Coconut

Release v1.3.0 [Dead Parrot]

Sep 04, 2017

Contents

CHAPTER 1

Coconut FAQ

1.1 Frequently Asked Questions

1.1.1 Can I use Python modules from Coconut and Coconut modules from Python?

Yes and yes! Coconut compiles to Python, so Coconut modules are accessible from Python and Python modules are accessible from Coconut, including the entire Python standard library.

1.1.2 What versions of Python does Coconut support?

Coconut supports any Python version `>= 2.6` on the `2.x` branch or `>= 3.2` on the `3.x` branch. In fact, Coconut code is compiled to run the same on every one of those supported versions! See compatible Python versions for more information.

1.1.3 Can Coconut be used to convert Python from one version to another?

Yes! But only in the backporting direction: Coconut can convert Python 3 to Python 2, but not the other way around. Coconut really can, though, turn Python 3 code into version-independent Python. Coconut will compile Python 3 syntax, built-ins, and even imports to code that will work on any supported Python version (`2.6`, `2.7`, `>=3.2`).

There a couple of caveats to this, however: some constructs, like `async`, are for all intents and purposes impossible to recreate in lower Python versions, and require a particular `--target` to make them work. For a full list, see compatible Python versions.

1.1.4 How do I release a Coconut package on PyPI?

Since Coconut just compiles to Python, releasing a Coconut package on PyPI is exactly the same as releasing a Python package, with an extra compilation step. Just write your package in Coconut, run `coconut` on the source code, and upload the compiled code to PyPI. You can even mix Python and Coconut code, since the compiler will only touch `.coco` files. If you want to see an example of a PyPI package written in Coconut, including a Makefile with the exact compiler commands being used, check out pyprover.

1.1.5 I saw that Coconut was recently updated. Where is the change log?

Information on every Coconut release is chronicled on the GitHub releases page. There you can find all of the new features and breaking changes introduced in each release.

1.1.6 Does Coconut support static type checking?

Yes! Coconut compiles the newest, fanciest type annotation syntax into version-independent type comments which can then by checked using Coconut's built-in MyPy Integration.

1.1.7 Help! I tried to write a recursive iterator and my Python segfaulted!

No problem—just use Coconut's `recursive_iterator` decorator and you should be fine. This is a known Python issue but `recursive_iterator` will fix it for you.

1.1.8 How do I split an expression across multiple lines in Coconut?

Since Coconut syntax is a superset of Python 3 syntax, Coconut supports the same line continuation syntax as Python. That means both backslash line continuation and implied line continuation inside of parentheses, brackets, or braces will all work. Parenthetical continuation is the recommended method, and Coconut even supports an enhanced version of it.

1.1.9 If I'm already perfectly happy with Python, why should I learn Coconut?

You're exactly the person Coconut was built for! Coconut lets you keep doing the thing you do well—write Python—without having to worry about annoyances like version compatibility, while also allowing you to do new cool things you might never have thought were possible before like pattern-matching and lazy evaluation. If you've ever used a functional programming language before, you'll know that functional code is often much simpler, cleaner, and more readable (but not always, which is why Coconut isn't purely functional). Python is a wonderful imperative language, but when it comes to modern functional programming—which, in Python's defense, it wasn't designed for—Python falls short, and Coconut corrects that shortfall.

1.1.10 How will I be able to debug my Python if I'm not the one writing it?

Ease of debugging has long been a problem for all compiled languages, including languages like `C` and `C++` that these days we think of as very low-level languages. The solution to this problem has always been the same: line number maps. If you know what line in the compiled code corresponds to what line in the source code, you can easily debug just from the source code, without ever needing to deal with the compiled code at all. In Coconut, this can easily be accomplished by passing the `--line-numbers` or `-l` flag, which will add a comment to every line in the compiled code with the number of the corresponding line in the source code. Alternatively, `--keep-lines` or `-k` will put in the verbatim source line instead of or in addition to the source line number. Then, if Python raises an error, you'll be able to see from the snippet of the compiled code that it shows you a comment telling you what line in your source code you need to look at to debug the error.

1.1.11 I don't like functional programming, should I still learn Coconut?

Definitely! While Coconut is great for functional programming, it also has a bunch of other awesome features as well, including the ability to compile Python 3 code into universal Python code that will run the same on *any version*. And that's not even mentioning all of the features like pattern-matching and destructuring assignment with utility extending far beyond just functional programming. That being said, I'd highly recommend you give functional programming a shot, and since Coconut isn't purely functional, it's a great introduction to the functional style.

1.1.12 I don't know functional programming, should I still learn Coconut?

Yes, absolutely! Coconut's tutorial assumes absolutely no prior knowledge of functional programming, only Python. Because Coconut is not a purely functional programming language, and all valid Python is valid Coconut, Coconut is a great introduction to functional programming. If you learn Coconut, you'll be able to try out a new functional style of programming without having to abandon all the Python you already know and love.

1.1.13 I don't know Python very well, should I still learn Coconut?

Maybe. If you know the very basics of Python, and are also very familiar with functional programming, then definitely—Coconut will let you continue to use all your favorite tools of functional programming while you make your way through learning Python. If you're not very familiar either with Python, or with functional programming, then you may be better making your way through a Python tutorial before you try learning Coconut. That being said, using

Coconut to compile your pure Python code might still be very helpful for you, since it will alleviate having to worry about version incompatibility.

1.1.14 Why isn't Coconut purely functional?

The short answer is that Python isn't purely functional, and all valid Python is valid Coconut. The long answer is that Coconut isn't purely functional for the same reason Python was never purely imperative—different problems demand different approaches. Coconut is built to be *useful*, and that means not imposing constraints about what style the programmer is allowed to use. That being said, Coconut is built specifically to work nicely when programming in a functional style, which means if you want to write all your code purely functionally, Coconut will make it a smooth experience, and allow you to have good-looking code to show for it.

1.1.15 Won't a transpiled language like Coconut be bad for the Python community?

I certainly hope not! Unlike most transpiled languages, all valid Python is valid Coconut. Coconut's goal isn't to replace Python, but to *extend* it. If a newbie learns Coconut, it won't mean they have a harder time learning Python, it'll mean they *already know* Python. And not just any Python, the newest and greatest—Python 3. And of course, Coconut is perfectly interoperable with Python, and uses all the same libraries—thus, Coconut can't split the Python community, because the Coconut community *is* the Python community.

1.1.16 I want to use Coconut in a production environment; how do I achieve maximum performance?

First, you're going to want a fast compiler, so you should either install Coconut with the `cPyparsing` option, or use `PyPy`. Second, there are two simple things you can do to make Coconut produce faster Python: compile with `--no-tco` and compile with a `--target` specification for the exact version of Python you want to run your code on. Passing `--target` helps Coconut optimize the compiled code for the Python version you want, and, though Tail Call Optimization is useful, it will usually significantly slow down functions that use it, so disabling it will often provide a major performance boost.

1.1.17 I want to contribute to Coconut, how do I get started?

That's great! Coconut is completely open-source, and new contributors are always welcome. Check out Coconut's contributing guidelines for more information.

1.1.18 Why the name Coconut?

If you don't get the reference, the image above is from Monty Python and the Holy Grail, in which the Knights of the Round Table bang Coconuts together to mimic the sound of riding a horse. The name was chosen to reference the fact that Python is named after Monty Python as well.

1.1.19 Who developed Coconut?

Evan Hubinger is an undergraduate student studying mathematics and computer science at Harvey Mudd College. He can be reached by asking a question on Coconut's Gitter chat room, through email at evanjhub@gmail.com, or on LinkedIn.

CHAPTER 2

Coconut Tutorial

2.1 Introduction

Welcome to the tutorial for the Coconut Programming Language! Coconut is a variant of Python built for **simple, elegant, Pythonic functional programming**. But those are just words; what they mean in practice is that *all valid Python 3 is valid Coconut* but Coconut builds on top of Python a suite of *simple, elegant utilities for functional programming*.

Why use Coconut? Coconut is built to be useful. Coconut enhances the repertoire of Python programmers to include the tools of modern functional programming, in such a way that those tools are *easy* to use and immensely *powerful;* that is, Coconut does to functional programming what Python did to imperative programming. And Coconut code runs the same on *any Python version*, making the Python 2/3 split a thing of the past.

Specifically, Coconut adds to Python *built-in, syntactical support* for:

- pattern-matching
- algebraic data types
- destructuring assignment
- partial application
- lazy lists
- function composition
- prettier lambdas
- infix notation
- pipeline-style programming
- operator functions
- tail call optimization
- parallel programming

and much more!

2.1.1 Installation

At its very core, Coconut is a compiler that turns Coconut code into Python code. That means that anywhere where you can use a Python script, you can also use a compiled Coconut script. To access that core compiler, Coconut comes with a command-line utility, which can

- compile single Coconut files or entire Coconut projects,
- interpret Coconut code on-the-fly, and
- hook into existing Python applications like IPython/Jupyter and MyPy.

Installing Coconut, including all the features above, is drop-dead simple. Just

1. install Python,
2. open a command-line prompt,
3. and enter:

```
pip install coconut
```

Note: Try re-running the above command with the `--user` option if you are encountering errors. Be sure that Coconut's installation location (on UNIX `/usr/local/bin` if you didn't use `--user` or `${HOME}/.local/bin/` if you did) is in your `PATH` environment variable. If you are still encountering errors installing Coconut with `pip`, you can also install Coconut with `conda` by following the conda installation instructions in the documentation.

To check that your installation is functioning properly, try entering into the command line

```
coconut -h
```

which should display Coconut's command-line help.

Note: If you're having trouble installing Coconut, or if anything else mentioned in this tutorial doesn't seem to work for you, feel free to ask for help on Gitter and somebody will try to answer your question as soon as possible.

2.2 Starting Out

2.2.1 Using the Interpreter

Now that you've got Coconut installed, the obvious first thing to do is to play around with it. To launch the Coconut interpreter, just go to the command line and type

```
coconut
```

and you should see something like

```
Coconut Interpreter:
(type 'exit()' or press Ctrl-D to end)
>>>
```

which is Coconut's way of telling you you're ready to start entering code for it to evaluate. So let's do that!

In case you missed it earlier, *all valid Python 3 is valid Coconut*. That doesn't mean compiled Coconut will only run on Python 3—in fact, compiled Coconut will run the same on any Python version—but it does mean that only Python 3 code is guaranteed to compile as Coconut code.

That means that if you're familiar with Python, you're already familiar with a good deal of Coconut's core syntax and Coconut's entire standard library. To show that, let's try entering some basic Python into the Coconut interpreter.

```
>>> "hello, world!"
'hello, world!'
>>> 1 + 1
2
```

2.2.2 Using the Compiler

Of course, while being able to interpret Coconut code on-the-fly is a great thing, it wouldn't be very useful without the ability to write and compile larger programs. To that end, it's time to write our first Coconut program: "hello, world!" Coconut-style.

First, we're going to need to create a file to put our code into. The recommended file extension for Coconut source files is `.coco`, so let's create the new file `hello_world.coco`. After you do that, you should take the time now to set up your text editor to properly highlight Coconut code. For instructions on how to do that, see the documentation on Coconut syntax highlighting.

Now let's put some code in our `hello_world.coco` file. Unlike in Python, where headers like

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
from __future__ import print_function, absolute_import, unicode_literals, division
```

are common and often very necessary, the Coconut compiler will automatically take care of all of that for you, so all you need to worry about is your own code. To that end, let's add the code for our "hello, world!" program.

In pure Python 3, "hello, world!" is

```
print("hello, world!")
```

and while that will work in Coconut, equally as valid is to use a pipeline-style approach, which is what we'll do, and write

```
"hello, world!" |> print
```

which should let you see very clearly how Coconut's `|>` operator enables pipeline-style programming: it allows an object to be passed along from function to function, with a different operation performed at each step. In this case, we are piping the object `"hello, world!"` into the operation `print`. Now let's save our simple "hello, world!" program, and try to run it.

Compiling Coconut files and projects with the Coconut command-line utility is incredibly simple. Just type

```
coconut hello_world.coco
```

which should give the output

```
Coconut: Compiling       hello_world.coco ...
Coconut: Compiled to     hello_world.py .
```

and deposit a new `hello_world.py` file in the same directory as the `hello_world.coco` file. You should then be able to run that file with

```
python hello_world.py
```

which should produce `hello, world!` as the output.

Note: You can compile and run your code all in one step if you use Coconut's `--run` option.

Compiling single files is not the only way to use the Coconut command-line utility, however. We can also compile all the Coconut files in a given directory simply by passing that directory as the first argument, which will get rid of the need to run the same Coconut header code in each file by storing it in a `__coconut__.py` file in the same directory.

The Coconut compiler supports a large variety of different compilation options, the help for which can always be accessed by entering `coconut -h` into the command line. One of the most useful of these is `--line-numbers` (or `-l` for short). Using `--line-numbers` will add the line numbers of your source code as comments in the compiled code, allowing you to see what line in your source code corresponds to a line in the compiled code where an error occurred, for ease of debugging.

2.2.3 Using IPython/Jupyter

Although all different types of programming can benefit from using more functional techniques, scientific computing, perhaps more than any other field, lends itself very well to functional programming, an observation the case studies in this tutorial are very good examples of. To that end, Coconut aims to provide extensive support for the established tools of scientific computing in Python.

That means supporting IPython/Jupyter, as modern Python programming, particularly in the sciences, has gravitated towards the use of IPython (the python kernel for the Jupyter framework) instead of the classic Python shell. Coconut supports being used as a kernel for Jupyter notebooks and consoles, allowing Coconut code to be used alongside powerful IPython features such as `%magic` commands.

To launch a Jupyter notebook with Coconut as the kernel, use the command

```
coconut --jupyter notebook
```

and to launch a Jupyter console, use the command

```
coconut --jupyter console
```

or equivalently, `--ipython` can be substituted for `--jupyter` in either command.

2.2.4 Case Studies

Because Coconut is built to be fundamentally *useful*, the best way to demo it is to show it in action. To that end, the majority of this tutorial will be showing how to apply Coconut to solve particular problems, which we'll call case studies.

These case studies are not intended to provide a complete picture of all of Coconut's features. For that, see Coconut's documentation. Instead, they are intended to show how Coconut can actually be used to solve practical programming problems.

2.3 Case Study 1: `factorial`

In the first case study we will be defining a `factorial` function, that is, a function that computes `n!` where `n` is an integer `>= 0`. This is somewhat of a toy example, since Python can fairly easily do this, but it will serve as a good showcase of some of the basic features of Coconut and how they can be used to great effect.

To start off with, we're going to have to decide what sort of an implementation of `factorial` we want. There are many different ways to tackle this problem, but for the sake of concision we'll split them into four major categories: imperative, recursive, iterative, and `addpattern`.

2.3.1 Imperative Method

The imperative approach is the way you'd write `factorial` in a language like C. Imperative approaches involve lots of state change, where variables are regularly modified and loops are liberally used. In Coconut, the imperative approach to the `factorial` problem looks like this:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    if n `isinstance` int and n >= 0:
        acc = 1
        for x in range(1, n+1):
            acc *= x
        return acc
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print  # TypeError
0.5 |> factorial |> print  # TypeError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Before we delve into what exactly is happening here, let's give it a run and make sure the test cases check out. If we were really writing a Coconut program, we'd want to save and compile an actual file, but since we're just playing around, let's try copy-pasting into the interpreter. Here, you should get two `TypeErrors`, then `1`, then `6`.

Now that we've verified it works, let's take a look at what's going on. Since the imperative approach is a fundamentally non-functional method, Coconut can't help us improve this example very much. Even here, though, the use of Coconut's infix notation (where the function is put in-between its arguments, surrounded in backticks) in `n `isinstance` int` makes the code slightly cleaner and easier to read.

2.3.2 Recursive Method

The recursive approach is the first of the fundamentally functional approaches, in that it doesn't involve the state change and loops of the imperative approach. Recursive approaches avoid the need to change variables by making that variable change implicit in the recursive function call. Here's the recursive approach to the `factorial` problem in Coconut:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    case n:
        match 0:
            return 1
        match x is int if x > 0:
            return x * factorial(x-1)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print  # TypeError
0.5 |> factorial |> print  # TypeError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Copy and paste the code and tests into the interpreter. You should get the same test results as you got for the imperative version—but you can probably tell there's quite a lot more going on here than there. That's intentional: Coconut is

intended for functional programming, not imperative programming, and so its new features are built to be most useful when programming in a functional style.

Let's take a look at the specifics of the syntax in this example. The first thing we see is `case n`. This statement starts a `case` block, in which only `match` statements can occur. Each `match` statement will attempt to match its given pattern against the value in the `case` block. Only the first successful match inside of any given `case` block will be executed. When a match is successful, any variable bindings in that match will also be performed. Additionally, as is true in this case, `match` statements can also have `if` guards that will check the given condition before the match is considered final. Finally, after the `case` block, an `else` statement is allowed, which will only be executed if no `match` statement is.

Specifically, in this example, the first `match` statement checks whether `n` matches to `0`. If it does, it executes `return 1`. Then the second `match` statement checks whether `n` matches to `x is int`, which checks that `n` is an `int` (using `isinstance`) and assigns `x = n` if so, then checks whether `x > 0`, and if so, executes `return x * factorial(x-1)`. If neither of those two statements are executed, the `else` statement triggers and executes `raise TypeError("the argument to factorial must be an integer >= 0")`.

Although this example is very basic, pattern-matching is both one of Coconut's most powerful and most complicated features. As a general intuitive guide, it is helpful to think *assignment* whenever you see the keyword `match`. A good way to showcase this is that all `match` statements can be converted into equivalent destructuring assignment statements, which are also valid Coconut. In this case, the destructuring assignment equivalent to the `factorial` function above would be:

```coconut
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    try:
        # The only value that can be assigned to 0 is 0, since 0 is an
        # immutable constant; thus, this assignment fails if n is not 0.
        0 = n
    except MatchError:
        pass
    else:
        return 1
    try:
        # This attempts to assign n to x, which has been declared to be
        # an int; since only an int can be assigned to an int, this
        # fails if n is not an int.
        x is int = n
    except MatchError:
        pass
    else: if x > 0:  # in Coconut, if, match, and try are allowed after else
        return x * factorial(x-1)
    raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print  # TypeError
0.5 |> factorial |> print  # TypeError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

First, copy and paste! While this destructuring assignment equivalent should work, it is much more cumbersome than `match` statements when you expect that they'll fail, which is why `match` statement syntax exists. But the destructuring assignment equivalent illuminates what exactly the pattern-matching is doing, by making it clear that `match` statements are really just fancy destructuring assignment statements. In fact, to be explicit about using destructuring assignment instead of normal assignment, the `match` keyword can be put before a destructuring assignment statement to signify it as such.

