Coconut

Release

June 24, 2016

Contents

CHAPTER 1

Coconut Frequently Asked Questions

1.1 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.

1.2 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 `--linenumbers` 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. 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.

1.3 I don't like functional programming, should I still use 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.

1.4 I've never done functional programming before, should I still try to 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.

1.5 I don't know Python very well, should I still try to 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.

1.6 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.

1.7 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.

1.8 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. Contributing to Coconut is as simple as forking Coconut's `develop` branch on GitHub, making your changes, and proposing a pull request. If you have 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, try asking around at Coconut's Gitter, a GitHub-integrated chat room for Coconut developers.

1.9 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.

1.10 Who developed Coconut?

Evan Hubinger is an undergraduate student studying mathematics and computer science at Harvey Mudd College.

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 fundamentally *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 recursion 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.

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

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

```
python -m pip install coconut
```

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 open an issue and it'll be addressed 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.

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 `--linenumbers` (or `-l` for short). Using `--linenumbers` 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 is occurring, 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 both as a kernel for Jupyter notebooks and consoles, and as an extension inside of the IPython kernel.

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.

To use Coconut as an extension inside of the IPython kernel, type the code

```
%load_ext coconut
```

into your IPython notebook or console, and then to run Coconut code, use

```
%coconut <code>
```

or

```
%%coconut <command-line-args>
<code>
```

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 comprehensive 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 `1`, `6`, and then two `TypeErrors`.

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 _ 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 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 `_ is int`, which performs an `isinstance` check on `n` against `int`, then checks whether `n > 0`, and if those are true, executes `return n * factorial(n-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:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    try:
        0 = n # destructuring assignment
    except MatchError:
        try:
            _ is int = n # also destructuring assignment
        except MatchError:
            pass
        else: if n > 0: # in Coconut, if, match, and try are allowed after else
            return n * factorial(n-1)
    else:
        return 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, which *are really just fancy normal 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`.

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: by writing it in a tail-recursive style, where it directly returns all calls to itself, and using Coconut's `recursive` decorator, like so:

```
@recursive
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 version is exactly equivalent to the original version, with the exception that it will never raise a `MaximumRecursionDepthError`, because Coconut's `recursive` decorator will optimize away the tail recursion into a `while` loop.

2.3.3 Iterative Method

The final, and other 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 on 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. In fact, 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 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 built-in Coconut function `addpattern`, we can take that from three indentation levels down to one. Take a look:

```
def factorial(0):
    return 1

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

```
    return 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 two major new concepts to talk about here: `addpattern`, of course, and pattern-matching function definition—how both of the functions above are defined.

First, pattern-matching function definition. Pattern-matching function definition does exactly that—pattern-matches against all the arguments that are passed to the function. There are a couple of things to watch out for when using pattern-matching function definition, however. First, that if the pattern doesn't match (if for example the wrong number of arguments are passed), your function will raise a `MatchError`, and second, that keyword arguments aren't allowed. 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`.

Second, `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 imperative 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."""
    return 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. That's because since we're not going to write `quick_sort` in a tail-recursive style, we can't use `recursive`, and thus there's no reason to write the whole thing as one function and we might as well use `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([]):
    return []

@addpattern(quick_sort)
def quick_sort([head] + tail):
    """Sort the input sequence using the quick sort algorithm."""
    return (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, and without using any data until nece
    match [head] :: tail in l:
        tail, tail_ = tee(tail)
        yield from (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 |> 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 built-in function `tee`. `tee` solves a problem for functional programming created by the implementation of Python's iterators: whenever an element of an iterator is accessed, it's lost. `tee` solves this problem by splitting an iterator in two (or more if the optional argument `n` is passed) independent iterators that both use the same underlying iterator to access their data, thus when an element of one is accessed, it isn't lost in the other.

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, and without using any data until nece
    match [head] :: tail in l:
        tail, tail_ = tee(tail)
        yield from (quick_sort((x for x in tail if x < head))
            :: (head,)
            :: quick_sort((x for x in tail_ if x >= head))
            )
```

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. 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."""
        return (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
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`.

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."""
        if len(pts) == 1 and pts[0] `isinstance` vector:
            return pts[0] # vector(v) where v is a vector should return v
        else:
            return pts |> tuple |> datamaker(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 access to the underlying `data` constructor will be necessary. To achieve this, Coconut provides the built-in function `datamaker`, which takes a data type, often the first argument to `__new__`, and returns its underlying `data` constructor.

