hy-files Documentation

Release 0.1

Tuukka Turto

Jun 13, 2017

Contents:

CHAPTER 1

Introduction

Welcome to hy-files. This book explores some aspect of a exciting new language called Hy, which is a dialect of Lisp[1]. What makes Hy somewhat different is that the language is *embedded* inside other language, Python. This allows routines written in Hy call code written in Python and vice-versa.

A challenge with programming books is that while books is usually read sequentially, learning isn't sequential. There's all kinds of paths, loops and detours through the subject. Some may be faster than others, while some are more useful than others. Concepts in programming relate to each other and form a web where almost everything is connected with everything in a manner or another. This books tries to present a path through subject of programming in Hy. While it covers substantial amount of matter, even more matter had to be left out. Otherwise this book would have been an unwieldy tome that was never finished. Sometimes it refers to concepts that will be covered later in detail, but I have tried to keep that in minimum.

[1] Lisp is a whole family of languages, originating from the 1950s. Modern variants include such languages as Clojure, LFE and Racket.

CHAPTER 2

Getting Started

Fill in

CHAPTER 3

Crash Course

CHAPTER 4

Variables

CHAPTER 5

Control Structures

As the name implies, control structures are used to control which parts of the program will execute. They are used to select between different branches in code or repeat some action multiple times. Often the choice of executing a specific branch is done based on some value or combination of several values. First we'll have a look at some common predicates[1] and how to combine them. After covering them, we'll dive into various control structures and their applications.

Boolean algebra

Boolean algebra[2] is very basis of how computers operate. It consists of two values *True* and *False* and set of basic operations *and*, *or*, *not* and *xor*.

Hy adds a little twist in the Boolean algebra. Instead of operating strictly only on True and False values, it can operate on almost any value. Values that evaluate to True in Boolean algebra are considered to be *truthy* values. Conversely, values that evaluate to False in Boolean algebra are considered to be *falsey* values. Following list shows values considered falsey, all other values are considered truthy:

- None
- number 0
- any empty sequence or collection

And operator takes 0 or more arguments and returns last argument if all of them are True or no parameters were passed at all, as shown below. Notice how in the last example we're passing in three numbers and last one of them is returned as the result. This is because all of them are truthy, so the final one will be returned. Sometimes this technique can be useful, for example when you want to first check that set of variables are truthy and then perform an operation with the last one of them.

```
=> (and True True)
True
=> (and True False)
```

[1] predicate is a test that evaluates to True or False
[2] Boolean algebra, also known as Boolean logic, is named after its inventor George Boole

```
False
=> (and False False)
False
=> (and)
True
=> (and 1 2 3)
3
```

Or operator takes 0 or more arguments and returns first truthy argument if one or more of them are True. Some examples of usage are shown below. Notice how *or* without any arguments doesn't seem to return anything. This is because in that case it returns *None*, a special value denoting non-existant value (which is also considered falsey) and REPL doesn't print it on screen.

```
=> (or True True)
True
=> (or True False)
False
=> (or False False)
False
=> (or)
=> (or 1 2 3)
1
```

In order to see actual return *type*, one can use *type* as shown here:

```
=> (type (or))
<class 'NoneType'>
```

Sometimes there's need to reverse Boolean value, so that True turns to False and False turns to True. That can be achieved with *not* operator. It takes one argument and always return either True or False, as shown below. While it's possible to call *not* with truthy values, it will not be able to deduce corresponding opposite value, which is the reason why only True or False is returned.

```
=> (not True)
False
=> (not False)
True
=> (not 1)
False
=> (not [])
True
```

The final operator we're going to learn now is *xor*, short for exclusive or.

fill in xor here

Short circuiting

fill in details here

Common predicates

<, >, <=, >=, =, !=, integer?, odd?, even?

Branching

do

if, if*, if-not, when, cond, lif, lif-not, while, unless

every?

Looping

for, break, continue

while

reference to recursion

CHAPTER 6

Functions

Functions are basic building block of any Hy program. They let you to bundle up functionality into a reusable piece of code with a well defined interface and then use that function in other parts of your programs. So far we have been using functions that come with Hy and Python, but now it's time to look into how to define them by yourselves.

Defining functions

Functions are the main mean of packaging some functionality into a reusable box and giving it a name. Once declared, a function can be used over and over again in different parts of the program. It both saves typing and helps you keep your program organized. A function may accept parameters that are used to relay information into it, these can be anything data that function is supposed to process, parameters that control how function behaves and even other functions. Functions often have return value, that is the result of a function. In this sense they're akin to mathematical functions. In Hy functions can also have side-effects. These are changes that occur somewhere in the program (like setting a variable or printing on screen) or elsewhere in the computer system (like writing into a file).

Functions are defined using *defn*, as show in listing below. It defines a function *sum-if-even*, which has two formal parameters *a* and *b*. Inside of the function there's if statement that will first check if both arguments a and b are even and then add them together and return the resulting number. If either one of the arguments is not even, function simply returns 0. Defn is relatively complex tool and has several options. Next we'll take closer look on how to use them to your advantage.

```
=> (defn sum-if-even [a b]
...   (if (and (even? a)
...            (even? b))
...     (+ a b)
...     0))
=> (sum-if-even 1 2)
0
=> (sum-if-even 2 4)
6
=> (+ (sum-if-even 2 4)
...    (sum-if-even 4 4)
```

```
...    (sum-if-even 1 2))
14
```