It will be helpful to, as we continue to use Coconut's pattern-matching and destructuring assignment statements in

further examples, think *assignment* whenever you see the keyword `match`.

Next, one easy improvement we can make to our `factorial` function is to make use of the wildcard pattern _. We don't actually need to assign `x` as a new variable, since it has the same value as `n`, so if we use _ instead of `x`, Coconut won't ever actually assign the variable. Thus, we can rewrite our `factorial` function as:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    case n:
        match 0:
            return 1
        match _ is int if n > 0:
            return n * factorial(n-1)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print  # TypeError
0.5 |> factorial |> print  # TypeError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Copy, paste! This new `factorial` function should behave exactly the same as before.

Up until now, for the recursive method, we have only dealt with pattern-matching, but there's actually another way that Coconut allows us to improve our `factorial` function. Coconut performs automatic tail call optimization, which means that whenever a function directly returns a call to another function, Coconut will optimize away the additional call. Thus, we can improve our `factorial` function by rewriting it to use a tail call:

```
def factorial(n, acc=1):
    """Compute n! where n is an integer >= 0."""
    case n:
        match 0:
            return acc
        match _ is int if n > 0:
            return factorial(n-1, acc*n)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print  # TypeError
0.5 |> factorial |> print  # TypeError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Copy, paste! This new `factorial` function is equivalent to the original version, with the exception that it will never raise a `RuntimeError` due to reaching Python's maximum recursion depth, since Coconut will optimize away the tail call.

2.3.3 Iterative Method

The other main functional approach is the iterative one. Iterative approaches avoid the need for state change and loops by using higher-order functions, those that take other functions as their arguments, like `map` and `reduce`, to abstract out the basic operations being performed. In Coconut, the iterative approach to the `factorial` problem is:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
```

```
    case n:
        match 0:
            return 1
        match _ is int if n > 0:
            return range(1, n+1) |> reduce$(*)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print  # TypeError
0.5 |> factorial |> print  # TypeError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Copy, paste! This definition differs from the recursive definition only by one line. That's intentional: because both the iterative and recursive approaches are functional approaches, Coconut can provide a great assist in making the code cleaner and more readable. The one line that differs is this one:

```
return range(1, n+1) |> reduce$(*)
```

Let's break down what's happening on this line. First, the `range` function constructs an iterator of all the numbers that need to be multiplied together. Then, it is piped into the function `reduce$(*)`, which does that multiplication. But how? What is `reduce$(*)`?

We'll start with the base, the `reduce` function. `reduce` used to exist as a built-in in Python 2, and Coconut brings it back. `reduce` is a higher-order function that takes a function of two arguments as its first argument, and an iterator as its second argument, and applies that function to the given iterator by starting with the first element, and calling the function on the accumulated call so far and the next element, until the iterator is exhausted. Here's a visual representation:

```
reduce(f, (a, b, c, d))

acc                 iter
                    (a, b, c, d)
a                   (b, c, d)
f(a, b)             (c, d)
f(f(a, b), c)       (d)
f(f(f(a, b), c), d)

return acc
```

Now let's take a look at what we do to `reduce` to make it multiply all the numbers we feed into it together. The Coconut code that we saw for that was `reduce$(*)`. There are two different Coconut constructs being used here: the operator function for multiplication in the form of `(*)`, and partial application in the form of `$`.

First, the operator function. In Coconut, a function form of any operator can be retrieved by surrounding that operator in parentheses. In this case, `(*)` is roughly equivalent to `lambda x, y: x*y`, but much cleaner and neater. In Coconut's lambda syntax, `(*)` is also equivalent to `(x, y) -> x*y`, which we will use from now on for all lambdas, even though both are legal Coconut, because Python's `lambda` statement is too ugly and bulky to use regularly.

Note: If Coconut's `--strict` mode is enabled, which will force your code to obey certain cleanliness standards, it will raise an error whenever Python `lambda` statements are used.

Second, the partial application. Think of partial application as *lazy function calling*, and `$` as the *lazy-ify* operator, where lazy just means "don't evaluate this until you need to." In Coconut, if a function call is prefixed by a `$`, like in this example, instead of actually performing the function call, a new function is returned with the given arguments already provided to it, so that when it is then called, it will be called with both the partially-applied arguments and

the new arguments, in that order. In this case, `reduce$(*)` is roughly equivalent to `(*args, **kwargs) -> reduce((*), *args, **kwargs)`.

Putting it all together, we can see how the single line of code

```
range(1, n+1) |> reduce$(*)
```

is able to compute the proper factorial, without using any state or loops, only higher-order functions, in true functional style. By supplying the tools we use here like partial application (`$`), pipeline-style programming (`|>`), higher-order functions (`reduce`), and operator functions (`(*)`), Coconut enables this sort of functional programming to be done cleanly, neatly, and easily.

2.3.4 `addpattern` Method

While the iterative approach is very clean, there are still some bulky pieces—looking at the iterative version below, you can see that it takes three entire indentation levels to get from the function definition to the actual objects being returned:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    case n:
        match 0:
            return 1
        match _ is int if n > 0:
            return range(1, n+1) |> reduce$(*)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")
```

By making use of the Coconut built-in `addpattern`, we can take that from three indentation levels down to one. Take a look:

```
def factorial(0) = 1

@addpattern(factorial)
def factorial(n is int if n > 0) =
    """Compute n! where n is an integer >= 0."""
    range(1, n+1) |> reduce$(*)

# Test cases:
-1 |> factorial |> print  # MatchError
0.5 |> factorial |> print  # MatchError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Copy, paste! This should work exactly like before, except now it raises `MatchError` as a fall through instead of `TypeError`. There are three major new concepts to talk about here: `addpattern`, of course, assignment function notation, and pattern-matching function definition—how both of the functions above are defined.

First, assignment function notation. This one's pretty straightforward. If a function is defined with an `=` instead of a `:`, the last line is required to be an expression, and is automatically returned.

Second, pattern-matching function definition. Pattern-matching function definition does exactly that—pattern-matches against all the arguments that are passed to the function. Unlike normal function definition, however, if the pattern doesn't match (if for example the wrong number of arguments are passed), your function will raise a `MatchError`. Finally, like destructuring assignment, if you want to be more explicit about using pattern-matching function definition, you can add a `match` before the `def`.

Third, `addpattern`. `addpattern` takes one argument, a previously-defined pattern-matching function, and returns a decorator that decorates a new pattern-matching function by adding the new pattern as an additional case to the old patterns. Thus, `addpattern` can be thought of as doing exactly what it says—it adds a new pattern to an existing pattern-matching function.

Finally, not only can we rewrite the iterative approach using `addpattern`, as we did above, we can also rewrite the recursive approach using `addpattern`, like so:

```
def factorial(0) = 1

@addpattern(factorial)
def factorial(n is int if n > 0) =
    """Compute n! where n is an integer >= 0."""
    n * factorial(n - 1)

# Test cases:
-1 |> factorial |> print  # MatchError
0.5 |> factorial |> print  # MatchError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Copy, paste! It should work exactly like before, except, as above, with `TypeError` replaced by `MatchError`.

2.4 Case Study 2: `quick_sort`

In the second case study, we will be implementing the quick sort algorithm. We will implement two versions: first, a `quick_sort` function that takes in a list and outputs a list, and second, a `quick_sort` function that takes in an iterator and outputs an iterator.

2.4.1 Sorting a Sequence

First up is `quick_sort` for lists. We're going to use a recursive `addpattern`-based approach to tackle this problem—a similar approach to the very last `factorial` function we wrote, using `addpattern` to reduce the amount of indentation we're going to need. Without further ado, here's our implementation of `quick_sort` for lists:

```
def quick_sort([]) = []

@addpattern(quick_sort)
def quick_sort([head] + tail) =
    """Sort the input sequence using the quick sort algorithm."""
    (quick_sort([x for x in tail if x < head])
        + [head]
        + quick_sort([x for x in tail if x >= head]))

# Test cases:
[] |> quick_sort |> print  # []
[3] |> quick_sort |> print  # [3]
[0,1,2,3,4] |> quick_sort |> print  # [0,1,2,3,4]
[4,3,2,1,0] |> quick_sort |> print  # [0,1,2,3,4]
[3,0,4,2,1] |> quick_sort |> print  # [0,1,2,3,4]
```

Copy, paste! Only one new feature here: head-tail pattern-matching. Here, we see the head-tail pattern `[head] + tail`, which more generally just follow the form of a list or tuple added to a variable. When this appears in any pattern-matching context, the value being matched against will be treated as a sequence, the list or tuple matched

against the beginning of that sequence, and the rest of it bound to the variable. In this case, we use the head-tail pattern to remove the head so we can use it as the pivot for splitting the rest of the list.

2.4.2 Sorting an Iterator

Now it's time to try `quick_sort` for iterators. Our method for tackling this problem is going to be a combination of the recursive and iterative approaches we used for the `factorial` problem, in that we're going to be lazily building up an iterator, and we're going to be doing it recursively. Here's the code:

```
def quick_sort(l):
    """Sort the input iterator using the quick sort algorithm."""
    match [head] :: tail in l:
        tail = reiterable(tail)
        yield from (quick_sort(x for x in tail if x < head)
            :: (head,)
            :: quick_sort(x for x in tail if x >= head)
            )
    # We implicitly return an empty iterator here if the match falls through.

# Test cases:
[] |> quick_sort |> list |> print  # []
[3] |> quick_sort |> list |> print  # [3]
[0,1,2,3,4] |> quick_sort |> list |> print  # [0,1,2,3,4]
[4,3,2,1,0] |> quick_sort |> list |> print  # [0,1,2,3,4]
[3,0,4,2,1] |> quick_sort |> list |> print  # [0,1,2,3,4]
```

Copy, paste! This `quick_sort` algorithm works uses a bunch of new constructs, so let's go over them.

First, the `::` operator, which appears here both in pattern-matching and by itself. In essence, the `::` operator is lazy `+` for iterators. On its own, it takes two iterators and concatenates, or chains, them together, and it does this lazily, not evaluating anything until its needed, so it can be used for making infinite iterators. In pattern-matching, it inverts that operation, destructuring the beginning of an iterator into a pattern, and binding the rest of that iterator to a variable.

Which brings us to the second new thing, `match ... in ...` notation. The notation

```
match pattern in item:
    <body>
else:
    <else>
```

is shorthand for

```
case item:
    match pattern:
        <body>
else:
    <else>
```

that avoids the need for an additional level of indentation when only one `match` is being performed.

The third new construct is the Coconut built-in `reiterable`. There is a problem in doing immutable functional programming with Python iterators: whenever an element of an iterator is accessed, it's lost. `reiterable` solves this problem by allowing the iterable it's called on to be iterated over multiple times while still yielding the same result each time

Finally, although it's not a new construct, since it exists in Python 3, the use of `yield from` here deserves a mention. In Python, `yield` is the statement used to construct iterators, functioning much like `return`, with the exception that

multiple `yield`s can be encountered, and each one will produce another element. `yield from` is very similar, except instead of adding a single element to the produced iterator, it adds another whole iterator.

Putting it all together, here's our `quick_sort` function again:

```
def quick_sort(l):
    """Sort the input iterator using the quick sort algorithm."""
    match [head] :: tail in l:
        tail = reiterable(tail)
        yield from (quick_sort(x for x in tail if x < head)
            :: (head,)
            :: quick_sort(x for x in tail if x >= head)
            )
    # We implicitly return an empty iterator here if the match falls through.
```

The function first attempts to split `l` into an initial element and a remaining iterator. If `l` is the empty iterator, that match will fail, and it will fall through, yielding the empty iterator (that's how the function handles the base case). Otherwise, we make a copy of the rest of the iterator, and yield the join of (the quick sort of all the remaining elements less than the initial element), (the initial element), and (the quick sort of all the remaining elements greater than the initial element).

The advantages of the basic approach used here, heavy use of iterators and recursion, as opposed to the classical imperative approach, are numerous. First, our approach is more clear and more readable, since it is describing *what* `quick_sort` is instead of *how* `quick_sort` could be implemented. Second, our approach is *lazy* in that our `quick_sort` won't evaluate any data until it needs it. Finally, and although this isn't relevant for `quick_sort` it is relevant in many other cases, an example of which we'll see later in this tutorial, our approach allows for working with *infinite* series just like they were finite.

And Coconut makes programming in such an advantageous functional approach significantly easier. In this example, Coconut's pattern-matching lets us easily split the given iterator, and Coconut's `::` iterator joining operator lets us easily put it back together again in sorted order.

2.5 Case Study 3: `vector` Part I

In the next case study, we'll be doing something slightly different—instead of defining a function, we'll be creating an object. Specifically, we're going to try to implement an immutable n-vector that supports all the basic vector operations.

In functional programming, it is often very desirable to define *immutable* objects, those that can't be changed once created—like Python's strings or tuples. Like strings and tuples, immutable objects are useful for a wide variety of reasons:

- they're easier to reason about, since you can be guaranteed they won't change,
- they're hashable and pickleable, so they can be used as keys and serialized,
- they're significantly more efficient since they require much less overhead,
- and when combined with pattern-matching, they can be used as what are called *algebraic data types* to build up and then match against large, complicated data structures very easily.

2.5.1 2-Vector

Coconut's `data` statement brings the power and utility of *immutable, algebraic data types* to Python, and it is this that we will be using to construct our `vector` type. The demonstrate the syntax of `data` statements, we'll start

by defining a simple 2-vector. Our vector will have one special method `__abs__` which will compute the vector's magnitude, defined as the square root of the sum of the squares of the elements. Here's our 2-vector:

```
data vector2(x, y):
    """Immutable 2-vector."""
    def __abs__(self) =
        """Return the magnitude of the 2-vector."""
        (self.x**2 + self.y**2)**0.5

# Test cases:
vector2(1, 2) |> print  # vector2(x=1, y=2)
vector2(3, 4) |> abs |> print  # 5
vector2(1, 2) |> fmap$(x -> x*2) |> print  # vector2(x=2, y=4)
v = vector2(2, 3)
v.x = 7  # AttributeError
```

Copy, paste! This example shows the basic syntax of `data` statements:

```
data <name>(<attributes>):
    <body>
```

where `<name>` and `<body>` are the same as the equivalent `class` definition, but `<attributes>` are the different attributes of the data type, in order that the constructor should take them as arguments. In this case, `vector2` is a data type of two attributes, `x` and `y`, with one defined method, `__abs__`, that computes the magnitude. As the test cases show, we can then create, print, but *not modify* instances of `vector2`.

One other thing to call attention to here is the use of the Coconut built-in `fmap`. `fmap` allows you to map functions over algebraic data types. In fact, Coconut's `data` types support iteration, so the standard `map` works on them, but it doesn't return another object of the same data type. Thus, `fmap` is simply `map` plus a call to the object's constructor.

2.5.2 n-Vector Constructor

Now that we've got the 2-vector under our belt, let's move to back to our original, more complicated problem: n-vectors, that is, vectors of arbitrary length. We're going to try to make our n-vector support all the basic vector operations, but we'll start out with just the `data` definition and the constructor:

```
data vector(*pts):
    """Immutable n-vector."""
    def __new__(cls, *pts):
        """Create a new vector from the given pts."""
        match [v is vector] in pts:
            return v  # vector(v) where v is a vector should return v
        else:
            return pts |*> makedata$(cls)  # accesses base constructor

# Test cases:
vector(1, 2, 3) |> print  # vector(*pts=(1, 2, 3))
vector(4, 5) |> vector |> print  # vector(*pts=(4, 5))
```

Copy, paste! The big new thing here is how to write `data` constructors. Since `data` types are immutable, `__init__` construction won't work. Instead, a different special method `__new__` is used, which must return the newly constructed instance, and unlike most methods, takes the class not the object as the first argument. Since `__new__` needs to return a fully constructed instance, in almost all cases it will be necessary to access the underlying `data` constructor. To achieve this, Coconut provides the built-in `makedata` function, which takes a data type and calls its underlying `data` constructor with the rest of the arguments.

In this case, the constructor checks whether nothing but another `vector` was passed, in which case it returns that, otherwise it returns the result of passing the arguments to the underlying constructor, the form of which is `vector(*pts)`, since that is how we declared the data type. We use sequence pattern-matching to determine whether we were passed a single vector, which is just a list or tuple of patterns to match against the contents of the sequence.

The other new construct used here is the `|*>`, or star-pipe, operator, which functions exactly like the normal pipe, except that instead of calling the function with one argument, it calls it with as many arguments as there are elements in the sequence passed into it. The difference between `|*>` and `|>` is exactly analogous to the difference between `f(args)` and `f(*args)`.

2.5.3 n-Vector Methods

Now that we have a constructor for our n-vector, it's time to write its methods. First up is `__abs__`, which should compute the vector's magnitude. This will be slightly more complicated than with the 2-vector, since we have to make it work over an arbitrary number of `pts`. Fortunately, we can use Coconut's pipeline-style programming and partial application to make it simple:

```
    def __abs__(self) =
        """Return the magnitude of the vector."""
        self.pts |> map$(pow$(?, 2)) |> sum |> pow$(?, 0.5)
```

The basic algorithm here is map square over each element, sum them all, then square root the result. The one new construct used here is the use of a `?` in partial application, which simply allows skipping an argument from being partially applied and deferring it to when the function is called. In this case, the `?` lets us partially apply the exponent instead of the base in `pow` (we could also have equivalently used `(**)`).