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 creating a tuple of the arguments and passing that to the underlying constructor, the form of which is `vector(pts)`, thus assigning the tuple to the `pts` attribute.

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."""
        return self.pts |> map$((x) -> x**2) |> sum |> ((s) -> s**0.5)
```

The basic algorithm here is map square over each element, sum them all, then square root the result, an algorithm which is so clean to implement in Coconut that it can be read right off the code.

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, other):
        """Add two vectors together."""
        vector(other_pts) = other
        assert len(other_pts) == len(self.pts)
        return map((+), self.pts, other_pts) |*> vector
```

There are a couple of new constructs here, but the main notable one is the destructuring assignment statement `vector(other_pts) = other` 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) = other` 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`.

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)`.

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

```
    def __sub__(self, other):
        """Subtract one vector from another."""
        vector(other_pts) = other
        assert len(other_pts) == len(self.pts)
        return 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."""
        return self.pts |> map$((-)) |*> vector
```

Our next method will be equality. We're again going to use `data` pattern-matching to implement this, but this time inside of a `match` statement instead of with destructuring assignment, since we want to `return False` not raise an error if the match fails. Here's the code:

```
    def __eq__(self, other):
        """Compare whether two vectors are equal."""
        match vector(=self.pts) in other:
            return True
        else:
            return False
```

The only new construct here is the use of `=self.pts` in the `match` statement. This construct is used to perform a check inside of the pattern-matching, making sure the `match` only succeeds if `other.pts == self.pts`.

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."""
        return 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."""
        if len(pts) == 1 and pts[0] `isinstance` vector:
            return pts[0] # vector(v) where v is a vector should return v
        else:
            return pts |> tuple |> datamaker(cls) # accesses base constructor
    def __abs__(self):
        """Return the magnitude of the vector."""
        return self.pts |> map$((x) -> x**2) |> sum |> ((s) -> s**0.5)
```

```
    def __add__(self, other):
        """Add two vectors together."""
        vector(other_pts) = other
        assert len(other_pts) == len(self.pts)
        return map((+), self.pts, other_pts) |*> vector
    def __sub__(self, other):
        """Subtract one vector from another."""
        vector(other_pts) = other
        assert len(other_pts) == len(self.pts)
        return map((-), self.pts, other_pts) |*> vector
    def __neg__(self):
        """Retrieve the negative of the vector."""
        return self.pts |> map$((-)) |*> vector
    def __eq__(self, other):
        """Compare whether two vectors are equal."""
        match vector(=self.pts) in other:
            return True
        else:
            return False
    def __mul__(self, other):
        """Scalar multiplication and dot product."""
        match vector(other_pts) in other:
            assert len(other_pts) == len(self.pts)
            return map((*), self.pts, other_pts) |> sum # dot product
        else:
            return self.pts |> map$((*)$(other)) |*> vector # scalar multiplication
    def __rmul__(self, other):
        """Necessary to make scalar multiplication commutative."""
        return 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. To help with that, we're going to introduce a new concept up front, shorthand functions. A shorthand function looks like this

```
def <name>(<args>) = <return value>
```

which has the advantage over the classic Python

```
def <name>(<args>): return <return value>
```

of being shorter, more readable, and not requiring `return` to be typed out. If you try to go for the one-liner approach, using shorthand functions will help keep your lines short and your code readable.

With that out of the way, it's time to 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:

```
...

(0,2)<  ...
      \_
(0,1)<  (1,1)<  ...
      \_      \_
(0,0) > (1,0) > (2,0) > ...
```

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 `n`th 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.

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

```
def vector_field() = linearized_plane() |> map$((xy) -> vector(*xy))
```

All we're doing is taking our `linearized_plane` and mapping `vector` over it, but making sure to call vector 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."""
        if len(pts) == 1 and pts[0] `isinstance` vector:
            return pts[0] # vector(v) where v is a vector should return v
        else:
            return pts |> tuple |> datamaker(cls) # accesses base constructor
    def __abs__(self):
        """Return the magnitude of the vector."""
        return self.pts |> map$((x) -> x**2) |> sum |> ((s) -> s**0.5)
    def __add__(self, other):
        """Add two vectors together."""
        vector(other_pts) = other
        assert len(other_pts) == len(self.pts)
        return map((+), self.pts, other_pts) |*> vector
    def __sub__(self, other):
        """Subtract one vector from another."""
        vector(other_pts) = other
        assert len(other_pts) == len(self.pts)
        return map((-), self.pts, other_pts) |*> vector
    def __neg__(self):
        """Retrieve the negative of the vector."""
        return self.pts |> map$((-)) |*> vector
    def __eq__(self, other):
        """Compare whether two vectors are equal."""
        match vector(=self.pts) in other:
            return True
        else:
            return False
    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."""
        return self * other

