only when fooC is
    # greater than 456
    fooA = zort()
    self.assertNotEqual(fooA, 123)
```

The first method causes a skip when a condition is met, the second when a condition is not met. Both get two arguments: the condition and the reason for the skip. Both must be placed before the test routine. The condition may involve a class property or another class method.

To fail a test routine, use the instance method `fail()` as in Listing Thirteen:

Listing Thirteen.

```
def testB():
    """Test routine B"""
    self.fail("Force this routine to fail.")

    # the following code will not run
```

```
    fooA = narf()
    self.assertNotEqual(fooA, 123)

def testC():
    """Test routine C"""
    print "This routine still runs after testB."
```

This method takes a log message explaining the reason for failure. Like `skipTest()`, the `fail()` method goes inside the test routine. Code placed after the call to `fail()` will not run, but test routines after the failed routine still get to run.

Viewing the Test Results

There are two possible forms of output from the `TextTestRunner`: console text or a `TestResult` object. First, let's look at the console output, which shows each test's result. You can control this output by passing three optional arguments to the class constructor.

```
unittest.TextTestRunner(stream=sys.stderr,
descriptions=True, verbosity=1)
```

The first argument (labelled `stream`) sets the output destination. This defaults to `sys.stderr` if one is not specified. The next argument (labelled `descriptions`) controls how errors and failures are reported. Passing a `True` (default) tells the runner to name those routines that erred, failed, or skipped. Passing a `False` tells it not to.

The last constructor argument (labelled `verbosity`) sets the level of detail. There are three possible levels. For a `verbosity` of 0, the results show only the number of executed tests and the final outcome of those tests. For a `verbosity` of 1, the results marks a successful test with a dot, a failed one with an `F`, a skipped one with an `s`, and an erroneous one with an `E`. And for a `verbosity` of 2, the results lists each test case and test routine, plus the outcome of each routine.

To demonstrate, let's run the test script in Listing Fourteen:

Listing Fourteen: Test script.

```
import sys
import unittest

class FooTest(unittest2.TestCase):
    """Sample test case"""

    # preparing to test
    def setUp(self):
        """ Setting up for the test """
        print "FooTest:setUp_"

    # ending the test
    def tearDown(self):
        """Cleaning up after the test"""
        print "FooTest:tearDown_"

    # test routine A
    #@unittest2.skip("FooTest:test_A:skipped")
    def test_A(self):
        """Test routine A"""
        self.skipTest("FooTest:test_A:skipped")
        print "FooTest:test_A"

    # test routine B
    def test_B(self):
        """Test routine B"""
        fooA = 123
        fooB = 234
        self.assertEqual(fooA, fooB, "A is not equal to B")
        print "FooTest:test_B"

    # test routine C
    def test_C(self):
        """Test routine C"""
        fooA = 123
        self.assertEqual(fooA, fooB, "A is not equal to B")
        print "FooTest:test_C"
```

```
    # test routine D
    def test_D(self):
        """Test routine D"""
        self.fail("FooTest:test_D:fail_")
        print "FooTest:test_D"

# Run the test case
if __name__ == '__main__':
    fooSuite =
        unittest.TestLoader().loadTestsFromTestCase(FooTest)
```

If you create the test runner as follows:

```
fooRunner = unittest.TextTestRunner(description=True)
fooRunner.run(fooSuite)
```

The results appear as shown next:

```
sFEF
======================================================================
ERROR: test_C (__main__.FooTest)
Test routine C
----------------------------------------------------------------------

======================================================================
FAIL: test_B (__main__.FooTest)
Test routine B
----------------------------------------------------------------------

======================================================================
FAIL: test_D (__main__.FooTest)
Test routine D
----------------------------------------------------------------------

----------------------------------------------------------------------
Ran 4 tests in 0.004s

FAILED (failures=2, errors=1, skipped=1)
```

But if you create the runner as follows:

```
fooRunner = unittest.TextTestRunner(description=False)
```

The `docstr` comment line from each test routine will not be included.

The console output just shown has a `verbosity` level of 1. It starts with the line `sFEF`, which states two failed tests, one skipped, and one erroneous test. But suppose you created the test runner by typing

```
fooRunner = unittest.TextTestRunner(verbosity=0)
```

Then, the `sFEF` line will not appear. If you create the test runner by typing

```
fooRunner = unittest.TextTestRunner(verbosity=2)
```

Then the console output appears as:

```
test_A (__main__.FooTest)
Test routine A ... skipped 'FooTest:test_A:skipped'
test_B (__main__.FooTest)
Test routine B ... FAIL
test_C (__main__.FooTest)
Test routine C ... ERROR
test_D (__main__.FooTest)
Test routine D ... FAIL