Next up is vector addition. The goal here is to add two vectors of equal length by adding their components. To do this, we're going to make use of Coconut's ability to perform pattern-matching, or in this case destructuring assignment, to data types, like so:

```
    def __add__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Add two vectors together."""
        map((+), self.pts, other_pts) |*> vector
```

There are a couple of new constructs here, but the main notable one is the pattern-matching `vector(*other_pts)` which showcases the syntax for pattern-matching against data types: it mimics exactly the original `data` declaration of that data type. In this case, `vector(*other_pts)` will only match a vector, raising a `MatchError` otherwise, and if it does match a vector, will assign the vector's `pts` attribute to the variable `other_pts`.

Next is vector subtraction, which is just like vector addition, but with `(-)` instead of `(+)`:

```
    def __sub__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Subtract one vector from another."""
        map((-), self.pts, other_pts) |*> vector
```

One thing to note here is that unlike the other operator functions, `(-)` can either mean negation or subtraction, the meaning of which will be inferred based on how many arguments are passed, 1 for negation, 2 for subtraction. To show this, we'll use the same `(-)` function to implement vector negation, which should simply negate each element:

```
    def __neg__(self) =
        """Retrieve the negative of the vector."""
        self.pts |> map$(-) |*> vector
```

The last method we'll implement is multiplication. This one is a little bit tricky, since mathematically, there are a whole bunch of different ways to multiply vectors. For our purposes, we're just going to look at two: between two vectors of equal length, we want to compute the dot product, defined as the sum of the corresponding elements multiplied together, and between a vector and a scalar, we want to compute the scalar multiple, which is just each element multiplied by that scalar. Here's our implementation:

```
    def __mul__(self, other):
        """Scalar multiplication and dot product."""
        match vector(*other_pts) in other:
            assert len(other_pts) == len(self.pts)
            return map((*), self.pts, other_pts) |> sum  # dot product
        else:
            return self.pts |> map$((*)$(other)) |*> vector  # scalar multiple
    def __rmul__(self, other) =
        """Necessary to make scalar multiplication commutative."""
        self * other
```

The first thing to note here is that unlike with addition and subtraction, where we wanted to raise an error if the vector match failed, here, we want to do scalar multiplication if the match fails, so instead of using destructuring assignment, we use a `match` statement. The second thing to note here is the combination of pipeline-style programming, partial application, operator functions, and higher-order functions we're using to compute the dot product and scalar multiple. For the dot product, we map multiplication over the two vectors, then sum the result. For the scalar multiple, we take the original points, map multiplication by the scalar over them, then use them to make a new vector.

Finally, putting everything together:

```
data vector(*pts):
    """Immutable n-vector."""
    def __new__(cls, *pts):
        """Create a new vector from the given pts."""
        match [v is vector] in pts:
            return v  # vector(v) where v is a vector should return v
        else:
            return pts |*> makedata$(cls)  # accesses base constructor
    def __abs__(self) =
        """Return the magnitude of the vector."""
        self.pts |> map$(pow$(?, 2)) |> sum |> pow$(?, 0.5)
    def __add__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Add two vectors together."""
        map((+), self.pts, other_pts) |*> vector
    def __sub__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Subtract one vector from another."""
        map((-), self.pts, other_pts) |*> vector
    def __neg__(self) =
        """Retrieve the negative of the vector."""
        self.pts |> map$(-) |*> vector
    def __mul__(self, other):
        """Scalar multiplication and dot product."""
        match vector(*other_pts) in other:
            assert len(other_pts) == len(self.pts)
            return map((*), self.pts, other_pts) |> sum  # dot product
        else:
            return self.pts |> map$((*)$(other)) |*> vector  # scalar multiplication
    def __rmul__(self, other) =
        """Necessary to make scalar multiplication commutative."""
        self * other
```

```
# Test cases:
vector(1, 2, 3) |> print  # vector(*pts=(1, 2, 3))
vector(4, 5) |> vector |> print  # vector(*pts=(4, 5))
vector(3, 4) |> abs |> print  # 5
vector(1, 2) + vector(2, 3) |> print  # vector(*pts=(3, 5))
vector(2, 2) - vector(0, 1) |> print  # vector(*pts=(2, 1))
-vector(1, 3) |> print  # vector(*pts=(-1, -3))
(vector(1, 2) == "string") |> print  # False
(vector(1, 2) == vector(3, 4)) |> print  # False
(vector(2, 4) == vector(2, 4)) |> print  # True
2*vector(1, 2) |> print  # vector(*pts=(2, 4))
vector(1, 2) * vector(1, 3) |> print  # 7
```

Copy, paste! Now that was a lot of code. But looking it over, it looks clean, readable, and concise, and it does precisely what we intended it to do: create an algebraic data type for an immutable n-vector that supports the basic vector operations. And we did the whole thing without needing any imperative constructs like state or loops—pure functional programming.

2.6 Case Study 4: `vector_field`

For the final case study, instead of me writing the code, and you looking at it, you'll be writing the code—of course, I won't be looking at it, but I will show you how I would have done it after you give it a shot by yourself.

The bonus challenge for this section is to write each of the functions we'll be defining in just one line. Try using assignment functions to help with that!

First, let's introduce the general goal of this case study. We want to write a program that will allow us to produce infinite vector fields that we can iterate over and apply operations to. And in our case, we'll say we only care about vectors with positive components.

Our first step, therefore, is going to be creating a field of all the points with positive `x` and `y` values—that is, the first quadrant of the `x-y` plane, which looks something like this:

```
...

(0,2)   ...

(0,1)   (1,1)   ...

(0,0)   (1,0)   (2,0)   ...
```

But since we want to be able to iterate over that plane, we're going to need to linearize it somehow, and the easiest way to do that is to split it up into diagonals, and traverse the first diagonal, then the second diagonal, and so on, like this:

```
(0, 0), (1, 0), (0, 1), (2, 0), (1, 1), (0, 2), ...
```

2.6.1 `diagonal_line`

Thus, our first function `diagonal_line(n)` should construct an iterator of all the points, represented as coordinate tuples, in the `n`th diagonal, starting with `(0, 0)` as the `0`th diagonal. Like we said at the start of this case study, this is where we I let go and you take over. Using all the tools of functional programming that Coconut provides, give `diagonal_line` a shot. When you're ready to move on, scroll down.

Here are some tests that you can use:

```
diagonal_line(0) `isinstance` (list, tuple) |> print  # False (should be an iterator)
diagonal_line(0) |> list |> print  # [(0, 0)]
diagonal_line(1) |> list |> print  # [(0, 1), (1, 0)]
```

Hint: the nth diagonal should contain `n+1` elements, so try starting with `range(n+1)` and then transforming it in some way.

That wasn't so bad, now was it? Now, let's take a look at my solution:

```
def diagonal_line(n) = range(n+1) |> map$(i -> (i, n-i))
```

Pretty simple, huh? We take `range(n+1)`, and use `map` to transform it into the right sequence of tuples.

2.6.2 linearized_plane

Now that we've created our diagonal lines, we need to join them together to make the full linearized plane, and to do that we're going to write the function `linearized_plane()`. `linearized_plane` should produce an iterator that goes through all the points in the plane, in order of all the points in the first `diagonal(0)`, then the second `diagonal(1)`, and so on. `linearized_plane` is going to be, by necessity, an infinite iterator, since it needs to loop through all the points in the plane, which have no end. To help you accomplish this, remember that the `::` operator is lazy, and won't evaluate its operands until they're needed, which means it can be used to construct infinite iterators. When you're ready to move on, scroll down.

Tests:

```
# Note: these tests use $[] notation, which we haven't introduced yet
#  but will introduce later in this case study; for now, just run the
#  tests, and make sure you get the same result as is in the comment
linearized_plane()$[0] |> print  # (0, 0)
linearized_plane()$[:3] |> list |> print  # [(0, 0), (0, 1), (1, 0)]
```

Hint: instead of defining the function as `linearized_plane()`, try defining it as `linearized_plane(n=0)`, where `n` is the diagonal to start at, and use recursion to build up from there.

That was a little bit rougher than the first one, but hopefully still not too bad. Let's compare to my solution:

```
def linearized_plane(n=0) = diagonal_line(n) :: linearized_plane(n+1)
```

As you can see, it's a very fundamentally simple solution: just use `::` and recursion to join all the diagonals together in order.

2.6.3 vector_field

Now that we have a function that builds up all the points we need, it's time to turn them into vectors, and to do that we'll define the new function `vector_field()`, which should turn all the tuples in `linearized_plane` into vectors, using the n-vector class we defined earlier.

Tests:

```
# You'll need to bring in the vector class from earlier to make these work
vector_field()$[0] |> print  # vector(*pts=(0, 0))
vector_field()$[2:3] |> list |> print  # [vector(*pts=(1, 0))]
```

Hint: Remember, the way we defined vector it takes the components as separate arguments, not a single tuple. You may find the Coconut built-in `starmap` *useful in dealing with that.*

We're making good progress! Before we move on, check your solution against mine:

```
def vector_field() = linearized_plane() |> starmap$(vector)
```

All we're doing is taking our `linearized_plane` and mapping `vector` over it, but using `starmap` instead of `map` so that `vector` gets called with each element of the tuple as a separate argument.

2.6.4 Applications

Now that we've built all the functions we need for our vector field, it's time to put it all together and test it. Feel free to substitute in your versions of the functions below:

```
data vector(*pts):
    """Immutable n-vector."""
    def __new__(cls, *pts):
        """Create a new vector from the given pts."""
        match [v is vector] in pts:
            return v  # vector(v) where v is a vector should return v
        else:
            return pts |*> makedata$(cls)  # accesses base constructor
    def __abs__(self) =
        """Return the magnitude of the vector."""
        self.pts |> map$(pow$(?, 2)) |> sum |> pow$(?, 0.5)
    def __add__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Add two vectors together."""
        map((+), self.pts, other_pts) |*> vector
    def __sub__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Subtract one vector from another."""
        map((-), self.pts, other_pts) |*> vector
    def __neg__(self) =
        """Retrieve the negative of the vector."""
        self.pts |> map$(-) |*> vector
    def __mul__(self, other):
        """Scalar multiplication and dot product."""
        match vector(*other_pts) in other:
            assert len(other_pts) == len(self.pts)
            return map((*), self.pts, other_pts) |> sum  # dot product
        else:
            return self.pts |> map$((*)$(other)) |*> vector  # scalar multiplication
    def __rmul__(self, other) =
        """Necessary to make scalar multiplication commutative."""
        self * other

def diagonal_line(n) = range(n+1) |> map$(i -> (i, n-i))
def linearized_plane(n=0) = diagonal_line(n) :: linearized_plane(n+1)
def vector_field() = linearized_plane() |> starmap$(vector)

# Test cases:
diagonal_line(0) `isinstance` (list, tuple) |> print  # False (should be an iterator)
diagonal_line(0) |> list |> print  # [(0, 0)]
diagonal_line(1) |> list |> print  # [(0, 1), (1, 0)]
linearized_plane()$[0] |> print  # (0, 0)
```

```
linearized_plane()$[:3] |> list |> print  # [(0, 0), (0, 1), (1, 0)]
vector_field()$[0] |> print  # vector(*pts=(0, 0))
vector_field()$[2:3] |> list |> print  # [vector(*pts=(1, 0))]
```

Copy, paste! Once you've made sure everything is working correctly if you substituted in your own functions, take a look at the last 4 tests. You'll notice that they use a new notation, similar to the notation for partial application we saw earlier, but with brackets instead of parentheses. This is the notation for iterator slicing. Similar to how partial application was lazy function calling, iterator slicing is *lazy sequence slicing*. Like with partial application, it is helpful to think of `$` as the *lazy-ify* operator, in this case turning normal Python slicing, which is evaluated immediately, into lazy iterator slicing, which is evaluated only when the elements in the slice are needed.

With that in mind, now that we've built our vector field, it's time to use iterator slicing to play around with it. Try doing something cool to our vector fields like

- create a `magnitude_field` where each point is that vector's magnitude
- combine entire vector fields together with `map` and the vector addition and multiplication methods we wrote earlier

then use iterator slicing to take out portions and examine them.

2.7 Case Study 5: `vector` Part II

For the some of the applications you might want to use your `vector_field` for, it might be desirable to add some useful methods to our `vector`. In this case study, we're going to be focusing on one in particular: `.angle`.

`.angle` will take one argument, another vector, and compute the angle between the two vectors. Mathematically, the formula for the angle between two vectors is the dot product of the vectors' respective unit vectors. Thus, before we can implement `.angle`, we're going to need `.unit`. Mathematically, the formula for the unit vector of a given vector is that vector divided by its magnitude. Thus, before we can implement `.unit`, and by extension `.angle`, we'll need to start by implementing division.

2.7.1 `__truediv__`

Vector division is just scalar division, so we're going to write a `__truediv__` method that takes `self` as the first argument and `other` as the second argument, and returns a new vector the same size as `self` with every element divided by `other`. For an extra challenge, try writing this one in one line using assignment function notation.

Tests:

```
vector(3, 4) / 1 |> print  # vector(*pts=(3.0, 4.0))
vector(2, 4) / 2 |> print  # vector(*pts=(1.0, 2.0))
```

Hint: Look back at how we implemented scalar multiplication.

Here's my solution for you to check against:

```
    def __truediv__(self, other) = self.pts |> map$(x -> x/other) |*> vector
```

2.7.2 `.unit`

Next up, `.unit`. We're going to write a `unit` method that takes just `self` as its argument and returns a new vector the same size as `self` with each element divided by the magnitude of `self`, which we can retrieve with `abs`. This should be a very simple one-line function.

Tests:

```
vector(0, 1).unit() |> print  # vector(*pts=(0.0, 1.0))
vector(5, 0).unit() |> print  # vector(*pts=(1.0, 0.0))
```

Here's my solution:

```
    def unit(self) = self / abs(self)
```

2.7.3 `.angle`

This one is going to be a little bit more complicated. For starters, the mathematical formula for the angle between two vectors is the `math.acos` of the dot product of those vectors' respective unit vectors, and recall that we already implemented the dot product of two vectors when we wrote `__mul__`. So, `.angle` should take `self` as the first argument and `other` as the second argument, and if `other` is a vector, use that formula to compute the angle between `self` and `other`, or if `other` is not a vector, `.angle` should raise a `MatchError`. To accomplish this, we're going to want to use destructuring assignment to check that `other` is indeed a `vector`.

Tests:

```
import math
vector(2, 0).angle(vector(3, 0)) |> print  # 0.0
print(vector(1, 0).angle(vector(0, 2)), math.pi/2)  # should be the same
vector(1, 2).angle(5)  # MatchError
```

Hint: Look back at how we checked whether the argument to `factorial` was an integer using destructuring assignment.

Here's my solution—take a look:

```
    def angle(self, other is vector) = math.acos(self.unit() * other.unit())
```

And now it's time to put it all together. Feel free to substitute in your own versions of the methods we just defined.

```
import math  # necessary for math.acos in .angle

data vector(*pts):
    """Immutable n-vector."""
    def __new__(cls, *pts):
        """Create a new vector from the given pts."""
        match [v is vector] in pts:
            return v  # vector(v) where v is a vector should return v
        else:
            return pts |*> makedata$(cls)  # accesses base constructor
    def __abs__(self) =
        """Return the magnitude of the vector."""
        self.pts |> map$(pow$(?, 2)) |> sum |> pow$(?, 0.5)
    def __add__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Add two vectors together."""
        map((+), self.pts, other_pts) |*> vector
    def __sub__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Subtract one vector from another."""
        map((-), self.pts, other_pts) |*> vector
    def __neg__(self) =
        """Retrieve the negative of the vector."""
```

```
        self.pts |> map$(-) |*> vector
    def __mul__(self, other):
        """Scalar multiplication and dot product."""
        match vector(*other_pts) in other:
            assert len(other_pts) == len(self.pts)
            return map((*), self.pts, other_pts) |> sum  # dot product
        else:
            return self.pts |> map$((*)$(other)) |*> vector  # scalar multiplication
    def __rmul__(self, other) =
        """Necessary to make scalar multiplication commutative."""
        self * other
     # New one-line functions necessary for finding the angle between vectors:
    def __truediv__(self, other) = self.pts |> map$(x -> x/other) |*> vector
    def unit(self) = self / abs(self)
    def angle(self, other is vector) = math.acos(self.unit() * other.unit())

# Test cases:
vector(3, 4) / 1 |> print  # vector(*pts=(3.0, 4.0))
vector(2, 4) / 2 |> print  # vector(*pts=(1.0, 2.0))
vector(0, 1).unit() |> print  # vector(*pts=(0.0, 1.0))
vector(5, 0).unit() |> print  # vector(*pts=(1.0, 0.0))
vector(2, 0).angle(vector(3, 0)) |> print  # 0.0
print(vector(1, 0).angle(vector(0, 2)), math.pi/2)  # should be the same
vector(1, 2).angle(5)  # MatchError
```

One note of warning here: be careful not to leave a blank line when substituting in your methods, or the interpreter will cut off the code for the `vector` *there. This isn't a problem in normal Coconut code, only here because we're copy-and-pasting into the command line.*

Copy, paste! If everything is working, I'd recommend going back to playing around with `vector_field` *applications* using our new methods.

2.8 Filling in the Gaps

And with that, this tutorial is out of case studies—but that doesn't mean Coconut is out of features! In this last section, we'll touch on three of the most important features of Coconut that we managed to miss in our case studies: lazy lists, function composition, and implicit partials.