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

# 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 shorthand 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."""
        if len(pts) == 1 and pts[0] `isinstance` vector:
            return pts[0] # vector(v) where v is a vector should return v
        else:
            return pts |> tuple |> datamaker(cls) # accesses base constructor
    def __abs__(self):
        """Return the magnitude of the vector."""
        return self.pts |> map$((x) -> x**2) |> sum |> ((s) -> s**0.5)
    def __add__(self, other):
        """Add two vectors together."""
        vector(other_pts) = other
        assert len(other_pts) == len(self.pts)
        return map((+), self.pts, other_pts) |*> vector
    def __sub__(self, other):
        """Subtract one vector from another."""
        vector(other_pts) = other
        assert len(other_pts) == len(self.pts)
        return map((-), self.pts, other_pts) |*> vector
    def __neg__(self):
        """Retrieve the negative of the vector."""
        return self.pts |> map$((-)) |*> vector
    def __eq__(self, other):
        """Compare whether two vectors are equal."""
        match vector(=self.pts) in other:
            return True
        else:
            return False
    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."""
        return 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
```

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:

```
.name
obj.
func$
seq[]
iter$[]
```

2.8.4 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 documented in detail in Coconut's comprehensive 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.

CHAPTER 3

Coconut Documentation

3.1 Overview

This documentation covers all the technical details 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 HELP.

Coconut is a variant of Python built for **simple, elegant, Pythonic functional programming**. Coconut syntax is a strict superset of Python 3 syntax. That means 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

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:

```
python -m pip install coconut
```

3.2.2 Usage

```
coconut [-h] [-v] [source] [dest] [-t version] [-s] [-l] [-p] [-a] [-f] [-d] [-r] [-n] [-m] [-i] [-q
```

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, --target              specify target Python version (defaults to universal)
-s, --strict              enforce code cleanliness standards
-l, --linenumbers         add line number comments for ease of debugging
-p, --package             compile source as part of a package (defaults to only if source is a directo
-a, --standalone          compile source as standalone files (defaults to only if source is a single f
-f, --force               force overwriting of compiled Python (otherwise only overwrites when source
-d, --display             print compiled Python
-r, --run                 run compiled Python (often used with --nowrite)
-n, --nowrite             disable writing compiled Python
-m, --minify              compress compiled Python
-i, --interact            force the interpreter to start (otherwise starts if no other command is give
-q, --quiet               suppress all informational output (combine with --display to write runnable
-c code, --code code      run a line of Coconut passed in as a string (can also be passed into stdin)
--jupyter, --ipython      run Jupyter/IPython with Coconut as the kernel (remaining args passed to Jupy
--autopep8 ...            use autopep8 to format compiled code (remaining args passed to autopep8)
--recursionlimit          set maximum recursion depth (default is system dependent)
--tutorial                open the Coconut tutorial in the default web browser
--documentation           open the Coconut documentation in the default web browser
--color color             show all Coconut messages in the given color
--verbose                 print verbose debug output
```

3.2.3 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`.

3.2.4 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.

3.2.5 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` and PyPy `2.7`, `3.2`.

As part of Coconut's cross-compatibility efforts, Coconut adds in new Python 3 built-ins and overwrites Python 2 built-ins to use the Python 3 versions where possible. If access to the Python 2 versions is desired, the old built-ins can be retrieved by prefixing them with `py2_`. The old built-ins available are:

- `py2_chr`
- `py2_filter`
- `py2_hex`
- `py2_input`
- `py2_int`
- `py2_map`
- `py2_oct`
- `py2_open`
- `py2_print`
- `py2_range`
- `py2_raw_input`
- `py2_str`
- `py2_xrange`
- `py2_zip`

Additionally, since Coconut also overrides some Python 3 built-ins for optimization purposes, those can be retrieved by prefixing them with `py3_`. The overwritten built-ins available are:

- `py3_map`
- `py3_zip`

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

- destructuring assignment with `*`s (use Coconut pattern-matching instead),
- function type annotation,
- the `nonlocal` keyword,
- keyword class definition,
- `@` as matrix multiplication (requires `--target 3.5`),
- `async` and `await` statements (requires `--target 3.5`), and
- formatting `f` strings (requires `--target 3.6`).