======================================================================
ERROR: test_C (__main__.FooTest)
Test routine C
----------------------------------------------------------------------

======================================================================
FAIL: test_B (__main__.FooTest)
Test routine B
----------------------------------------------------------------------
```

```
======================================================================
FAIL: test_D (__main__.FooTest)
Test routine D
----------------------------------------------------------------------

----------------------------------------------------------------------
Ran 4 tests in 0.009s

FAILED (failures=2, errors=1, skipped=1)
```

Note that the `sFEF` line is replaced by four pairs of lines, each pair naming the test case and routine, the final test state, the `docstr` comment of each routine, and any `assert` messages.

The second type of output is the `TestResults` object. This one is returned by the runner object after all the tests have run:

```
fooResult = fooRunner.run(fooSuite)
```

The `TestResult` object has two sets of property accessors (as shown in Figure 6).

This is not a complete set of accessors, but these are ones you will most likely use. The first set returns a list of tuples. Each tuple reveals how each test routine fared when executed. The `errors` accessor lists identified routines that raised an exception. Each tuple has the name of the test case and routine, the location of the test script, the line position of the error, a trace-back, and a reason for the error.

Figure 6: The TestResult object (in part).

The `failures` accessor lists test routines that failed. Its tuples contain the same information as tuples from the `errors` accessor. The `skipped` accessor lists routines that were skipped, conditionally or not. Its tuples name the test case and routine, and give the reason for the skip.

The second set of accessors provides additional data. `testsRun` gives the number of test routines that ran, regardless of outcome. And `wasSuccessful()` returns a `True` if all routines ran without problems, `False` if at least one routine had a problem. Notice this last accessor is written as a function.

Listing Fifteen demonstrates how the `TestResult` object works.

Listing Fifteen.

```
import sys
import unittest

class FooTest(unittest.TestCase):
    """Sample test case"""

# Run the test case
if __name__ == '__main__':
    fooSuite =
        unittest.TestLoader().loadTestsFromTestCase(FooTest)

    fooRunner = unittest.TextTestRunner()
    fooResult = fooRunner.run(fooSuite)

    print
    print "---- START OF TEST RESULTS"
```

```
print fooResult
print
print "fooResult::errors"
print fooResult.errors
print
print "fooResult::failures"
print fooResult.failures
print
print "fooResult::skipped"
print fooResult.skipped
print
print "fooResult::successful"
print fooResult.wasSuccessful()
print
print "fooResult::test-run"
print fooResult.testsRun
print "---- END OF TEST RESULTS"
print
```

This script uses the same `FooTest` case defined in Listing Fourteen. After it invokes the `run()` method in the runner object `fooRunner`, the script stores the results into the local `fooResults`. Then it invokes each accessor and prints the test result on the console window.

Here are the test results returned by `fooRunner`:

```
---- START: Test Results:
<unittest2.runner.TextTestResult run=4 errors=1 failures=2>

fooResult::errors
[(<__main__.FooTest testMethod=test_C>, 'Traceback (most
recent        call        last):\n                File
"/Volumes/Projects/Pro_Articles/_ddj/
18_pyUnitTest/ddj18_code/foo_testRun.py", line 49, in
test_C\n
self.assertEqual(fooA, fooB, "A is not equal to B")\nNameEr-
ror:
global name \'fooB\' is not defined\n')]

fooResult::failures
[(<__main__.FooTest testMethod=test_B>, 'Traceback (most
recent        call        last):\n                File
"/Volumes/Projects/Pro_Articles/_ddj/
18_pyUnitTest/ddj18_code/foo_testRun.py", line 41, in
test_B\n
self.assertEqual(fooA, fooB, "A is not equal to B")
\nAssertionError: 123 != 234 : A is not equal to B\n'),
(<__main__.FooTest testMethod=test_D>, 'Traceback (most
recent        call        last):\n                File
"/Volumes/Projects/Pro_Articles/_ddj/
18_pyUnitTest/ddj18_code/foo_testRun.py", line 55, in
test_D\n
self.fail("FooTest:test_D:fail_")\nAssertionError:
FooTest:test_D:fail_\n')]

fooResult::skipped
[(<__main__.FooTest testMethod=test_A>,
'FooTest:test_A:skipped')]