2.8.1 Lazy Lists

First up is lazy lists. Lazy lists are lazily-evaluated iterator literals, similar in their laziness to Coconut's `::` operator, in that any expressions put inside a lazy list won't be evaluated until that element of the lazy list is needed. The syntax for lazy lists is exactly the same as the syntax for normal lists, but with "banana brackets" (`(|` and `|)`) instead of normal brackets, like so:

```
abc = (| a, b, c |)
```

2.8.2 Function Composition

Next is function composition. In Coconut, this is accomplished through the `..` operator, which takes two functions and composes them, creating a new function equivalent to `(*args, **kwargs) -> f1(f2(*args,`

`**kwargs))`. This can be useful in combination with partial application for piecing together multiple higher-order functions, like so:

```
zipsum = map$(sum)..zip
```

Function composition also gets rid of the need for lots of parentheses when chaining function calls, like so:

```
plus1..square(3) == 10
```

Note: Coconut also supports the function composition pipe operators `..>`, `<..`, `..*>`, and `<*..`.

2.8.3 Implicit Partials

Last is implicit partials. Coconut supports a number of different "incomplete" expressions that will evaluate to a function that takes in the part necessary to complete them, that is, an implicit partial application function. The different allowable expressions are:

```
.attr
.method(args)
obj.
func$
seq[]
iter$[]
.[slice]
.$[slice]
```

2.8.4 Type Annotations

For many people, one of the big downsides of Python is the fact that it is dynamically-typed. In Python, this problem is addressed by MyPy, a static type analyzer for Python, which can check Python-3-style type annotations such as

```
def plus1(x: int) -> int:
    return x + 1
a: int = plus1(10)
```

Unfortunately, in Python, such type annotation syntax only exists in Python 3. Not to worry in Coconut, however, which compiles Python-3-style type annotations to universally compatible type comments. Not only that, but Coconut has built-in MyPy integration for automatically type-checking your code, and its own enhanced type annotation syntax for more easily expressing complex types.

2.8.5 Further Reading

And that's it for this tutorial! But that's hardly it for Coconut. All of the features examined in this tutorial, as well as a bunch of others, are detailed in Coconut's documentation.

Also, if you have any other questions not covered in this tutorial, feel free to ask around at Coconut's Gitter, a GitHub-integrated chat room for Coconut developers.

Finally, Coconut is a new, growing language, and if you'd like to get involved in the development of Coconut, all the code is available completely open-source on Coconut's GitHub. Contributing is a simple as forking the code, making your changes, and proposing a pull request! See Coconuts contributing guidelines for more information.

CHAPTER 3

Coconut Documentation

3.1 Overview

This documentation covers all the features of the Coconut Programming Language, and is intended as a reference/specification, not a tutorialized introduction. For a full introduction and tutorial of Coconut, see the tutorial.

Coconut is a variant of Python built for **simple, elegant, Pythonic functional programming**. Coconut syntax is a strict superset of Python 3 syntax. Thus, users familiar with Python will already be familiar with most of Coconut.

The Coconut compiler turns Coconut code into Python code. The primary method of accessing the Coconut compiler is through the Coconut command-line utility, which also features an interpreter for real-time compilation. In addition to the command-line utility, Coconut also supports the use of IPython/Jupyter notebooks.

While most of Coconut gets its inspiration simply from trying to make functional programming work in Python, additional inspiration came from Haskell, CoffeeScript, F#, and patterns.py.

3.2 Compilation

3.2.1 Installation

Using Pip

Since Coconut is hosted on the Python Package Index, it can be installed easily using `pip`. Simply install Python, open up a command-line prompt, and enter

```
pip install coconut
```

which will install Coconut and its required dependencies.

Note: If you have an old version of Coconut installed and you want to upgrade, run `pip install --upgrade coconut` instead.

If you are encountering errors running `pip install coconut`, try re-running it with the `--user` option. If `pip install coconut` works, but you cannot access the `coconut` command, be sure that Coconut's installation location is in your `PATH` environment variable. On UNIX, that is `/usr/local/bin` (without `--user`) or `${HOME}/.local/bin/` (with `--user`).

Using Conda

If you prefer to use `conda` instead of `pip` to manage your Python packages, you can also install Coconut using `conda`. Just install `conda`, open up a command-line prompt, and enter

```
conda config --add channels conda-forge
conda install coconut
```

which will properly create and build a `conda` recipe out of Coconut's `conda-forge` feedstock.

Note: To use `conda` to install `coconut-develop` instead, just replace `coconut` with `coconut-develop` in the last three commands above.

Optional Dependencies

Coconut also has optional dependencies, which can be installed by entering

```
pip install coconut[name_of_optional_dependency]
```

or, to install multiple optional dependencies,

```
pip install coocnut[opt_dep_1,opt_dep_2]
```

The full list of optional dependencies is:

- `all`: alias for `jupyter`, `watch`, `jobs`, `mypy` (this is the recommended way to install a feature-complete version of Coconut),
- `jupyter`/`ipython`: enables use of the `--jupyter` / `--ipython` flag,
- `watch`: enables use of the `--watch` flag,
- `jobs`: improves use of the `--jobs` flag,
- `mypy`: enables use of the `--mypy` flag,
- `cPyparsing`: significantly speeds up compilation if your platform supports it by making use of `cPyparsing`,
- `tests`: everything necessary to run Coconut's test suite,
- `docs`: everything necessary to build Coconut's documentation, and
- `dev`: everything necessary to develop on Coconut, including all of the dependencies above.

Develop Version

Alternatively, if you want to test out Coconut's latest and greatest, enter

```
pip install coconut-develop
```

which will install the most recent working version from Coconut's `develop` branch. Optional dependency installation is supported in the same manner as above. For more information on the current development build, check out the development version of this documentation. Be warned: `coconut-develop` is likely to be unstable—if you find a bug, please report it by creating a new issue.

3.2.2 Usage

```
coconut [-h] [-v] [-t version] [-i] [-p] [-a] [-l] [-k] [-w] [-r] [-n]
        [-d] [-q] [-s] [--no-tco] [-c code] [-j processes] [-f]
        [--minify] [--jupyter ...] [--mypy ...] [--argv ...]
        [--tutorial] [--documentation] [--style name]
        [--recursion-limit limit] [--verbose] [--trace]
        [source] [dest]
```

Positional Arguments

```
source                path to the Coconut file/folder to compile
dest                  destination directory for compiled files (defaults to
                      the source directory)
```

Optional Arguments

```
-h, --help            show this help message and exit
-v, --version         print Coconut and Python version information
-t version, --target version
                      specify target Python version (defaults to universal)
-i, --interact        force the interpreter to start (otherwise starts if no
                      other command is given) (implies --run)
-p, --package         compile source as part of a package (defaults to only
                      if source is a directory)
-a, --standalone      compile source as standalone files (defaults to only
                      if source is a single file)
-l, --line-numbers, --linenumbers
                      add line number comments for ease of debugging
-k, --keep-lines, --keeplines
                      include source code in comments for ease of debugging
-w, --watch           watch a directory and recompile on changes
-r, --run             execute compiled Python
-n, --no-write, --nowrite
                      disable writing compiled Python
-d, --display         print compiled Python
-q, --quiet           suppress all informational output (combine with
                      --display to write runnable code to stdout)
-s, --strict          enforce code cleanliness standards
--no-tco, --notco     disable tail call optimization
-c code, --code code  run Coconut passed in as a string (can also be piped
                      into stdin)
```

```
-j processes, --jobs processes
                      number of additional processes to use (defaults to 0)
                      (pass 'sys' to use machine default)
-f, --force           force overwriting of compiled Python (otherwise only
                      overwrites when source code or compilation parameters
                      change)
--minify              reduce size of compiled Python
--jupyter ..., --ipython ...
                      run Jupyter/IPython with Coconut as the kernel
                      (remaining args passed to Jupyter)
--mypy ...            run MyPy on compiled Python (remaining args passed to
                      MyPy) (implies --package --no-tco)
--argv ...            set sys.argv to source plus remaining args for use in
                      Coconut script being run
--tutorial            open Coconut's tutorial in the default web browser
--documentation       open Coconut's documentation in the default web
                      browser
--style name          Pygments syntax highlighting style (or 'none' to
                      disable) (defaults to COCONUT_STYLE environment
                      variable, if it exists, otherwise 'default')
--recursion-limit limit, --recursionlimit limit
                      set maximum recursion depth in compiler (defaults to
                      2000)
--verbose             print verbose debug output
--trace               print verbose parsing data (only available in coconut-
                      develop)
```

3.2.3 Coconut Scripts

To run a Coconut file as a script, Coconut provides the command

```
coconut-run <source> <args>
```

as an alias for

```
coconut --run --quiet --target sys <source> --argv <args>
```

which will quietly compile and run `<source>`, passing any additional arguments to the script, mimicking how the `python` command works.

`coconut-run` can be used in a Unix shebang line to create a Coconut script by adding the following line to the start of your script:

```
#!/usr/bin/env coconut-run
```

3.2.4 Naming Source Files

Coconut source files should, so the compiler can recognize them, use the extension `.coco` (preferred), `.coc`, or `.coconut`. When Coconut compiles a `.coco` (or `.coc`/`.coconut`) file, it will compile to another file with the same name, except with `.py` instead of `.coco`, which will hold the compiled code. If an extension other than `.py` is desired for the compiled files, such as `.pyde` for Python Processing, then that extension can be put before `.coco` in the source file name, and it will be used instead of `.py` for the compiled files. For example, `name.coco` will compile to `name.py`, whereas `name.pyde.coco` will compile to `name.pyde`.

3.2.5 Compilation Modes

Files compiled by the `coconut` command-line utility will vary based on compilation parameters. If an entire directory of files is compiled (which the compiler will search recursively for any folders containing `.coco`, `.coc`, or `.coconut` files), a `__coconut__.py` file will be created to house necessary functions (package mode), whereas if only a single file is compiled, that information will be stored within a header inside the file (standalone mode). Standalone mode is better for single files because it gets rid of the overhead involved in importing `__coconut__.py`, but package mode is better for large packages because it gets rid of the need to run the same Coconut header code again in every file, since it can just be imported from `__coconut__.py`.

By default, if the `source` argument to the command-line utility is a file, it will perform standalone compilation on it, whereas if it is a directory, it will recursively search for all `.coco` (or `.coc` / `.coconut`) files and perform package compilation on them. Thus, in most cases, the mode chosen by Coconut automatically will be the right one. But if it is very important that no additional files like `__coconut__.py` be created, for example, then the command-line utility can also be forced to use a specific mode with the `--package` (`-p`) and `--standalone` (`-a`) flags.

3.2.6 Compatible Python Versions

While Coconut syntax is based off of Python 3, Coconut code compiled in universal mode (the default `--target`), and the Coconut compiler, should run on any Python version `>= 2.6` on the `2.x` branch or `>= 3.2` on the `3.x` branch.

Note: The tested against implementations are CPython `2.6`, `2.7`, `3.2`, `3.3`, `3.4`, `3.5`, `3.6` and PyPy `2.7`, `3.2`.

To make Coconut built-ins universal across Python versions, **Coconut automatically overwrites Python 2 built-ins with their Python 3 counterparts**. Additionally, Coconut also overwrites some Python 3 built-ins for optimization and enhancement purposes. If access to the original Python versions of any overwritten built-ins is desired, the old built-ins can be retrieved by prefixing them with `py_`.

For standard library compatibility, **Coconut automatically maps imports under Python 3 names to imports under Python 2 names**. Thus, Coconut will automatically take care of any standard library modules that were renamed from Python 2 to Python 3 if just the Python 3 name is used. For modules or objects that only exist in Python 3, however, Coconut has no way of maintaining compatibility.

Finally, while Coconut will try to compile Python-3-specific syntax to its universal equivalent, the following constructs have no equivalent in Python 2, and require the specification of a target of at least `3` to be used:

- destructuring assignment with `*`s (use Coconut pattern-matching instead),
- the `nonlocal` keyword,
- `exec` used in a context where it must be a function,
- keyword class definition,
- tuples and lists with `*` unpacking or dicts with `**` unpacking (requires `--target 3.5`),
- `@` as matrix multiplication (requires `--target 3.5`),
- `async` and `await` statements (requires `--target 3.5`), and
- formatting strings by prefixing them with `f` (requires `--target 3.6`).

3.2.7 Allowable Targets

If the version of Python that the compiled code will be running on is known ahead of time, a target should be specified with `--target`. The given target will only affect the compiled code and whether or not the Python-3-specific syntax

detailed above is allowed. Where Python 3 and Python 2 syntax standards differ, Coconut syntax will always follow Python 3 across all targets. The supported targets are:

- universal (default) (will work on *any* of the below),
- `2`, `2.6` (will work on any Python `>= 2.6` but `< 3`),
- `2.7` (will work on any Python `>= 2.7` but `< 3`),
- `3`, `3.2` (will work on any Python `>= 3.2`),
- `3.3`, `3.4` (will work on any Python `>= 3.3`),
- `3.5` (will work on any Python `>= 3.5`),
- `3.6` (will work on any Python `>= 3.6`),
- `sys` (chooses the specific target corresponding to the current version).

Note: Periods are ignored in target specifications, such that the target `27` is equivalent to the target `2.7`.

3.2.8 `strict` Mode

If the `--strict` (or `-s`) flag is enabled, Coconut will throw errors on various style problems. These are

- mixing of tabs and spaces (without `--strict` will show a Warning),
- use of `from __future__` imports (without `--strict` will show a Warning)
- missing new line at end of file,
- trailing whitespace at end of lines,
- semicolons at end of lines,
- use of the Python-style `lambda` statement,
- inheriting from `object` in classes (Coconut does this automatically),
- use of `u` to denote Unicode strings (all Coconut strings are Unicode strings), and
- use of backslash continuations (use *parenthetical continuation* instead).

Additionally, `--strict` disables deprecated features, making them entirely unavailable to code compiled with `--strict`. It is recommended that you use the `--strict` (or `-s`) flag if you are starting a new Coconut project, as it will help you write cleaner code.

3.3 Integrations

3.3.1 Syntax Highlighting

Text editors with support for Coconut syntax highlighting are:

- **SublimeText**: See SublimeText section below.
- **Vim**: See `coconut.vim`.
- **Emacs**: See `coconut-mode`.
- **Atom**: See `language-coconut`.
- Any editor that supports **Pygments**: See Pygments section below.

Alternatively, if none of the above work for you, you can just treat Coconut as Python. Simply set up your editor so it interprets all `.coco` files as Python and that should highlight most of your code well enough.

SublimeText

Coconut syntax highlighting for SublimeText requires that Package Control, the standard package manager for SublimeText, be installed. Once that is done, simply:

1. open the SublimeText command palette by pressing `Ctrl+Shift+P` (or `Cmd+Shift+P` on Mac),
2. type and enter `Package Control: Install Package`, and
3. finally type and enter `Coconut`.

To make sure everything is working properly, open a `.coco` file, and make sure `Coconut` appears in the bottom right-hand corner. If something else appears, like `Plain Text`, click on it, select `Open all with current extension as...` at the top of the resulting menu, and then select `Coconut`.

Note: Coconut syntax highlighting for SublimeText is provided by the sublime-coconut package.

Pygments

The same `pip install coconut` command that installs the Coconut command-line utility will also install the `coconut` Pygments lexer. How to use this lexer depends on the Pygments-enabled application being used, but in general simply enter `coconut` as the language being highlighted and/or use a valid Coconut file extension (`.coco`, `.coc`, or `.coconut`) and Pygments should be able to figure it out. For example, this documentation is generated with Sphinx, with the syntax highlighting you see created by adding the line

```
highlight_language = "coconut"
```

to Coconut's `conf.py`.

3.3.2 IPython/Jupyter Support

If you prefer IPython (the python kernel for the Jupyter framework) to the normal Python shell, Coconut can be used as a Jupyter kernel or IPython extension.

Kernel

If Coconut is used as a kernel, all code in the console or notebook will be sent directly to Coconut instead of Python to be evaluated. Otherwise, the Coconut kernel behaves exactly like the IPython kernel, including support for `%magic` commands.

The command `coconut --jupyter notebook` (or `coconut --ipython notebook`) will launch an IPython/Jupyter notebook using Coconut as the kernel and the command `coconut --jupyter console` (or `coconut --ipython console`) will launch an IPython/Jupyter console using Coconut as the kernel. Additionally, the command `coconut --jupyter` (or `coconut --ipython`) will add Coconut as a language option inside of all IPython/Jupyter notebooks, even those not launched with Coconut. This command may need to be re-run when a new version of Coconut is installed.

Extension

If Coconut is used as an extension, a special magic command will send snippets of code to be evaluated using Coconut instead of IPython, but IPython will still be used as the default.

The line magic `%load_ext coconut` will load Coconut as an extension, providing the `%coconut` and `%%coconut` magics and adding Coconut built-ins. The `%coconut` line magic will run a line of Coconut with default parameters, and the `%%coconut` block magic will take command-line arguments on the first line, and run any Coconut code provided in the rest of the cell with those parameters.

3.3.3 MyPy Integration

Coconut has the ability to integrate with MyPy to provide optional static type-checking, including for all Coconut built-ins. Simply pass `--mypy` to enable MyPy integration, though be careful to pass it only as the last argument, since all arguments after `--mypy` are passed to `mypy`, not Coconut.

Note: Since tail call optimization prevents proper type-checking, `--mypy` implicitly disables it.

To explicitly annotate your code with types for MyPy to check, Coconut supports Python 3 function type annotations, Python 3.6 variable type annotations, and even Coconut's own *enhanced type annotation syntax*. By default, all type annotations are compiled to Python-2-compatible type comments, which means all of the above works on any Python version.

Coconut even supports `--mypy` in the interpreter, which will intelligently scan each new line of code, in the context of previous lines, for newly-introduced MyPy errors. For example:

```
>>> a: str = count()[0]
<string>:14: error: Incompatible types in assignment (expression has type "int",␣
↪variable has type "str")
```

Note: Sometimes, MyPy will not know how to handle certain Coconut constructs, such as `addpattern`. In that case, simply put a `# type: ignore` comment on the Coconut line which is generating the line MyPy is complaining about (you can figure out what line this is using `--line-numbers`) and the comment will be added to every generated line.