3.2.6 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 certain Python-3-specific syntax is allowed, detailed below. 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`, `26` (will work on any Python `>= 2.6` but `< 3`),
- `27` (will work on any Python `>= 2.7` but `< 3`),
- `3`, `32` (will work on any Python `>= 3.2`),
- `33`, `34` (will work on any Python `>= 3.3`),
- `35` (will work on any Python `>= 3.5`),
- `36` (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 `2.7` is equivalent to the target `27`.

3.2.7 `--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` Coconut just shows a Warning),
- use of the Python-style `lambda` statement,
- use of `u` to denote Unicode strings,
- use of *reserved variables* (without `--strict` Coconut just shows a Warning),
- use of backslash continuations (implicit continuations are preferred), and
- trailing whitespace at the end of lines.

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.

3.2.8 IPython/Jupyter Support

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

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, adding the `%coconut` and `%%coconut` magics. 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.

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. 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.

3.3 Operators

3.3.1 Lambdas

Coconut provides the simple, clean `->` operator as an alternative to Python's `lambda` statements. The operator has the same precedence as the old statement.

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))
```

3.3.2 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. It has the same precedence as subscription.

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

Python

```
import functools
expnums = map(functools.partial(pow, 2), range(5))
print(list(expnums))
```

3.3.3 Pipeline

Coconut uses pipe operators for pipeline-style function application. All the operators have a precedence in-between infix calls 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
```

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)))
```

3.3.4 Compose

Coconut uses the `..` operator for function composition. It has a precedence in-between subscription and exponentiation. The in-place operator is `..=`.

Example

Coconut

```
fog = f..g
```

Python

```
# unlike this simple lambda, .. produces a pickleable object
fog = lambda *args, **kwargs: f(g(*args, **kwargs))
```

3.3.5 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):
    return (0,) :: 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.

3.3.6 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. It has the same precedence as subscription.

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` function*).

Coconut's iterator slicing is very similar to Python's `itertools.islice`, but unlike `itertools.islice`, Coconut's iterator slicing supports negative indices, and is optimized to play nicely with custom or built-in sequence types as well as Coconut's `map`, `zip`, `range`, and `count` objects, only computing the elements of each that are actually necessary to extract the desired slice. This behavior can also be extended to custom objects if they define their `__getitem__` method lazily and set `__coconut_is_lazy__` to `True`.

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.

3.3.7 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/)                    => "//"
 (\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)
```

3.4 Keywords

3.4.1 `data`

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` blocks create immutable classes derived from `collections.namedtuple` and made immutable with `__slots__`. Coconut data statement syntax looks like:

```
data <name>(<args>):
    <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, and `<body>` contains the data type's methods.

Subclassing `data` types can be done easily by inheriting from them in a normal Python `class`, although to make the new subclass immutable, the line

```
__slots__ = ()
```

will need to be added to the subclass 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.

Example

Coconut

```
data vector(x, y):
    def __abs__(self):
        return (self.x**2 + self.y**2)**.5

v = vector(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
```

Python

```
import collections
class vector(collections.namedtuple("vector", "x, y")):
    __slots__ = ()
    def __abs__(self):
        return (self.x**2 + self.y**2)**.5

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

3.4.2 `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                        # strings
    | [pattern "as"] NAME           # capture
    | NAME "(" patterns ")"         # data types
    | "(" patterns ")"              # sequences can be in tuple form
    | "[" patterns "]"              #  or in list form
    | "(|" patterns "|)"            # lazy lists
    | "{" pattern_pairs "}"         # dictionaries
    | ["s"] "{" pattern_consts "}"  # sets
    | (                             # 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 "|)"            # lazy lists
      ) "::" pattern
    | pattern "is" exprs              # type-checking
    | pattern "and" pattern           # match all
    | pattern "or" pattern            # match any
    )
```

Semantic 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>)`), or Lazy lists (`(|<patterns>|)`): will only match a sequence (`collections.abc.Sequence`) of the same length, and will check the contents against `<patterns>`.
- Dicts (`{<pairs>}`): will only match a mapping (`collections.abc.Mapping`) of the same length, and will check the contents against `<pairs>`.
- 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>`, or `<lazy list>`): 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.

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).__eq__ |> 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(): pass
data leaf(n): pass
data node(l, r): pass
tree = (empty, leaf, node)

def depth(t):
    match tree() in t:
        return 0
    match tree(n) in t:
        return 1
    match tree(l, r) in t:
        return 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(value):
    match [x] + xs as l in value:
        return [x] + l
```

```
    else:
        raise TypeError()

[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.

Python

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

3.4.3 `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.*

3.4.4 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)
```

3.4.5 Reserved Variables

In Coconut, all variable names starting with `_coconut` are reserved. The Coconut compiler will modify and reference these variables with the assumption that the code being compiled does not modify them in any way. If your code does modify any such variables, your code is unlikely to work properly.