fooResult::successful?
False

fooResult::test-run
4
```

The second line summarizes the results. It shows the total number of test routines and how many routines have erred or failed. The paragraph below it reveals the erroneous routine: `test_C()`. It also reveals the cause of the error: the undefined variable `fooB`.

The next clump of text reveals the failed routines: `test_B()` and `test_D()`. It reveals why `test_B()` failed: two unequal values passed to `assertEqual()`. And it reveals that `test_D()` has explicitly called the instance method `fail()`.

Below that is shown the skipped routine, `test_A()`. The next line shows what problems were encountered. And the last two lines report that a total of four test routines ran, confirming what was reported initially.

Conclusion

Python has substantial resources to enable unit testing. In this article, I looked into the unittest module and examined those classes essential for unit testing. I showed how to create a test case, how to design the test routines, and how to gather several test cases into one test suite.

The unittest module can do more than just run basic tests. With it, we can design tests that focus on exceptions or tests that do pattern matching. We can load and execute tests stored in separate files. We can even design tests that use simulated objects (known as mocks). But these are topics for a later article. Meanwhile, the "Recommended References" can provide additional guidance on these more-advanced features.

Recommended References

Test-Driven Development in Python (http://is.gd/NsN4XG), Jason Diamond. ONLamp.com/O'Reilly Publishing.

Introduction to unittest (http://is.gd/qC3fkw), Michael Foord. Voidspace.

unittest: Automated testing framework (http://is.gd/EXqfw6), Doug Hellman. PyMOTW.

unittest: Unit testing framework (http://is.gd/qiKLMH), Python 2.7.6 documentation.

Testing Your Code (http://is.gd/BmzXWD), Kenneth Reitz. T*he Hitchhiker's Guide to Python*.

— José R. C. Cruz is a freelance technical writer based in British Columbia. He frequently contributes articles to Dr. Dobb's, MacTech, and REALStudio Developer.

Comment

From the Vault

C++ Compilation Speed

There are many reasons why C++ compiles slowly

By Walter Bright

I often hear a complaint that C++ code tends to be slow to compile, sometimes even taking overnight. Slow compiles were one of the motivations for exported templates, and are even listed as one of the reasons for the development of the Go language (http://golang.org/doc/faq). It's a real problem, and since I'm in the C++ compiler business, I get asked about this. Why is C++ compilation slow? As it's reasonable to assume that C++ compiler implementers are pretty good at writing fast code, there must be something inherent in the language. C++ compilers do vary widely in their compilation speeds (http://is.gd/TUsy7M). But that isn't the whole story, since other languages routinely compile orders of magnitude faster, and it can't be true that the good compiler guys only implement other languages (!).

I've been working on C++ compilers since 1987. Back then, machines were extremely slow relative to today, and I paid enormous attention to trying to make the compiler fast. I've spent a lot of time doing performance profiling and tweaking the guts of the compiler to make it fast, and found what aspects of the language slow things down.

The reasons are:

1. The 7 phases of translation [1]. Although some of these can be combined, there are still at least 3 passes over the source text. At least I never was able to figure out how to reduce it below 3. A fast language design would have just one. C++0x exacerbates this by requiring that trigraph and \ line splicing be unwindable to support raw string literals [2].

2. Each phase is completely dependent on the previous one, meaning that there's no reliable way to look ahead and, for example, look for #includes and fire off an asynchronous read in advance for them. The compiler cannot look ahead to see if there's a raw string literal and so not do trigraph translation, it must do the trigraphs, and keep some sort of undo list. I've never figured out a way to parallelize C++ compilation other than at the gross level that make provides with the `-j` switch.

"The meaning of every semantic and syntactic (not just lexical) construct depends on the totality of the source text that precedes it."

3. Because `#includes` are a textual insertion, rather than a symbolic one, the compiler is doomed to uselessly reprocess them when one file is `#included` multiple times, even if it is protected by `#ifndef` pairs. (Kenneth Boyd tells me that upon careful reading the Standard may allow a compiler to skip reprocessing `#includes` protected by `#ifndef` pairs. I don't know which compilers, if any, take advantage of this.)
4. There's a tendency for source files to just `#include` everything, and when it's all accounted for by the compiler, there's often a truly epic amount of source text that has to be processed for every .cpp file. Just `#include`ing the Standard `<iostream>` results, on Ubuntu, in 74 files being read of 37,687 lines (not including any lines from multiple `#includes` of the same file). Templates and the rise of generic programming has exacerbated this, and there's increasing pressure to put more and more of the code of a program into header files, making this problem even worse.
5. The meaning of every semantic and syntactic (not just lexical) construct depends on the totality of the source text that precedes it. Nothing is context independent. There's no way to correctly preparse, or even lex, a file without looking at the `#include` file contents. Headers can mean different things the second time they are `#included` (and in fact, there are headers that take advantage of this).
6. Because of reason 5, the compiler cannot share results from compiling a `#include` from one Translation Unit (TU) [3] to the next. It must start all over again from scratch for each TU.
7. Because different TUs don't know about each other, commonly used templates get instantiated all over again for each TU. The linker removes the duplicates, but there's a lot of wasted effort generating those instances.

Precompiled headers address some of these issues by making certain simplifying assumptions about C++ that are non-Standard, such as a header will mean the same thing if `#incuded` twice, and you have to be careful not to violate them.

Trying to fix these issues while maintaining legacy compatibility would be challenging. I expect there to be some significant effort to solve this problem in the C++ standard following C++0x, but that's at least 10 years out.

In the meantime, there isn't much of a solution. Exported templates were deprecated, precompiled headers are non-Standard, imports were dropped from C++0x, often you don't have a choice about which

compiler to use, etc. Effective use of the `-j` switch to make is the best solution out there at the moment.

Notes

[1] Paraphrased from C++98 2.1, the seven phases are:

1. Trigraph and Universal character name conversion.
2. Backslash line splicing.
3. Conversion to preprocessing tokens. The Standard notes this is context dependent.
4. Preprocessing directives executed, macros expanded, #includes read and run through phases 1..4.
5. Conversion of source characters inside char and string literals to the execution character set.
6. String literal concatenation.
7. Conversion of preprocessing tokens to C++ tokens.

[2] The example in the C++0x Standard is at 2.14.5-4:

```
const char *p = R"(a\
b
c)";
assert(std::strcmp(p, "a\\\nb\nc") == 0);
```

[3] A TU, or Translation Unit, is typically one C++ source file that usually has a .cpp filename extension. Compiling one TU results in one object file. The compilation process compiles each TU independently of any other TU's, and then the linker combines the object file output of those compilations into a single executable file.

Acknowledgements

Thanks to Andrei Alexandrescu, Jason House, Brad Roberts and Eric Niebler for their helpful comments on a draft of this.

— Walter Bright is a computer programmer known for being the designer of the D programming language. He was also the main developer of the first native C++ compiler, Zortech C++ (later to become Symantec C++, now Digital Mars C++).

Comment

This Month on DrDobbs.com

Items of special interest posted on www.drdobbs.com over the past month that you may have missed

TESTING IS NOT VERIFICATION AND VICE VERSA

Internal validity checking might be able to reveal bugs that neither black- nor white-box testing would be likely to encounter.
http://www.drdobbs.com/240165079

JAVA COMMUNICATIONS WITHOUT JNI

PureJavaComm's goal is to provide Java-only access to the comms port requiring only the Java Native Access (JNA) library. JNA is an open source project maintained by Tim Wall that allows you to write Java code only to call into native libraries (i.e., DLLs on Windows). Without JNA, you would have to write potentially complex Java Native Interface (JNI) code or other glue code to make this work.
http://www.drdobbs.com/240164940

THREE-WAY MERGING: A LOOK UNDER THE HOOD

Automating three-way code merges requires considerable sophistication from the version control system.
http://www.drdobbs.com/240164902

JSON AND THE MICROSOFT C++ REST SDK

The C++ standard library provides useful synchronous streams.The Microsoft C++ REST SDK provides asynchronous streams that resemble those included in the C++ standard template library (STL). Let's use the SDK to retrieve and send JSON data.
http://www.drdobbs.com/240164821

TO BE CONTINUED: LOCAL CONTINUATIONS WITH THE PROTOTHREADS LIBRARY

You might wonder why Al Williams has been thinking about operating systems on Arduino lately. Part of it is because he recently acquired what can only be described as a "huge" Arduino board: the Intel Galileo. This behemoth is really an embedded Linux board. It has an x86 processor, Ethernet, a flash memory slot, and USB. It even has a mini PCIe slot!
http://www.drdobbs.com/240162981

THE BEST OF 2013

The most popular articles of the past 12 months.
http://www.drdobbs.com/240164956

Dr.Dobb's

Andrew Binstock Editor in Chief, Dr. Dobb's
andrew.binstock@ubm.com

Deirdre Blake Managing Editor, Dr. Dobb's
deirdre.blake@ubm.com

Amy Stephens Copyeditor, Dr. Dobb's
amy.stephens@ubm.com

Jon Erickson Editor in Chief Emeritus, Dr. Dobb's

DR. DOBB'S EDITORIAL
751 Laurel Street #614
San Carlos, CA
94070
USA

UBM TECH
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000