3.4 Operators

In order of precedence, highest first, the operators supported in Coconut are:

```
==================== ==========================
Symbol(s)            Associativity
==================== ==========================
..                   n/a won't capture call
**                   right
+, -, ~              unary
*, /, //, %, @       left
+, -                 left
<<, >>               left
&                    left
^                    left
|                    left
::                   n/a lazy
a `b` c              left captures lambda
??                   left short-circuit
..>, <.., ..*>, <*.. n/a captures lambda
```

```
|>, <|, |*>, <*|       left captures lambda
==, !=, <, >,
  <=, >=,
  in, not in,
  is, is not          n/a
not                   unary
and                   left short-circuit
or                    left short-circuit
a if b else c         ternary left short-circuit
->                    right
===================== ========================
```

3.4.1 Lambdas

Coconut provides the simple, clean `->` operator as an alternative to Python's `lambda` statements. The syntax for the `->` operator is `(parameters) -> expression` (or `parameter -> expression` for one-argument lambdas). The operator has the same precedence as the old statement, which means it will often be necessary to surround the lambda in parentheses, and is right-associative.

Additionally, Coconut also supports an implicit usage of the `->` operator of the form `(-> expression)`, which is equivalent to `((_=None) -> expression)`, which allows an implicit lambda to be used both when no arguments are required, and when one argument (assigned to `_`) is required.

Note: If normal lambda syntax is insufficient, Coconut also supports an extended lambda syntax in the form of statement lambdas.

Rationale

In Python, lambdas are ugly and bulky, requiring the entire word `lambda` to be written out every time one is constructed. This is fine if in-line functions are very rarely needed, but in functional programming in-line functions are an essential tool.

Python Docs

Lambda forms (lambda expressions) have the same syntactic position as expressions. They are a shorthand to create anonymous functions; the expression `(arguments) -> expression` yields a function object. The unnamed object behaves like a function object defined with:

```
def <lambda>(arguments):
    return expression
```

Note that functions created with lambda forms cannot contain statements or annotations.

Example

Coconut:

```
dubsums = map((x, y) -> 2*(x+y), range(0, 10), range(10, 20))
dubsums |> list |> print
```

Python:

```
dubsums = map(lambda x, y: 2*(x+y), range(0, 10), range(10, 20))
print(list(dubsums))
```

3.4.2 Partial Application

Coconut uses a `$` sign right after a function's name but before the open parenthesis used to call the function to denote partial application.

Coconut's partial application also supports the use of a `?` to skip partially applying an argument, deferring filling in that argument until the partially-applied function is called. This is useful if you want to partially apply arguments that aren't first in the argument order.

Rationale

Partial application, or currying, is a mainstay of functional programming, and for good reason: it allows the dynamic customization of functions to fit the needs of where they are being used. Partial application allows a new function to be created out of an old function with some of its arguments pre-specified.

Python Docs

Return a new `partial` object which when called will behave like *func* called with the positional arguments *args* and keyword arguments *keywords*. If more arguments are supplied to the call, they are appended to *args*. If additional keyword arguments are supplied, they extend and override *keywords*. Roughly equivalent to:

```
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*(args + fargs), **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

The `partial` object is used for partial function application which "freezes" some portion of a function's arguments and/or keywords resulting in a new object with a simplified signature.

Example

Coconut:

```
expnums = range(5) |> map$(pow$(?, 2))
expnums |> list |> print
```

Python:

```
# unlike this simple lambda, $ produces a pickleable object
expnums = map(lambda x: pow(x, 2), range(5))
print(list(expnums))
```

3.4.3 Pipeline

Coconut uses pipe operators for pipeline-style function application. All the operators have a precedence in-between function composition pipes and comparisons, and are left-associative. All operators also support in-place versions. The different operators are:

```
(|>)    => pipe forward
(|*>)   => multiple-argument pipe forward
(<|)    => pipe backward
(<*|)   => multiple-argument pipe backward
```

Additionally, all pipe operators support a lambda as the last argument, despite lambdas having a lower precedence. Thus, `10 |> x -> x**2` is valid, though the body of the lambda will still capture all following pipe operators.

Optimizations

It is common in Coconut to write code that uses pipes to pass an object through a series of *partials* and/or *implicit partials*, as in

```
obj |> .attribute |> .method(args) |> func$(args) |> .[index]
```

which is often much more readable, as it allows the operations to be written in the order in which they are performed, instead of as in

```
func(args, obj.attribute.method(args))[index]
```

where `func` has to go at the beginning.

If Coconut compiled each of the partials in the pipe syntax as an actual partial application object, it would make the Coconut-style syntax here so much slower than the Python-style syntax as to make it near-useless. Thus, Coconut does not do that. If any of the above styles of partials or implicit partials are used in pipes, they will whenever possible be compiled to the Python-style syntax, producing no intermediate partial application objects.

Example

Coconut:

```
def sq(x) = x**2
(1, 2) |*> (+) |> sq |> print
```

Python:

```
import operator
def sq(x): return x**2
print(sq(operator.add(1, 2)))
```

3.4.4 Compose

Coconut has three basic function composition operators: `..`, `..>`, and `<..` Both `..` and `<..` use math-style "backwards" function composition, where the first function is called last, while `..>` uses "forwards" function composition, where the first function is called first.

The `..>` and `<..` function composition pipe operators also have `..*>` and `<*..` forms which are, respectively, the equivalents of `|*>` and `<*|`. Forwards and backwards function composition pipes cannot be used together in the same expression (unlike normal pipes) and have precedence in-between `None`-coalescing and normal pipes.

The `..` operator has lower precedence than attribute access (`.`), slicing (`[]`), etc., except for function calling, which it has higher precedence than. Thus, `a.b..c.d` is equivalent to `(a.b)..(c.d)`, while `f..g(x)` is equivalent to `(f..g)(x)`.

The in-place function composition operators are `..=`, `..>=`, `<..=`, `..*>=`, and `<*..=`.

Example

Coconut:

```
fog = f..g
f_into_g = f ..> g
```

Python:

```
# unlike these simple lambdas, Coconut produces pickleable objects
fog = lambda *args, **kwargs: f(g(*args, **kwargs))
f_into_g = lambda *args, **kwargs: g(f(*args, **kwargs))
```

3.4.5 Chain

Coconut uses the `::` operator for iterator chaining. Coconut's iterator chaining is done lazily, in that the arguments are not evaluated until they are needed. It has a precedence in-between bitwise or and infix calls. The in-place operator is `::=`.

Rationale

A useful tool to make working with iterators as easy as working with sequences is the ability to lazily combine multiple iterators together. This operation is called chain, and is equivalent to addition with sequences, except that nothing gets evaluated until it is needed.

Python Docs

Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. Used for treating consecutive sequences as a single sequence. Chained inputs are evaluated lazily. Roughly equivalent to:

```
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

Example

Coconut:

```
def N(n=0) = (n,) :: N(n+1)  # no infinite loop because :: is lazy

(range(-10, 0) :: N())$[5:15] |> list |> print
```

Python: *Can't be done without a complicated iterator comprehension in place of the lazy chaining. See the compiled code for the Python syntax.*

3.4.6 Iterator Slicing

Coconut uses a `$` sign right after an iterator before a slice to perform iterator slicing. Coconut's iterator slicing works much the same as Python's sequence slicing, and looks much the same as Coconut's partial application, but with brackets instead of parentheses.

Iterator slicing works just like sequence slicing, including support for negative indices and slices, and support for `slice` objects in the same way as can be done with normal slicing. Iterator slicing makes no guarantee, however, that the original iterator passed to it be preserved (to preserve the iterator, use Coconut's *`tee`* or *`reiterable`* built-in).

Coconut's iterator slicing is very similar to Python's `itertools.islice`, but unlike `itertools.islice`, Coconut's iterator slicing supports negative indices, and will preferentially call an object's `__getitem__`, if it exists. Coconut's iterator slicing is also optimized to work well with all of Coconut's built-in objects, only computing the elements of each that are actually necessary to extract the desired slice.

Example

Coconut:

```
map(x -> x*2, range(10**100))$[-1] |> print
```

Python: *Can't be done without a complicated iterator slicing function and inspection of custom objects. The necessary definitions in Python can be found in the Coconut header.*

3.4.7 None Coalescing

Coconut provides `??` as a `None`-coalescing operator, similar to the `??` null-coalescing operator in C#. The `None`-coalescing operator evaluates to its left operand if that operand is not `None`, otherwise its right operand. The `None`-coalescing operator is short-circuiting, such that if the left operand is not `None`, it will not evaluate the right operand. The `None`-coalescing operator has a precedence in-between infix function calls and composition pipes, and is left-associative. The in-place operator is `??=`.

Coconut also allows a single `?` before attribute access, function calling, partial application, and (iterator) indexing to short-circuit the rest of the evaluation if everything so far evaluates to `None`. Thus, `a?.b` is equivalent to `a.b if a is not None else a`.

Example

Coconut:

```
could_be_none() ?? calculate_default_value()
could_be_none()?.attr[index].method()
```

Python:

```
(lambda result: result if result is not None else calculate_default_value())(could_be_
↪none())
(lambda result: None if result is None else result.attr[index].method())(could_be_
↪none())
```

3.4.8 Unicode Alternatives

Coconut supports Unicode alternatives to many different operator symbols. The Unicode alternatives are relatively straightforward, and chosen to reflect the look and/or meaning of the original symbol.

Full List

```
→ (\u2192)                    => "->"
 (\u21a6)                    => "|>"
⋆ (⋆\u21a6)                  => "|⋆>"
 (\u21a4)                    => "<|"
⋆ (\u21a4⋆)                  => "<⋆|"
 (\u22c5)                    => "⋆"
↑ (\u2191)                    => "**"
÷ (\xf7)                      => "/"
÷/ (\xf7/)                    => "//"
 (\u2218)                    => ".."
> (\u2218>)                   => "..>"
< (<\u2218)                   => "<.."
⋆> (\u2218⋆>)                 => "..⋆>"
<⋆ (<⋆\u2218)                 => "<⋆.."
 (\u2212)                    => "-" (only subtraction)
 (\u207b)                    => "-" (only negation)
¬ (\xac)                      => "~"
 (\u2260) or ¬= (\xac=)      => "!="
 (\u2264)                    => "<="
 (\u2265)                    => ">="
 (\u2227) or  (\u2229)      => "&"
 (\u2228) or  (\u222a)      => "|"
 (\u22bb) or  (\u2295)      => "^"
« (\xab)                      => "<<"
» (\xbb)                      => ">>"
... (\u2026)                    => "..."
× (\xd7)                      => "@" (only matrix multiplication)
```

3.5 Keywords

3.5.1 `data`

Coconut's `data` keyword is used to create immutable, algebraic data types with built-in support for destructuring *pattern-matching* and *`fmap`*.

The syntax for `data` blocks is a cross between the syntax for functions and the syntax for classes. The first line looks like a function definition, but the rest of the body looks like a class, usually containing method definitions. This is because while `data` blocks actually end up as classes in Python, Coconut automatically creates a special, immutable constructor based on the given arguments.

Coconut data statement syntax looks like:

```
data <name>(<args>) [from <inherits>]:
    <body>
```

<name> is the name of the new data type, <args> are the arguments to its constructor as well as the names of its attributes, <body> contains the data type's methods, and <inherits> optionally contains any desired base classes.

Coconut allows data fields in <args> to have defaults and/or *type annotations* attached to them, and supports a starred parameter at the end to collect extra arguments.

Writing constructors for data types must be done using the __new__ method instead of the __init__ method. For helping to easily write __new__ methods, Coconut provides the *makedata* built-in.

Subclassing data types can be done easily by inheriting from them either in another data statement or a normal Python class. If a normal class statement is used, making the new subclass immutable will require adding the line

```
__slots__ = ()
```

which will need to be put in the subclass body before any method or attribute definitions.

Rationale

A mainstay of functional programming that Coconut improves in Python is the use of values, or immutable data types. Immutable data can be very useful because it guarantees that once you have some data it won't change, but in Python creating custom immutable data types is difficult. Coconut makes it very easy by providing data blocks.

Python Docs

Returns a new tuple subclass. The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being indexable and iterable. Instances of the subclass also have a helpful docstring (with type names and field names) and a helpful __repr__() method which lists the tuple contents in a name=value format.

Any valid Python identifier may be used for a field name except for names starting with an underscore. Valid identifiers consist of letters, digits, and underscores but do not start with a digit or underscore and cannot be a keyword such as *class, for, return, global, pass, or raise*.

Named tuple instances do not have per-instance dictionaries, so they are lightweight and require no more memory than regular tuples.

Examples

Coconut:

```
data vector2(x:int=0, y:int=0):
    def __abs__(self):
        return (self.x**2 + self.y**2)**.5

v = vector2(3, 4)
v |> print  # all data types come with a built-in __repr__
v |> abs |> print
v.x = 2  # this will fail because data objects are immutable
vector2() |> print
```

Showcases the syntax, features, and immutable nature of `data` types, as well as the use of default arguments and type annotations.

```
data Empty()
data Leaf(n)
data Node(l, r)

def size(Empty()) = 0

@addpattern(size)
def size(Leaf(n)) = 1

@addpattern(size)
def size(Node(l, r)) = size(l) + size(r)

size(Node(Empty(), Leaf(10))) == 1
```

Showcases the algebraic nature of `data` types when combined with pattern-matching.

```
data vector(*pts):
    """Immutable arbitrary-length vector."""

    def __abs__(self) =
        self.pts |> map$(pow$(?, 2)) |> sum |> pow$(?, 0.5)

    def __add__(self, other) =
        vector(*other_pts) = other
        assert len(other_pts) == len(self.pts)
        map((+), self.pts, other_pts) |*> vector

    def __neg__(self) =
        self.pts |> map$((-)) |*> vector

    def __sub__(self, other) =
        self + -other
```

Showcases starred `data` declaration.

Python:

```
import typing
class vector2(typing.NamedTuple("vector2", [("x", int), ("y", int)]), object):
    __slots__ = ()
    def __new__(cls, x=0, y=0):
        return super(vector2, cls).__new__((x, y))
    def __abs__(self):
        return (self.x**2 + self.y**2)**.5

v = vector2(3, 4)
print(v)
print(abs(v))
v.x = 2
```

```
import collections
class Empty(collections.namedtuple("Empty", ""), object):
    __slots__ = ()
class Leaf(collections.namedtuple("Leaf", "n"), object):
    __slots__ = ()
class Node(collections.namedtuple("Node", "l, r"), object):
```

```
    __slots__ = ()

def size(tree):
    if isinstance(tree, Empty):
        return 0
    elif isinstance(tree, Leaf):
        return 1
    elif isinstance(tree, Node):
        return size(tree[0]) + size(tree[1])
    else:
        raise MatchError()

size(Node(Empty(), Leaf(10))) == 1
```

Starred data declarations can't be done without a long sequence of method definitions. See the compiled code for the Python syntax.

3.5.2 `match`

Coconut provides fully-featured, functional pattern-matching through its `match` statements.

Overview

Match statements follow the basic syntax `match <pattern> in <value>`. The match statement will attempt to match the value against the pattern, and if successful, bind any variables in the pattern to whatever is in the same position in the value, and execute the code below the match statement. Match statements also support, in their basic syntax, an `if <cond>` that will check the condition after executing the match before executing the code below, and an `else` statement afterwards that will only be executed if the `match` statement is not. What is allowed in the match statement's pattern has no equivalent in Python, and thus the specifications below are provided to explain it.

Syntax Specification

Coconut match statement syntax is

```
match <pattern> in <value> [if <cond>]:
    <body>
[else:
    <body>]
```

where `<value>` is the item to match against, `<cond>` is an optional additional check, and `<body>` is simply code that is executed if the header above it succeeds. `<pattern>` follows its own, special syntax, defined roughly like so:

```
pattern ::= (
    "(" pattern ")"                 # parentheses
    | "None" | "True" | "False"     # constants
    | "=" NAME                      # check
    | NUMBER                        # numbers
    | STRING                        # stringsh
    | [pattern "as"] NAME           # capture
    | NAME "(" patterns ")"         # data types
    | pattern "is" exprs            # type-checking
    | pattern "and" pattern         # match all
    | pattern "or" pattern          # match any
    | "{" pattern_pairs             # dictionaries
```

```
        ["," ** NAME] "}"
    | ["s"] "{" pattern_consts "}"  # sets
    | "(" patterns ")"              # sequences can be in tuple form
    | "[" patterns "]"              #  or in list form
    | "(|" patterns "|)"            # lazy lists
    | ("(" | "[")                   # star splits
        patterns
        "*" middle
        patterns
      (")" | "]")
    | (                             # head-tail splits
        "(" patterns ")"
      | "[" patterns "]"
      ) "+" pattern
    | pattern "+" (                 # init-last splits
        "(" patterns ")"
      | "[" patterns "]"
      )
    | (                             # head-last splits
        "(" patterns ")"
      | "[" patterns "]"
      ) "+" pattern "+" (
        "(" patterns ")"                # this match must be the same
      | "[" patterns "]"                #  construct as the first match
      )
    | (                             # iterator splits
        "(" patterns ")"
      | "[" patterns "]"
      | "(|" patterns "|)"
      ) "::" pattern
    | ([STRING "+"] NAME            # complex string matching
        ["+" STRING])
)
```

Semantics Specification

`match` statements will take their pattern and attempt to "match" against it, performing the checks and deconstructions on the arguments as specified by the pattern. The different constructs that can be specified in a pattern, and their function, are:

- Constants, Numbers, and Strings: will only match to the same constant, number, or string in the same position in the arguments.
- Variables: will match to anything, and will be bound to whatever they match to, with some exceptions:
  - If the same variable is used multiple times, a check will be performed that each use match to the same value.
  - If the variable name _ is used, nothing will be bound and everything will always match to it.
- Explicit Bindings (`<pattern> as <var>`): will bind `<var>` to `<pattern>`.
- Checks (`=<var>`): will check that whatever is in that position is equal to the previously defined variable `<var>`.
- Type Checks (`<var> is <types>`): will check that whatever is in that position is of type(s) `<types>` before binding the `<var>`.
- Data Types (`<name>(<args>)`): will check that whatever is in that position is of data type `<name>` and will match the attributes to `<args>`.