3.5 Expressions

3.5.1 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.

3.5.2 Implicit Partial Application

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

```
.name       =>      operator.attrgetter("name")
obj.        =>      getattr$(obj)
func$       =>      ($)$(func)
seq[]       =>      operator.__getitem__$(seq)
iter$[]     =>      # the equivalent of seq[] for iterators
```

Example

Coconut

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

Python

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

3.5.3 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()
```

3.5.4 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))
```

3.5.5 Underscore Separators

Coconut allows for one underscore between digits and after base specifiers in numeric literals. These underscores are ignored and should only be used to increase code readability.

Example

Coconut

```
10_000_000.0
```

Python

```
10000000.0
```

3.6 Function Notation

3.6.1 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
(..)        => # function composition
(.)         => (getattr)
(::)        => (itertools.chain) # will not evaluate its arguments lazily
($)         => (functools.partial)
(+)         => (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))))
```

3.6.2 Shorthand Functions

Coconut allows for shorthand in-line function definition, where the body of the function is assigned directly to the function call. The syntax for shorthand function definition is

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

where `<name>` is the name of the function, `<args>` are the functions arguments, and `<expr>` evaluates the value that the function should return.

Note: Shorthand function definition can be combined with infix and pattern-matching function definition.

Rationale

Coconut's shorthand 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))
```

3.6.3 Infix Functions

Coconut allows for infix function calling, where a function is surrounded by backticks and then can have arguments placed in front of or behind it. Backtick calling has a precedence in-between chaining and piping.

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 shorthand and 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))
```

3.6.4 Pattern-Matching Functions

Coconut supports pattern-matching / destructuring assignment syntax inside of function definition. The syntax for pattern-matching function definition is

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

where <name> is the name of the function, <cond> is an optional additional check, <body> is the body of the function, and <pattern> is defined by Coconut's *match statement*. 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. Coconut's pattern-matching function definition is equivalent to a *match statement* that looks like:

```
def <name>(*args):
    match (<pattern>, <pattern>, ...) in args:
        <body>
    else:
        err = MatchError(<error message>)
        err.pattern = "def <name>(<pattern>, <pattern>, ...):"
        err.value = args
        raise err
```

If pattern-matching function definition fails, it will raise a [`MatchError`](#matcherror) object just like *destructuring assignment*.

Note: Pattern-matching function definition can be combined with shorthand and 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 Statements

3.7.1 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

```
def last_two(l):
    _ + [a, b] = l
    return a, b

[0,1,2,3] |> last_two |> print
```

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.

3.7.2 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
```

3.7.3 `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

```
try:
    unsafe_1()
except MyError:
    handle_1()
else: try:
    unsafe_2()
except MyError:
    handle_2()
```

Python

```
try:
    unsafe_1()
except MyError:
    handle_1()
else:
    try:
        unsafe_2()
    except MyError:
        handle_2()
```

3.7.4 `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.

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)
```

3.7.5 Variable Lists

Coconut allows for the more elegant parenthetical continuation instead of the less elegant backslash continuation in `import`, `del`, `global`, and `nonlocal` statements.

Example

Coconut

```
global (really_long_global_variable_name_the_first_one,
        really_long_global_variable_name_the_second_one)
```

Python

```
global really_long_global_variable_name_the_first_one, \
        really_long_global_variable_name_the_second_one
```

3.7.6 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 Built-Ins

3.8.1 `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. 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
```

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.

3.8.2 `prepattern`

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 new patterns are checked first, then the existing. Equivalent to:

```
def prepattern(base_func):
    """Decorator to add a new case to a pattern-matching function, where the new case is checked firs
    def pattern_prepender(func):
        def pre_pattern_func(*args, **kwargs):
            try:
                return func(*args, **kwargs)
            except MatchError:
                return base_func(*args, **kwargs)
        return pre_pattern_func
    return pattern_prepender
```

Example

Coconut

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

@prepattern(factorial)
def factorial(0) = 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.

3.8.3 `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

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

Python

```
import operator
import functools
prod = functools.partial(functools.reduce, operator.__mul__)
print(prod(range(1, 10)))
```

3.8.4 `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)
```

3.8.5 `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)
```

3.8.6 `tee`

Coconut provides `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)
```

3.8.7 `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)
```

3.8.8 `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.

3.8.9 `map` and `zip`

Coconut's `map` and `zip` objects are enhanced versions of their Python equivalents that support normal slicing, optimized iterator slicing (through `__coconut_is_lazy__`), `reversed`, `len`, `repr`, and have added attributes which subclasses can make use of to get at the original arguments to the object (`map` supports `_func` and `_iters` attributes and `zip` supports the `_iters` attribute).

Example

Coconut

```
map((+), range(5), range(6)) |> len |> print
```

Python

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

3.8.10 `datamaker`

Coconut provides the `datamaker` 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__`. Equivalent to:

```
def datamaker(data_type):
    """Returns base data constructor of data_type."""
    return super(data_type, data_type).__new__$(data_type)
```

Example

Coconut

```
data trilen(h):
    def __new__(cls, a, b):
        return (a**2 + b**2)**0.5 |> datamaker(cls)
```

Python

```
import collections
class trilen(collections.namedtuple("trilen", "h")):
    __slots__ = ()
    def __new__(cls, a, b):
        return super(cls, cls).__new__(cls, (a**2 + b**2)**0.5)
```

3.8.11 `recursive`

Coconut provides a `recursive` decorator to perform tail recursion optimization on a function written in a tail-recursive style, where it directly returns all calls to itself. Do not use this decorator on a function not written in a tail-recursive style or the function will likely break.

Example

Coconut

```
@recursive
def factorial(n, acc=1):
    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")
```

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.

3.8.12 `parallel_map`

Coconut provides a parallel version of `map` under the name `parallel_map`. `parallel_map` makes use of multiple processes, and is therefore often much faster than `map`. Use of `parallel_map` requires `concurrent.futures`, which exits in the Python 3 standard library, but under Python 2 will require `python -m 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))))
```

3.8.13 `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.

3.9 Coconut Utilities

3.9.1 Syntax Highlighting

There are currently three options for Coconut syntax highlighting:

1. use SublimeText,
2. use an editor that supports Pygments, or
3. just treat Coconut as Python.

Instructions on how to set up syntax highlighting for SublimeText and Pygments are included below. If you don't like SublimeText and your chosen alternative text editor doesn't have pygments support, however, it should be sufficient to set up your editor so it interprets all `.coco` (also `.coc` and `.coconut`, although `.coco` is the preferred extension) files as Python code, as this 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 open the SublimeText command palette by entering `Ctrl+Shift+P`, enter `Package Control: Install Package`, and then `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`.

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`.

3.9.2 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. Possible values of *mode* are:

- `"exec"`: code for use in `exec` (the default)
- `"file"`: a stand-alone file
- `"single"`: a single line of code
- `"module"`: a file in a folder or module
- `"block"`: any number of lines of code
- `"eval"`: a single expression
- `"debug"`: lines of code with no header

setup

coconut.convenience.setup(***target, strict, minify, linenumbers, quiet, color***)**

If `--target`, `--strict`, `--minify`, `--linenumbers`, `--quiet`, or `--color` are desired for `parse`, the arguments to `setup` will each set the value of the corresponding flag. The possible values for each flag are:

- *target*: `None` (default), or any *allowable target*
- *strict*: `False` (default) or `True`
- *minify*: `False` (default) or `True`
- *linenumbers*: `False` (default) or `True`
- *quiet*: `False` (default) or `True`
- *color*: `None` (default) or `str`

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.

3.9.3 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

Python

```
from coconut.__coconut__ import recursive

@recursive
def recursive_func(args):
    ...
```

Coconut (coconut-lang.org) is a variant of Python built 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:

```
python -m 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:

- Help: If you're new to Coconut, Coconut's Help provides a **straightforward, easy-to-follow tutorial** of the Coconut Programming Language.
- Docs: If you're looking for info about a specific feature, Coconut's Docs provide a **complete documentation** of the language.

- 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 all general questions, concerns, or comments about anything Coconut-related, ask around at Coconut's Gitter, a GitHub-integrated chat room for Coconut developers.

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