- Lists (`[<patterns>]`), Tuples (`(<patterns>)`): will only match a sequence (`collections.abc.Sequence`) of the same length, and will check the contents against `<patterns>`.
- Lazy lists (`(|<patterns>|)`): same as list or tuple matching, but checks iterable (`collections.abc.Iterable`) instead of sequence.
- Fixed-Length Dicts (`{<pairs>}`): will only match a mapping (`collections.abc.Mapping`) of the same length, and will check the contents against `<pairs>`.
- Dicts With Rest (`{<pairs>, **<rest>}`): will match a mapping (`collections.abc.Mapping`) containing all the `<pairs>`, and will put a `dict` of everything else into `<rest>`.
- Sets (`{<constants>}`): will only match a set (`collections.abc.Set`) of the same length and contents.
- Head-Tail Splits (`<list/tuple> + <var>`): will match the beginning of the sequence against the `<list/tuple>`, then bind the rest to `<var>`, and make it the type of the construct used.
- Init-Last Splits (`<var> + <list/tuple>`): exactly the same as head-tail splits, but on the end instead of the beginning of the sequence.
- Head-Last Splits (`<list/tuple> + <var> + <list/tuple>`): the combination of a head-tail and an init-last split.
- Iterator Splits (`<list/tuple/lazy list> :: <var>`): will match the beginning of an iterable (`collections.abc.Iterable`) against the `<list/tuple/lazy list>`, then bind the rest to `<var>` or check that the iterable is done.
- Complex String Matching (`<string> + <var> + <string>`): matches strings that start and end with the given substrings, binding the middle to `<var>`.

Note: Like iterator slicing*, iterator and lazy list matching make no guarantee that the original iterator matched against be preserved (to preserve the iterator, use Coconut's* `tee` *or* `reiterable` *built-in).*

When checking whether or not an object can be matched against in a particular fashion, Coconut makes use of Python's abstract base classes. Therefore, to enable proper matching for a custom object, register it with the proper abstract base classes.

Examples

Coconut:

```
def factorial(value):
    match 0 in value:
        return 1
    else: match n is int in value if n > 0:  # possible because of Coconut's
        return n * factorial(n-1)            #   enhanced else statements
    else:
        raise TypeError("invalid argument to factorial of: "+repr(value))

3 |> factorial |> print
```

Showcases `else` *statements, which work much like* `else` *statements in Python: the code under an* `else` *statement is only executed if the corresponding match fails.*

```
data point(x, y):
    def transform(self, other):
        match point(x, y) in other:
            return point(self.x + x, self.y + y)
        else:
            raise TypeError("arg to transform must be a point")
```

```
    def __eq__(self, other):
        match point(=self.x, =self.y) in other:
            return True
        else:
            return False

point(1,2) |> point(3,4).transform |> print
point(1,2) |> (==)$(point(1,2)) |> print
```

Showcases matching to data types. Values defined by the user with the `data` *statement can be matched against and their contents accessed by specifically referencing arguments to the data type's constructor.*

```
data Empty()
data Leaf(n)
data Node(l, r)
Tree = (Empty, Leaf, Node)  # type union

def depth(Tree()) = 0

@addpattern(depth)
def depth(Tree(n)) = 1

@addpattern(depth)
def depth(Tree(l, r)) = 1 + max([depth(l), depth(r)])

Empty() |> depth |> print
Leaf(5) |> depth |> print
Node(Leaf(2), Node(Empty(), Leaf(3))) |> depth |> print
```

Showcases how the combination of data types and match statements can be used to powerful effect to replicate the usage of algebraic data types in other functional programming languages.

```
def duplicate_first([x] + xs as l) =
    [x] + l

[1,2,3] |> duplicate_first |> print
```

Showcases head-tail splitting, one of the most common uses of pattern-matching, where a `+ <var>` *(or* `:: <var>` *for any iterable) at the end of a list or tuple literal can be used to match the rest of the sequence.*

```
def sieve([head] :: tail) =
    [head] :: sieve(n for n in tail if n % head)

@addpattern(sieve)
def sieve((||)) = []
```

Showcases how to match against iterators, namely that the empty iterator case (`(||)`*) must come last, otherwise that case will exhaust the whole iterator before any other pattern has a chance to match against it.*

Python: *Can't be done without a long series of checks for each* `match` *statement. See the compiled code for the Python syntax.*

3.5.3 case

Coconut's `case` statement is an extension of Coconut's `match` statement for performing multiple `match` statements against the same value, where only one of them should succeed. Unlike lone `match` statements, only one match

statement inside of a `case` block will ever succeed, and thus more general matches should be put below more specific ones.

Each pattern in a case block is checked until a match is found, and then the corresponding body is executed, and the case block terminated. The syntax for case blocks is

```
case <value>:
    match <pattern> [if <cond>]:
        <body>
    match <pattern> [if <cond>]:
        <body>
    ...
[else:
    <body>]
```

where `<pattern>` is any `match` pattern, `<value>` is the item to match against, `<cond>` is an optional additional check, and `<body>` is simply code that is executed if the header above it succeeds. Note the absence of an `in` in the `match` statements: that's because the `<value>` in `case <value>` is taking its place.

Example

Coconut:

```
def classify_sequence(value):
    out = ""        # unlike with normal matches, only one of the patterns
    case value:     #  will match, and out will only get appended to once
        match ():
            out += "empty"
        match (_,):
            out += "singleton"
        match (x,x):
            out += "duplicate pair of "+str(x)
        match (_,_):
            out += "pair"
        match _ is (tuple, list):
            out += "sequence"
    else:
        raise TypeError()
    return out

[] |> classify_sequence |> print
() |> classify_sequence |> print
[1] |> classify_sequence |> print
(1,1) |> classify_sequence |> print
(1,2) |> classify_sequence |> print
(1,1,1) |> classify_sequence |> print
```

Python: *Can't be done without a long series of checks for each `match` statement. See the compiled code for the Python syntax.*

3.5.4 Backslash-Escaping

In Coconut, the keywords `data`, `match`, `case`, `async` (keyword in Python 3.5), and `await` (keyword in Python 3.5) are also valid variable names. While Coconut can disambiguate these two use cases, when using one of these keywords as a variable name, a backslash is allowed in front to be explicit about using a keyword as a variable name.

Example

Coconut:

```
\data = 5
print(\data)
```

Python:

```
data = 5
print(data)
```

3.6 Expressions

3.6.1 Statement Lambdas

The statement lambda syntax is an extension of the *normal lambda syntax* to support statements, not just expressions.

The syntax for a statement lambda is

```
def (arguments) -> statement; statement; ...
```

where `arguments` can be standard function arguments or *pattern-matching function definition* arguments and `statement` can be an assignment statement or a keyword statement. If the last `statement` (not followed by a semicolon) is an `expression`, it will automatically be returned.

Statement lambdas also support implicit lambda syntax, where when the arguments are omitted, as in `def ->  _`, `def (_=None) -> _` is assumed.

Example

Coconut:

```
L |> map$(def (x) -> y = 1/x; y*(1 - y))
```

Python:

```
def _lambda(x):
    y = 1/x
    return y*(1 - y)
map(_lambda, L)
```

3.6.2 Lazy Lists

Coconut supports the creation of lazy lists, where the contents in the list will be treated as an iterator and not evaluated until they are needed. Lazy lists can be created in Coconut simply by simply surrounding a comma-seperated list of items with `(|` and `|)` (so-called "banana brackets") instead of `[` and `]` for a list or `(` and `)` for a tuple.

Lazy lists use the same machinery as iterator chaining to make themselves lazy, and thus the lazy list `(| x, y |)` is equivalent to the iterator chaining expression `(x,) :: (y,)`, although the lazy list won't construct the intermediate tuples.

Rationale

Lazy lists, where sequences are only evaluated when their contents are requested, are a mainstay of functional programming, allowing for dynamic evaluation of the list's contents.

Example

Coconut:

```
(| print("hello,"), print("world!") |) |> consume
```

Python: *Can't be done without a complicated iterator comprehension in place of the lazy list. See the compiled code for the Python syntax.*

3.6.3 Implicit Partial Application

Coconut supports a number of different syntactical aliases for common partial application use cases. These are:

```
.attr           =>      operator.attrgetter("attr")
.method(args)   =>      operator.methodcaller("method", args)
obj.            =>      getattr$(obj)
func$           =>      ($)$(func)
seq[]           =>      operator.getitem$(seq)
iter$[]         =>      # the equivalent of seq[] for iterators
.[a:b:c]        =>      operator.itemgetter(slice(a, b, c))
.$[a:b:c]       =>      # the equivalent of .[a:b:c] for iterators
```

Example

Coconut:

```
1 |> "123"[]
mod$ <| 5 <| 3
```

Python:

```
"123"[1]
mod(5, 3)
```

3.6.4 Enhanced Type Annotations

Since Coconut syntax is a superset of Python 3 syntax, it supports Python 3 function type annotation syntax and Python 3.6 variable type annotation syntax. By default, Coconut compiles all type annotations into Python-2-compatible type comments. If you want to keep the type annotations instead, simply pass a `--target` that supports them.

Note: When compiling type annotations to Python 3 syntax, Coconut will wrap every annotation in a string when in a position where Python would otherwise evaluate it (Python 3 function annotation), so that all type annotations are only ever evaluated at compile time, never at run time.

Additionally, Coconut adds special syntax for making type annotations easier and simpler to write. When inside of a type annotation, Coconut treats certain syntax constructs differently, compiling them to type annotations instead of what they would normally represent. Specifically, Coconut applies the following transformations

```
<type>?
    => typing.Optional[<type>]
<type>[]
    => typing.Sequence[<type>]
<type>$[]
    => typing.Iterable[<type>]
() -> <ret>
    => typing.Callable[[], <ret>]
<arg> -> <ret>
    => typing.Callable[[<arg>], <ret>]
(<args>) -> <ret>
    => typing.Callable[[<args>], <ret>]
-> <ret>
    => typing.Callable[..., <ret>]
```

where `typing` is the Python 3.5 built-in `typing` module.

Example

Coconut:

```
def int_map(
    f: int -> int,
    xs: int[],
) -> int[] =
    xs |> map$(f) |> list
```

Python:

```
import typing  # unlike this typing import, Coconut produces universal code
def int_map(
    f,  # type: typing.Callable[[int], int]
    xs,  # type: typing.Sequence[int]
):
    # type: (...) -> typing.Sequence[int]
    return list(map(f, xs))
```

3.6.5 Operator Functions

Coconut uses a simple operator function short-hand: surround an operator with parentheses to retrieve its function. Similarly to iterator comprehensions, if the operator function is the only argument to a function, the parentheses of the function call can also serve as the parentheses for the operator function.

Rationale

A very common thing to do in functional programming is to make use of function versions of built-in operators: currying them, composing them, and piping them. To make this easy, Coconut provides a short-hand syntax to access operator functions.

Full List

```
(|>)          => # pipe forward
(|*>)         => # multi-arg pipe forward
(<|)          => # pipe backward
(<*|)         => # multi-arg pipe backward
(..), (<..)   => # backward function composition
(..>)         => # forward function composition
(<*..)        => # multi-arg backward function composition
(..*>)        => # multi-arg forward function composition
(.)           => (getattr)
(::)          => (itertools.chain)  # will not evaluate its arguments lazily
($)           => (functools.partial)
($[])         => # iterator slicing operator
(+)           => (operator.add)
(-)           => # 1 arg: operator.neg, 2 args: operator.sub
(*)           => (operator.mul)
(**)          => (operator.pow)
(/)           => (operator.truediv)
(//)          => (operator.floordiv)
(%)           => (operator.mod)
(&)           => (operator.and_)
(^)           => (operator.xor)
(|)           => (operator.or_)
(<<)          => (operator.lshift)
(>>)          => (operator.rshift)
(<)           => (operator.lt)
(>)           => (operator.gt)
(==)          => (operator.eq)
(<=)          => (operator.le)
(>=)          => (operator.ge)
(!=)          => (operator.ne)
(~)           => (operator.inv)
(@)           => (operator.matmul)
(not)         => (operator.not_)
(and)         => # boolean and
(or)          => # boolean or
(is)          => (operator.is_)
(in)          => (operator.contains)
```

Example

Coconut:

```
(range(0, 5), range(5, 10)) |*> map$(+) |> list |> print
```

Python:

```
import operator
print(list(map(operator.add, range(0, 5), range(5, 10))))
```

3.6.6 Set Literals

Coconut allows an optional `s` to be prepended in front of Python set literals. While in most cases this does nothing, in the case of the empty set it lets Coconut know that it is an empty set and not an empty dictionary. Additionally, an `f`

is also supported, in which case a Python `frozenset` will be generated instead of a normal set.

Example

Coconut:

```
empty_frozen_set = f{}
```

Python:

```
empty_frozen_set = frozenset()
```

3.6.7 Imaginary Literals

In addition to Python's `<num>j` or `<num>J` notation for imaginary literals, Coconut also supports `<num>i` or `<num>I`, to make imaginary literals more readable if used in a mathematical context.

Python Docs

Imaginary literals are described by the following lexical definitions:

```
imagnumber ::= (floatnumber | intpart) ("j" | "J" | "i" | "I")
```

An imaginary literal yields a complex number with a real part of 0.0. Complex numbers are represented as a pair of floating point numbers and have the same restrictions on their range. To create a complex number with a nonzero real part, add a floating point number to it, e.g., `(3+4i)`. Some examples of imaginary literals:

```
3.14i   10.i    10i     .001i   1e100i  3.14e-10i
```

Example

Coconut:

```
3 + 4i |> abs |> print
```

Python:

```
print(abs(3 + 4j))
```

3.7 Function Definition

3.7.1 Tail Call Optimization

Coconut will perform automatic tail call optimization and tail recursion elimination on any function that meets the following criteria:

1. it must directly return (using either `return` or *assignment function notation*) a call to itself (tail recursion elimination, the most powerful optimization) or another function (tail call optimization).
2. it must not be a generator (uses `yield`) or an asynchronous function (uses `async`).

Note: Tail call optimization (though not tail recursion elimination) will work even for 1) mutual recursion and 2) pattern-matching functions split across multiple definitions using `addpattern`*.*

If you are encountering a `RuntimeError` due to maximum recursion depth, it is highly recommended that you rewrite your function to meet either the criteria above for tail call optimization, or the corresponding criteria for *`recursive_iterator`*, either of which should prevent such errors.

Note: Tail call optimization (though not tail recursion elimination) will be turned off if you pass the `--no-tco` *command-line option, which is useful if you are having trouble reading your tracebacks and/or need maximum performance.*

Example

Coconut:

```
# unlike in Python, this function will never hit a maximum recursion depth error
def factorial(n, acc=1):
    case n:
        match 0:
            return acc
        match _ is int if n > 0:
            return factorial(n-1, acc*n)
```

Showcases tail recursion elimination.

```
# unlike in Python, neither of these functions will ever hit a maximum recursion␣
↪depth error
def is_even(0) = True
@addpattern(is_even)
def is_even(n is int if n > 0) = is_odd(n-1)

def is_odd(0) = False
@addpattern(is_odd)
def is_odd(n is int if n > 0) = is_even(n-1)
```

Showcases tail call optimization.

Python: *Can't be done without rewriting the function(s).*

3.7.2 Assignment Functions

Coconut allows for assignment function definition that automatically returns the last line of the function body. An assignment function is constructed by substituting `=` for `:` after the function definition line. Thus, the syntax for assignment function definition is either

```
def <name>(<args>) = <expr>
```

for one-liners or

```
def <name>(<args>) =
    <stmts>
    <expr>
```

for full functions, where `<name>` is the name of the function, `<args>` are the functions arguments, `<stmts>` are any statements that the function should execute, and `<expr>` is the value that the function should return.

Note: Assignment function definition can be combined with infix and/or pattern-matching function definition.

Rationale

Coconut's Assignment function definition is as easy to write as assignment to a lambda, but will appear named in tracebacks, as it compiles to normal Python function definition.

Example

Coconut:

```
def binexp(x) = 2**x
5 |> binexp |> print
```

Python:

```
def binexp(x): return 2**x
print(binexp(5))
```

3.7.3 Pattern-Matching Functions

Coconut supports pattern-matching on the arguments to a function in that function's definition. The syntax for pattern-matching function definition is

```
[match] def <name>(<pattern> [= <default>], ... [if <cond>]):
    <body>
```

where `<name>` is the name of the function, `<cond>` is an optional additional check, `<body>` is the body of the function, `<pattern>` is defined by Coconut's *match statement*, and `<default>` is the optional default if no argument is passed. The `match` keyword at the beginning is optional, but is sometimes necessary to disambiguate pattern-matching function definition from normal function definition, which will always take precedence.

If `<pattern>` has a variable name (either directly or with `as`), the resulting pattern-matching function will support keyword arguments using that variable name. If pattern-matching function definition fails, it will raise a *MatchError* object just like *destructuring assignment*.

Note: Pattern-matching function definition can be combined with assignment and/or infix function definition.

Example

Coconut:

```
def last_two(_ + [a, b]):
    return a, b
def xydict_to_xytuple({"x":x is int, "y":y is int}):
    return x, y

range(5) |> last_two |> print
{"x":1, "y":2} |> xydict_to_xytuple |> print
```

Python: *Can't be done without a long series of checks at the top of the function. See the compiled code for the Python syntax.*

3.7.4 Infix Functions

Coconut allows for infix function calling, where an expression that evaluates to a function is surrounded by backticks and then can have arguments placed in front of or behind it. Infix calling has a precedence in-between chaining and `None`-coalescing, and is left-associative.

Coconut also supports infix function definition to make defining functions that are intended for infix usage simpler. The syntax for infix function definition is

```
def <arg> `<name>` <arg>:
    <body>
```

where `<name>` is the name of the function, the `<arg>`s are the function arguments, and `<body>` is the body of the function. If an `<arg>` includes a default, the `<arg>` must be surrounded in parentheses.

Note: Infix function definition can be combined with assignment and/or pattern-matching function definition.

Rationale

A common idiom in functional programming is to write functions that are intended to behave somewhat like operators, and to call and define them by placing them between their arguments. Coconut's infix syntax makes this possible.

Example

Coconut:

```
def a `mod` b = a % b
(x `mod` 2) `print`
```

Python:

```
def mod(a, b): return a % b
print(mod(x, 2))
```

3.7.5 Dotted Function Definition

Coconut allows for function definition using a dotted name to assign a function as a method of an object as specified in PEP 542.

Example

Coconut:

```
def MyClass.my_method(self):
    ...
```

Python:

```
def my_method(self):
    ...
MyClass.my_method = my_method
```

3.8 Statements

3.8.1 Destructuring Assignment

Coconut supports significantly enhanced destructuring assignment, similar to Python's tuple/list destructuring, but much more powerful. The syntax for Coconut's destructuring assignment is

```
[match] <pattern> = <value>
```

where `<value>` is any expression and `<pattern>` is defined by Coconut's *`match` statement*. The `match` keyword at the beginning is optional, but is sometimes necessary to disambiguate destructuring assignment from normal assignment, which will always take precedence. Coconut's destructuring assignment is equivalent to a match statement that follows the syntax:

```
match <pattern> in <value>:
    pass
else:
    err = MatchError(<error message>)
    err.pattern = "<pattern>"
    err.value = <value>
    raise err
```

If a destructuring assignment statement fails, then instead of continuing on as if a `match` block had failed, a *`MatchError`* object will be raised describing the failure.

Example

Coconut:

```
_ + [a, b] = [0, 1, 2, 3]
print(a, b)
```

Python: *Can't be done without a long series of checks in place of the destructuring assignment statement. See the compiled code for the Python syntax.*

3.8.2 Decorators

Unlike Python, which only supports a single variable or function call in a decorator, Coconut supports any expression.

Example

Coconut:

```
@ wrapper1 .. wrapper2 $(arg)
def func(x) = x**2
```

Python:

```
def wrapper(func):
    return wrapper1(wrapper2(arg, func))
@wrapper
def func(x):
    return x**2
```

3.8.3 `else` Statements

Coconut supports the compound statements `try`, `if`, and `match` on the end of an `else` statement like any simple statement would be. This is most useful for mixing `match` and `if` statements together, but also allows for compound `try` statements.

Example

Coconut:

```
if invalid(input_list):
    raise Exception()
else: match [head] + tail in input_list:
    print(head, tail)
else:
    print(input_list)
```

Python:

```
from collections.abc import Sequence
if invalid(input_list):
    raise Exception()
elif isinstance(input_list, Sequence):
    head, tail = inputlist[0], inputlist[1:]
    print(head, tail)
else:
    print(input_list)
```

3.8.4 `except` Statements

Python 3 requires that if multiple exceptions are to be caught, they must be placed inside of parentheses, so as to disallow Python 2's use of a comma instead of `as`. Coconut allows commas in except statements to translate to catching multiple exceptions without the need for parentheses, since, as in Python 3, `as` is always required to bind the exception to a name.

Example

Coconut:

```
try:
    unsafe_func(arg)
except SyntaxError, ValueError as err:
    handle(err)
```

Python:

```
try:
    unsafe_func(arg)
except (SyntaxError, ValueError) as err:
    handle(err)
```

3.8.5 Implicit `pass`

Coconut supports the simple `class name(base)` and `data name(args)` as aliases for `class name(base): pass` and `data name(args): pass`.

Example

Coconut:

```
data Empty
data Leaf(item)
data Node(left, right)
```

Python:

```
import collections
class Empty(collections.namedtuple("Empty", ""), object):
    __slots__ = ()
class Leaf(collections.namedtuple("Leaf", "n"), object):
    __slots__ = ()
class Node(collections.namedtuple("Node", "l, r"), object):
    __slots__ = ()
```

3.8.6 In-line `global` And `nonlocal` Assignment

Coconut allows for `global` or `nonlocal` to precede assignment to a variable or list of variables to make that assignment `global` or `nonlocal`, respectively.

Example

Coconut:

```
global state_a, state_b = 10, 100
```

Python:

```
global state_a, state_b; state_a, state_b = 10, 100
```

3.8.7 Code Passthrough

Coconut supports the ability to pass arbitrary code through the compiler without being touched, for compatibility with other variants of Python, such as Cython or Mython. Anything placed between `\(` and the corresponding close parenthesis will be passed through, as well as any line starting with `\\`, which will have the additional effect of allowing indentation under it.

Example

Coconut:

```
\\cdef f(x):
    return x |> g
```

Python:

```
cdef f(x):
    return g(x)
```

3.8.8 Enhanced Parenthetical Continuation

Since Coconut syntax is a superset of Python 3 syntax, Coconut supports the same line continuation syntax as Python. That means both backslash line continuation and implied line continuation inside of parentheses, brackets, or braces will all work.

In Python, however, there are some cases (such as multiple `with` statements) where only backslash continuation, and not parenthetical continuation, is supported. Coconut adds support for parenthetical continuation in all these cases.

Supporting parenthetical continuation everywhere allows the PEP 8 convention, which avoids backslash continuation in favor of implied parenthetical continuation, to always be possible to follow. From PEP 8:

> The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. Long lines can be broken over multiple lines by wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation.

Note: Passing `--strict` will enforce the PEP 8 convention by disallowing backslash continuations.

Example

Coconut:

```
with (open('/path/to/some/file/you/want/to/read') as file_1,
      open('/path/to/some/file/being/written', 'w') as file_2):
    file_2.write(file_1.read())
```

Python:

```
# split into two with statements for Python 2.6 compatibility
with open('/path/to/some/file/you/want/to/read') as file_1:
    with open('/path/to/some/file/being/written', 'w') as file_2:
        file_2.write(file_1.read())
```

3.9 Built-Ins

3.9.1 Enhanced Built-Ins

Coconut's `map`, `zip`, `filter`, `reversed`, and `enumerate` objects are enhanced versions of their Python equivalents that support `reversed`, `repr`, optimized normal (and iterator) slicing (all but `filter`), `len` (all but `filter`), and have added attributes which subclasses can make use of to get at the original arguments to the object:

- `map`: `_func`, `_iters`
- `zip`: `_iters`
- `filter`: `_func`, `_iter`
- `reversed`: `_iter`

- `enumerate`: `_iter`, `_start`

Example

Coconut:

```
map((+), range(5), range(6)) |> len |> print
range(10) |> filter$((x) -> x < 5) |> reversed |> tuple |> print
```

Python: *Can't be done without defining a custom `map` type. The full definition of `map` can be found in the Coconut header.*

3.9.2 `addpattern`

Takes one argument that is a *pattern-matching function*, and returns a decorator that adds the patterns in the existing function to the new function being decorated, where the existing patterns are checked first, then the new. Roughly equivalent to:

```
def addpattern(base_func):
    """Decorator to add a new case to a pattern-matching function, where the new case␣
↪is checked last."""
    def pattern_adder(func):
        def add_pattern_func(*args, **kwargs):
            try:
                return base_func(*args, **kwargs)
            except MatchError:
                return func(*args, **kwargs)
        return add_pattern_func
    return pattern_adder
```

DEPRECATED: Coconut also has a `prepattern` built-in, which adds patterns in the opposite order of `addpattern`; `prepattern` is defined as:

```
def prepattern(base_func):
    """Decorator to add a new case to a pattern-matching function,
    where the new case is checked first."""
    def pattern_prepender(func):
        return addpattern(func)(base_func)
    return pattern_prepender
```

Note: Passing `--strict` disables deprecated features.

Example

Coconut:

```
def factorial(0) = 1

@addpattern(factorial)
def factorial(n) = n * factorial(n - 1)
```

Python: *Can't be done without a complicated decorator definition and a long series of checks for each pattern-matching. See the compiled code for the Python syntax.*

3.9.3 `reduce`

Coconut re-introduces Python 2's `reduce` built-in, using the `functools.reduce` version.

Python Docs

reduce(*function, iterable*[, *initializer*])

Apply *function* of two arguments cumulatively to the items of *sequence*, from left to right, so as to reduce the sequence to a single value. For example, `reduce((x, y) -> x+y, [1, 2, 3, 4, 5])` calculates `((((1+2)+3)+4)+5)`. The left argument, *x*, is the accumulated value and the right argument, *y*, is the update value from the *sequence*. If the optional *initializer* is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty. If *initializer* is not given and *sequence* contains only one item, the first item is returned.

Example

Coconut:

```
product = reduce$(*)
range(1, 10) |> product |> print
```

Python:

```
import operator
import functools
product = functools.partial(functools.reduce, operator.mul)
print(product(range(1, 10)))
```

3.9.4 `takewhile`

Coconut provides `itertools.takewhile` as a built-in under the name `takewhile`.

Python Docs

takewhile(*predicate, iterable*)

Make an iterator that returns elements from the *iterable* as long as the *predicate* is true. Equivalent to:

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

Example

Coconut:

```
negatives = takewhile(numiter, (x) -> x<0)
```

Python:

```
import itertools
negatives = itertools.takewhile(numiter, lambda x: x<0)
```

3.9.5 dropwhile

Coconut provides `itertools.dropwhile` as a built-in under the name `dropwhile`.

Python Docs

dropwhile(*predicate, iterable*)

Make an iterator that drops elements from the *iterable* as long as the *predicate* is true; afterwards, returns every element. Note: the iterator does not produce any output until the predicate first becomes false, so it may have a lengthy start-up time. Equivalent to:

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

Example

Coconut:

```
positives = dropwhile(numiter, (x) -> x<0)
```

Python:

```
import itertools
positives = itertools.dropwhile(numiter, lambda x: x<0)
```

3.9.6 groupsof

Coconut provides the `groupsof` built-in to split an iterable into groups of a specific length. Specifically, `groupsof(n, iterable)` will split `iterable` into tuples of length `n`, with only the last tuple potentially of size `< n` if the length of `iterable` is not divisible by `n`.

Example

Coconut:

```
pairs = range(1, 11) |> groupsof$(2)
```

Python:

```
pairs = []
group = []
for item in range(1, 11):
    group.append(item)
    if len(group) == 2:
        pairs.append(tuple(group))
        group = []
if group:
    pairs.append(tuple(group))
```

3.9.7 `tee`

Coconut provides an optimized version of `itertools.tee` as a built-in under the name `tee`.

Python Docs

tee(*iterable, n=2*)

Return *n* independent iterators from a single iterable. Equivalent to:

```
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:             # when the local deque is empty
                newval = next(it)       # fetch a new value and
                for d in deques:        # load it to all the deques
                    d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

Once `tee()` has made a split, the original *iterable* should not be used anywhere else; otherwise, the *iterable* could get advanced without the tee objects being informed.

This itertool may require significant auxiliary storage (depending on how much temporary data needs to be stored). In general, if one iterator uses most or all of the data before another iterator starts, it is faster to use `list()` instead of `tee()`.

Example

Coconut:

```
original, temp = tee(original)
sliced = temp$[5:]
```

Python:

```
import itertools
original, temp = itertools.tee(original)
sliced = itertools.islice(temp, 5, None)
```

3.9.8 `reiterable`

Sometimes, when an iterator may need to be iterated over an arbitrary number of times, `tee` can be cumbersome to use. For such cases, Coconut provides `reiterable`, which wraps the given iterator such that whenever an attempt to iterate over it is made, it iterates over a `tee` instead of the original.

Example

Coconut:

```
def list_type(xs):
    case reiterable(xs):
        match [fst, snd] :: tail:
            return "at least 2"
        match [fst] :: tail:
            return "at least 1"
        match (| |):
            return "empty"
```

Python: *Can't be done without a long series of checks for each `match` statement. See the compiled code for the Python syntax.*

3.9.9 `consume`

Coconut provides the `consume` function to efficiently exhaust an iterator and thus perform any lazy evaluation contained within it. `consume` takes one optional argument, `keep_last`, that defaults to 0 and specifies how many, if any, items from the end to return as an iterable (`None` will keep all elements).

Equivalent to:

```
def consume(iterable, keep_last=0):
    """Fully exhaust iterable and return the last keep_last elements."""
    return collections.deque(iterable, maxlen=keep_last)  # fastest way to exhaust an␣
↪iterator
```

Rationale

In the process of lazily applying operations to iterators, eventually a point is reached where evaluation of the iterator is necessary. To do this efficiently, Coconut provides the `consume` function, which will fully exhaust the iterator given to it.

Example

Coconut:

```
range(10) |> map$((x) -> x**2) |> map$(print) |> consume
```

Python:

```
collections.deque(map(print, map(lambda x: x**2, range(10))), maxlen=0)
```

3.9.10 `count`

Coconut provides a modified version of `itertools.count` that supports `in`, normal slicing, optimized iterator slicing, `count` and `index` sequence methods, `repr`, and `start` and `step` attributes as a built-in under the name `count`.

Python Docs

count(*start=0, step=1*)

Make an iterator that returns evenly spaced values starting with number *start*. Often used as an argument to `map()` to generate consecutive data points. Also, used with `zip()` to add sequence numbers. Roughly equivalent to:

```
def count(start=0, step=1):
    # count(10) --> 10 11 12 13 14 ...
    # count(2.5, 0.5) -> 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

Example

Coconut:

```
count()$[10**100] |> print
```

Python: *Can't be done quickly without Coconut's iterator slicing, which requires many complicated pieces. The necessary definitions in Python can be found in the Coconut header.*

3.9.11 `makedata`

Coconut provides the `makedata` function to allow direct access to the base constructor of data types created with the Coconut `data` statement. This is particularly useful when writing alternative constructors for data types by overwriting `__new__`.

`makedata` takes the data type to call as the first argument, and the arguments for constructing that data type as the rest of the arguments. For `data` objects, `makedata` behaves as the underlying data type's original constructor, exactly as the data type was declared. For non-`data` objects, `makedata` is equivalent to:

```
def makedata(data_type, *args, **kwargs):
    """Returns base data constructor of data_type."""
    return super(data_type, data_type).__new__(data_type, *args, **kwargs)
```

DEPRECATED: Coconut also has a `datamaker` built-in, which partially applies `makedata`; `datamaker` is defined as:

```
def datamaker(data_type):
    """Get the original constructor of the given data type or class."""
    return makedata$(data_type)
```

Note: Passing `--strict` disables deprecated features.

Example

Coconut:

```
data Tuple(elems):
    def __new__(cls, *elems):
        return elems |> makedata$(cls)
```

Python:

```
import collections
class Tuple(collections.namedtuple("Tuple", "elems"), object):
    __slots__ = ()
    def __new__(cls, *elems):
        return super(cls, cls).__new__(cls, elems)
```

3.9.12 `fmap`

In functional programming, `fmap(func, obj)` takes a data type `obj` and returns a new data type with `func` mapped over the contents. Coconut's `fmap` function does the exact same thing in Coconut.

`fmap` can also be used on built-ins such as `str`, `list`, `set`, and `dict` as a variant of `map` that returns back an object of the same type. The behavior of `fmap` for a given object can be overridden by defining an `__fmap__(self, func)` method that will be called whenever `fmap` is invoked on that object.

For `dict`, or any other `collections.abc.Mapping`, `fmap` will be called on the mapping's `.items()` instead of the default iteration through its `.keys()`.

Example

Coconut:

```
[1, 2, 3] |> fmap$(x -> x+1) == [2, 3, 4]

data Nothing()
data Just(n)

Just(3) |> fmap$(x -> x*2) == Just(6)
Nothing() |> fmap$(x -> x*2) == Nothing()
```

Python:

```
list(map(lambda x: x+1, [1, 2, 3])) == [2, 3, 4]

import collections
class Nothing(collections.namedtuple("Nothing", ""), object):
    __slots__ = ()
class Just(collections.namedtuple("Just", "n"), object):
    __slots__ = ()

Just(*map(lambda x: x*2, Just(3))) == Just(6)
Nothing(*map(lambda x: x*2, Nothing())) == Nothing()
```

3.9.13 `starmap`

Coconut provides a modified version of `itertools.starmap` that supports `reversed`, `repr`, optimized normal (and iterator) slicing, `len`, and `_func` and `_iter` attributes.

Python Docs

starmap(*function, iterable*)

Make an iterator that computes the function using arguments obtained from the iterable. Used instead of `map()` when argument parameters are already grouped in tuples from a single iterable (the data has been "pre-zipped"). The difference between `map()` and `starmap()` parallels the distinction between `function(a,b)` and `function(*c)`. Roughly equivalent to:

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

Example

Coconut:

```
range(1, 5) |> map$(range) |> starmap$(print) |> consume
```

Python:

```
import itertools, collections
collections.deque(itertools.starmap(print, map(range, range(1, 5))), maxlen=0)
```

3.9.14 `scan`

Coconut provides a modified version of `itertools.accumulate` with opposite argument order as `scan` that also supports `repr`, `len`, and `func` and `iter` attributes. `scan` works exactly like *reduce*, except that instead of only returning the last accumulated value, it returns an iterator of all the intermediate values.

Python Docs

scan(*func, iterable*)

Make an iterator that returns accumulated results of some function of two arguments. Elements of the input iterable may be any type that can be accepted as arguments to func. (For example, with the operation of addition, elements may be any addable type including Decimal or Fraction.) If the input iterable is empty, the output iterable will also be empty.

Roughly equivalent to:

```
def scan(func, iterable):
    'Return running totals'
    # scan(operator.add, [1,2,3,4,5]) --> 1 3 6 10 15
    # scan(operator.mul, [1,2,3,4,5]) --> 1 2 6 24 120
    it = iter(iterable)
    try:
```

```
        total = next(it)
    except StopIteration:
        return
    yield total
    for element in it:
        total = func(total, element)
        yield total
```

Example

Coconut:

```
input_data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
running_max = input_data |> scan$(max) |> list
```

Python:

```
input_data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
running_max = []
max_so_far = input_data[0]
for x in input_data:
    if x > max_so_far:
        max_so_far = x
    running_max.append(x)
```

3.9.15 `recursive_iterator`

Coconut provides a `recursive_iterator` decorator that provides significant optimizations for any stateless, recursive function that returns an iterator. To use `recursive_iterator` on a function, it must meet the following criteria:

1. your function either always `return`s an iterator or generates an iterator using `yield`,
2. when called multiple times with the same arguments, your function produces the same iterator (your function is stateless), and
3. your function gets called (usually calls itself) multiple times with the same arguments.

If you are encountering a `RuntimeError` due to maximum recursion depth, it is highly recommended that you rewrite your function to meet either the criteria above for `recursive_iterator`, or the corresponding criteria for Coconut's *tail call optimization*, either of which should prevent such errors.

Furthermore, `recursive_iterator` also allows the resolution of a nasty segmentation fault in Python's iterator logic that has never been fixed. Specifically, instead of writing

```
seq = get_elem() :: seq
```

which will crash due to the aforementioned Python issue, write

```
@recursive_iterator
def seq() = get_elem() :: seq()
```

which will work just fine.

Example

Coconut:

```
@recursive_iterator
def fib() = (1, 1) :: map((+), fib(), fib()$[1:])
```

Python: *Can't be done without a long decorator definition. The full definition of the decorator in Python can be found in the Coconut header.*

3.9.16 `parallel_map`

Coconut provides a parallel version of `map` under the name `parallel_map`. `parallel_map` makes use of multiple processes, and is therefore much faster than `map` for CPU-bound tasks. Use of `parallel_map` requires `concurrent.futures`, which exists in the Python 3 standard library, but under Python 2 will require `pip install futures` to function.

Because `parallel_map` uses multiple processes for its execution, it is necessary that all of its arguments be pickleable. Only objects defined at the module level, and not lambdas, objects defined inside of a function, or objects defined inside of the interpreter, are pickleable. Furthermore, on Windows, it is necessary that all calls to `parallel_map` occur inside of an `if __name__ == "__main__"` guard.

Python Docs

parallel_map(*func, *iterables*)

Equivalent to `map(func, *iterables)` except *func* is executed asynchronously and several calls to *func* may be made concurrently. If a call raises an exception, then that exception will be raised when its value is retrieved from the iterator.

Example

Coconut:

```
parallel_map(pow$(2), range(100)) |> list |> print
```

Python:

```
import functools
import concurrent.futures
with concurrent.futures.ProcessPoolExecutor() as executor:
    print(list(executor.map(functools.partial(pow, 2), range(100))))
```

3.9.17 `concurrent_map`

Coconut provides a concurrent version of `map` under the name `concurrent_map`. `concurrent_map` makes use of multiple threads, and is therefore much faster than `map` for IO-bound tasks. Use of `concurrent_map` requires `concurrent.futures`, which exists in the Python 3 standard library, but under Python 2 will require `pip install futures` to function.

Python Docs

concurrent_map(*func, *iterables*)

Equivalent to `map(func, *iterables)` except *func* is executed asynchronously and several calls to *func* may be made concurrently. If a call raises an exception, then that exception will be raised when its value is retrieved from the iterator.

Example

Coconut:

```
concurrent_map(get_data_for_user, get_all_users()) |> list |> print
```

Python:

```
import functools
import concurrent.futures
with concurrent.futures.ThreadPoolExecutor() as executor:
    print(list(executor.map(get_data_for_user, get_all_users())))
```

3.9.18 `MatchError`

A `MatchError` is raised when a *destructuring assignment* statement fails, and thus `MatchError` is provided as a built-in for catching those errors. `MatchError` objects support two attributes, `pattern`, which is a string describing the failed pattern, and `value`, which is the object that failed to match that pattern.

3.10 Coconut Modules

3.10.1 `coconut.__coconut__`

It is sometimes useful to be able to access Coconut built-ins from pure Python. To accomplish this, Coconut provides `coconut.__coconut__`, which behaves exactly like the `__coconut__.py` header file included when Coconut is compiled in package mode.

All Coconut built-ins are accessible from `coconut.__coconut__`. The recommended way to import them is to use `from coconut.__coconut__ import` and import whatever built-ins you'll be using.

Example

```
from coconut.__coconut__ import parallel_map
```

3.10.2 `coconut.convenience`

It is sometimes useful to be able to use the Coconut compiler from code, instead of from the command line. The recommended way to do this is to use `from coconut.convenience import` and import whatever convenience functions you'll be using. Specifications of the different convenience functions are as follows.

parse

coconut.convenience.parse([*code,* [*mode*]])

Likely the most useful of the convenience functions, `parse` takes Coconut code as input and outputs the equivalent compiled Python code. The second argument, *mode*, is used to indicate the context for the parsing.

If *code* is not passed, `parse` will output just the given *mode*'s header, which can be executed to set up an execution environment in which future code can be parsed and executed without a header.

Each *mode* has two components: what parser it uses, and what header it prepends. The parser determines what Coconut code is allowed as input, and the header determines how the compiled Python can be used. Possible values of *mode* are:

- `"sys"`: (the default)
  - parser: file
    - The file parser can parse any Coconut code.
  - header: sys
    - This header imports `coconut.__coconut__` to access the necessary Coconut objects.
- `"exec"`:
  - parser: file
  - header: exec
    - When passed to `exec` at the global level, this header will create all the necessary Coconut objects itself instead of importing them.
- `"file"`:
  - parser: file
  - header: file
    - This header is meant to be written to a `--standalone` file and should not be passed to `exec`.
- `"package"`:
  - parser: file
  - header: package
    - This header is meant to be written to a `--package` file and should not be passed to `exec`.
- `"block"`:
  - parser: file
  - header: none
    - No header is included, thus this can only be passed to `exec` if code with a header has already been executed at the global level.
- `"single"`:
  - parser: single
    - Can only parse one line of Coconut code.
  - header: none
- `"eval"`:
  - parser: eval

        * Can only parse a Coconut expression, not a statement.
    - header: none
- `"debug"`:
    - parser: debug
        * Can parse any Coconut code, allows leading whitespace, and has no trailing newline.
    - header: none

Example

```
from coconut.convenience import parse
exec(parse())
while True:
    exec(parse(input(), mode="block"))
```

setup

coconut.convenience.setup(*target, strict, minify, line_numbers, keep_lines, no_tco*)

`setup` can be used to pass command line flags for use in `parse`. The possible values for each flag argument are:

- *target*: `None` (default), or any *allowable target*
- *strict*: `False` (default) or `True`
- *minify*: `False` (default) or `True`
- *line_numbers*: `False` (default) or `True`
- *keep_lines*: `False` (default) or `True`
- *no_tco*: `False` (default) or `True`

cmd

coconut.convenience.cmd(*args*, [*interact*])

Executes the given *args* as if they were fed to `coconut` on the command-line, with the exception that unless *interact* is true or `-i` is passed, the interpreter will not be started. Additionally, since `parse` and `cmd` share the same convenience parsing object, any changes made to the parsing with `cmd` will work just as if they were made with `setup`.

version

coconut.convenience.version([*which*])

Retrieves a string containing information about the Coconut version. The optional argument *which* is the type of version information desired. Possible values of *which* are:

- `"num"`: the numerical version (the default)
- `"name"`: the version codename
- `"spec"`: the numerical version with the codename attached
- `"tag"`: the version tag used in GitHub and documentation URLs

- `"-v"`: the full string printed by `coconut -v`

CoconutException

If an error is encountered in a convenience function, a `CoconutException` instance may be raised. `coconut.convenience.CoconutException` is provided to allow catching such errors.

CHAPTER 4

Coconut Contributing Guidelines

By contributing to Coconut, you consent to your contribution being released under Coconut's Apache 2.0 license.

Anyone is welcome to submit an issue or pull request! The purpose of this document is simply to explain the contribution process and the internals of how Coconut works to make contributing easier.

Note: If you are considering contributing to Coconut, you'll be doing so on the `develop` branch, which means you should be viewing the `develop` version of the Contributing Guidelines, if you aren't doing so already.

4.1 Asking Questions

If you are thinking about contributing to Coconut, please don't hesitate to ask questions at Coconut's Gitter! That includes any questions at all about contributing, including understanding the source code, figuring out how to implement a specific change, or just trying to figure out what needs to be done.

4.2 Contribution Process

Contributing to Coconut is as simple as

1. forking Coconut on GitHub,
2. making changes to the `develop` branch, and
3. proposing a pull request.

Note: Don't forget to add yourself to the "Authors:" section in the docstrings of any files you modify!

4.3 Contributor Friendly Issues

Want to help out, but don't know what to work on? Head over to Coconut's open issues and look for ones labeled "contributor friendly." Contributor friendly issues are those that require less intimate knowledge of Coconut's inner workings, and are thus possible for new contributors to work on.

4.4 Testing New Changes

First, you'll want to set up a local copy of Coconut's recommended development environment. For that, just run `git checkout develop` and `make dev`. That should switch you to the `develop` branch, install all possible dependencies, bind the `coconut` command to your local copy, and set up pre-commit, which will check your code for errors for you whenever you `git commit`.

Then, you should be able to use the Coconut command-line for trying out simple things, and to run a paired-down version of the test suite locally, just `make test-basic`.

After you've tested your changes locally, you'll want to add more permanent tests to Coconut's test suite. Coconut's test suite is primarily written in Coconut itself, so testing new features just means using them inside of one of Coconut's `.coco` test files, with some `assert` statements to check validity.

4.5 File Layout

- `DOCS.md`
  - Markdown file containing detailed documentation on every Coconut feature. If you are adding a new feature, you should also add documentation on it to this file.
- `FAQ.md`
  - Markdown file containing frequently asked questions and their answers. If you had a question you wished was answered earlier when learning Coconut, you should add it to this file.
- `HELP.md`
  - Markdown file containing Coconut's tutorial. The tutorial should be a streamlined introduction to Coconut and all of its most important features.
- `Makefile`
  - Contains targets for installing Coconut, building the documentation, checking for dependency updates, etc.
- `setup.py`
  - Using information from `requirements.py` and `constants.py` to install Coconut. Also reads `README.rst` to generate the PyPI description.
- `conf.py`
  - Sphinx configuration file for Coconut's documentation.
- coconut
  - `__coconut__.py`
    - Mimics the Coconut header by generating and executing it when imported. Used by the REPL.
  - `__init__.py`
    - Includes the implementation of the `%coconut` IPython magic.

- `__main__.py`
  - Imports and runs `main` from `main.py`.
- `constants.py`
  - All constants used across Coconut are defined here, including dependencies, magic numbers/strings, etc.
- `convenience.py`
  - Contains `cmd`, `version`, `setup`, and `parse` functions as convenience utilities when using Coconut as a module. Documented in `DOCS.md`.
- `exceptions.py`
  - All of the exceptions raised by Coconut are defined here, both those shown to the user and those used only internally.
- `highlighter.py`
  - Contains Coconut's Pygments syntax highlighter, as well as modified Python highlighters that don't fail if they encounter unknown syntax.
- `main.py`
  - Contains `main` and `main_run`, the entry points for the `coconut` and `coconut-run` commands, respectively.
- `requirements.py`
  - Processes Coconut's requirements from `constants.py` into a form `setup.py` can use, as well as checks for updates to Coconut's dependencies.
- `root.py`
  - `root.py` creates and executes the part of Coconut's header that normalizes Python built-ins across versions. Whenever you are writing a new file, you should always add `from coconut.root import *` to ensure compatibility with different Python versions. `root.py` also sets basic version-related constants.
- `terminal.py`
  - Contains utilities for displaying messages to the console, mainly `logger`, which is Coconut's primary method of logging a message from anywhere.
- command
  - `__init__.py`
    - Imports everything in `command.py`.
  - `cli.py`
    - Creates the `ArgumentParser` object used to parse Coconut command-line arguments.
  - `command.py`
    - Contains `Command`, whose `start` method is the main entry point for the Coconut command-line utility.
  - `mypy.py`
    - Contains objects necessary for Coconut's `--mypy` flag.
  - `util.py`

        - Contains utilities used by `command.py`, including `Prompt` for getting syntax-highlighted input, and `Runner` for executing compiled Python.
    * `watch.py`
        - Contains objects necessary for Coconut's `--watch` flag.
- compiler
    * `__init__.py`
        - Imports everything in `compiler.py`.
    * `compiler.py`
        - Contains `Compiler`, the class that actually compiles Coconut code. `Compiler` inherits from `Grammar` in `grammar.py` to get all of the basic grammatical definitions, then extends them with all of the handlers that depend on the compiler's options (e.g. the current `--target`). `Compiler` also does pre- and post-processing, including replacing strings with markers (pre-processing) and adding the header (post-processing).
    * `grammar.py`
        - Contains `Grammar`, the class that specifies Coconut's grammar in PyParsing. Coconut performs one-pass compilation by attaching "handlers" to specific grammar objects to transform them into compiled Python. `grammar.py` contains all basic (non-option-dependent) handlers.
    * `header.py`
        - Contains `getheader`, which generates the header at the top of all compiled Coconut files.
    * `matching.py`
        - Contains `Matcher`, which handles the compilation of all Coconut pattern-matching, including `match` statements, destructuring assignment, and pattern-matching functions.
    * `util.py`
        - Contains utilities for working with PyParsing objects that are primarily used by `grammar.py`.
    * templates
        - `header.py_template`
        - Template for the main body of Coconut's header; use and formatting of this file is all in `header.py`.
- icoconut
    * `__init__.py`
        - Imports everything from `icoconut/root.py`.
    * `__main__.py`
        - Contains the main entry point for Coconut's Jupyter kernel.
    * `root.py`
        - Contains the implementation of Coconut's Jupyter kernel, made by subclassing the IPython kernel.
- stubs
    * `__coconut__.pyi`
        - A MyPy stub file for specifying the type of all the objects defined in Coconut's package header (which is saved as `__coconut__.py`).

- tests
  - `__init__.py`
    - Imports everything in `main_test.py`.
  - `__main__.py`
    - When run, compiles all of the test source code, but *does not run any tests*. To run the tests, the command `make test`, or a `pytest` command to run a specific test, is necessary.
  - `main_test.py`
    - Contains `TestCase` subclasses that run all of the commands for testing the Coconut files in `src`.
  - src
    - `extras.coco`
      - Directly imports and calls functions in the Coconut package, including from `convenience.py` and icoconut.
    - `runnable.coco`
      - Makes sure the argument `--arg` was passed when running the file.
    - `runner.coco`
      - Runs `main` from `cocotest/agnostic/main.py`.
    - cocotest
      - *Note: Files in the folders below all get compiled into the top-level cocotest directory. The folders are only for differentiating what files to compile on what Python version.*
      - agnostic
      - `__init__.coco`
      - Contains a docstring that `main.coco` asserts exists.
      - `main.coco`
      - Contains the main test entry point as well as many simple, one-line tests.
      - `specific.coco`
      - Tests to be run only on a specific Python version, but not necessarily only under a specific `--target`.
      - `suite.coco`
      - Tests objects defined in `util.coco`.
      - `tutorial.coco`
      - Tests all the examples in `TUTORIAL.md`.
      - `util.coco`
      - Contains objects used in `suite.coco`.
      - python2
      - `py2_test.coco`
      - Tests to be run only on Python 2 with `--target 2`.
      - python3

- `py3_test.coco`
- Tests to be run only on Python 3 with `--target 3`.
- python35
- `py35_test.coco`
- Tests to be run only on Python 3.5 with `--target 3.5`.

4.6 Release Process

1. Preparation:
   (a) Run `make check` and update dependencies as necessary
   (b) Run `make format`
   (c) Check changes in `compiled-cocotest` and `pyprover`
   (d) Check Codacy issues (for `coconut` and `compiled-cocotest`) and LGTM alerts
   (e) Make sure `coconut-develop` package looks good
   (f) Run `make docs` and ensure local documentation looks good
   (g) Make sure develop documentation looks good
   (h) Make sure Travis and AppVeyor are passing
   (i) Turn off `develop` in `root.py`
   (j) Set `root.py` to new version number
   (k) If major release, set `root.py` to new version name
2. Pull Request:
   (a) Create a pull request to merge `develop` into `master`
   (b) Link contributors on pull request
   (c) Wait until everything is passing
3. Release:
   (a) Release `sublime-coconut` first if applicable
   (b) Merge pull request and mark as resolved
   (c) Release `master` on GitHub
   (d) Fetch and switch to `master` locally
   (e) Run `make upload`
   (f) Run `make docs` and upload docs to PyPI
   (g) Switch back to `develop` locally
   (h) Update from master
   (i) Turn on `develop` in `root`
   (j) Run `make dev`
   (k) Push to `develop`

(l) Update website if it needs updating

(m) Wipe all updated versions on readthedocs

(n) Copy PyPI keywords to readthedocs tags

(o) Build all updated versions on readthedocs

(p) Submit PR to update Coconut's `conda-forge` feedstock

(q) Close release milestone

Coconut (coconut-lang.org) is a variant of Python that **adds on top of Python syntax** new features for simple, elegant, Pythonic **functional programming**.

Coconut is developed on GitHub and hosted on PyPI. Installing Coconut is as easy as opening a command prompt and entering:

```
pip install coconut
```

after which the entire world of Coconut will be at your disposal. To help you get started, check out these links for more information about Coconut:

- Tutorial: If you're new to Coconut, a good place to start is Coconut's **tutorial**.
- Documentation: If you're looking for info about a specific feature, check out Coconut's **documentation**.
- FAQ: If you have questions about who Coconut is built for and whether or not you should use it, Coconut's frequently asked questions have you covered.
- Create a New Issue: If you're having a problem with Coconut, creating a new issue detailing the problem will allow it to be addressed as soon as possible.
- Gitter: For any questions, concerns, or comments about anything Coconut-related, ask around at Coconut's Gitter, a GitHub-integrated chat room for Coconut developers.
- Releases: Want to know what's been added in recent Coconut versions? Check out the release log for all the new features and fixes.

Note: If the above documentation links are not working, try the mirror .