Remember how I mentioned that functions let you to abstract away functionality behind a nicely defined interface? This actually has two facets. On the other hand, you're giving a specific name to a specific type of functionality. This lets you to think in terms of that name, instead of trying to keep track of everything that is going inside of that function. Another related matter is called variable scope. If you're defining formal parameters for your function, they're unique inside of it. It doesn't matter if you (or somebody else) has already used those names somewhere in the program. Inside of your function they're yours to do as you please, without causing mix-up somewhere else in the program. We can demonstrate this as shown below:

```
=> (defn scope-test []
...   (setv a 1
...         b 2)
...   (+ a b))
=> (setv a 10
...      b 5)
=> (scope-test)
3
=> a
10
=> b
5
```

Variables *a* and *b* are declared outside of the *scope-test* function. Variables with same names are also declared inside of the function and used to perform a calculation. But the variables declared inside the function cease to exist after the function completes. Hy (and Python) use something called *lexical scoping*, originally introduced by ALGOL. The name itself isn't that important, but the idea is. It might be worth your time to write little play programs and try out different things with variables and functions to get a good grip on this.

Functions are good for breaking down a complex problem into more manageable chunks. Instead of writing down complete instructions in one huge block how to solve the problem, you can write the basic structure or the bare essence of the problem. A hypothetical AI routine for a wizard is shown here:

```
(defn wizard-ai [wizard]
  (if (and (in-combat? wizard)
           (badly-hurt? wizard)) (cast-teleport wizard)
      (in-combat? wizard) (cast-magic-missile wizard)
      (in-laboratory? wizard) (research-spells wizard)
      (wander-around wizard)))
```

It's very simple and hopefully easy to read too. At this level, we aren't interested what kind of magical components teleport spell requires or what spell research actually means. We're just interested on breaking down the problem into more manageable pieces. In a way, we're coming up with our own language too, a language that talks about wizards and spells. And it's perfectly ok to write this part down (at least the first version), without knowing all the details of the functions we're using. Those details can be sorted out later and it might even be someone else's task to do so. Later on, we might want to add a new creature in our game and realize that we can actually use some of the functions we came up earlier as shown below. In a way we're building our own mini-language that talks about wizards, combat and spells.

```
(defn warrior-ai [warrior]
  (if (in-combat? warrior) (hit-enemy warrior)
      (badly-hurt? warrior) (find-wizard warrior)
      (wander-around warrior)))
```

Optional parameters

Sometimes you might need to write a function or method that takes several parameters that either aren't always needed or can be supplied with reasonable default. One such method is *string.rjust* that pads a string to certain length. By default a space is used, but different character will be used if supplied as show in next. In such occasions *optional parameters* are used.

```
=> (.ljust "hello" 10)
"hello     "
=> (.ljust "hello" 10 ".")
"hello....."
```

Optional parameters are declared using *&optional* keyword as shown in the example about fireballs. Parameters after optional are declared having default values that are denoted as two item lists with the parameter name being first and default value being the second element. If the default value isn't supplied (as is the case with strength in the example), None is used. Be mindful to use only immutable values as defaults. Using things like lists will lead into very unexpected results.

```
=> (defn cast [character &optional [name "fireball"] strength]
...  (if strength
...    (.join " " [character "casts" strength name])
...    (.join " " [character "casts" name])))
```

Our cast function has three parameters, out of which one (the caster) must always be given. Second parameter can defaults to *"fireball"* and third one (strenght of the spell) doesn't have default value. Inside of the function parameters are joined together to form a string that represents spell casting. There are several ways of calling the function, as shown here:

```
=> (cast "wizard")
"wizard casts fireball"

=> (cast "wizard" "lightning")
"wizard casts lightning"

=> (cast "mage" "acid cloud" "super-strong")
"mage casts super-strong acid cloud"
```

Positional parameters

Sometimes you might want to write a function that handles varying amount of parameters. One way to get around that is to define large number of optional parameters, but that is both clumsy and error prone. Also, you would have to guess maximum amount of parameters that will ever be needed and such guesses tend to go wrong.

Luckilly, there's elegant way around the problem: *positional parameters*. They allow you to define a special parameter, that holds 0 or more arguments when the function is called, depending on the amount of arguments supplied. And of course you can mix them with the regular parameters, just make sure you don't try to declare regular or optional parameters after the positional one.

Positional arguments are defined with *&rest* keyword as shown below, where a thief err.. treasure hunter collects some loot, which is defined as positional parameters.

```
=> (defn collect [character &rest loot]
...  (if loot
...    (.join " " [character "collected:"
```

```
...             (.join ", " loot)])
...     (.join " " [character "didn't find anything"])))
```

In *Working with Sequences* we'll go through some useful information for working with positional arguments. After all, they're supplied to you as a list, so things like *map*, *filter* and *reduce* might become handy. Below is excerpt of REPL session showing our little looting routing in action. As you can see, we can define a variable amount of items that the characters has found and decides to collect for the future use. In case where no positional arguments haven't been supplied, a different message is given.

```
=> (collect "tresure hunter" "diamond")
"tresure hunter collected: diamond"

=> (collect "thief" "goblet" "necklace" "purse")
"thief collected: goblet, necklace, purse"

=> (collect "burglar")
"burglar didn't find anything"
```

Higher order functions

Decorators

Recursion

tco and all that

CHAPTER 7

Data Structures

Lists

Dictionaries

Tuples

Sets

CHAPTER 8

Working with Sequences

Map, Filter, Reduce

Comprehensions

Lazy sequences

CHAPTER 9

Multimethods

CHAPTER 10

Building Pipelines for Data

Function composition

Threading macros

doto

CHAPTER 11

Reading and Writing Files

with file foo

CHAPTER 12

Classes and Objects

CHAPTER 13

Modules and Packages

CHAPTER 14

Being Friends with Python

CHAPTER 15

Macros

Bots build bots

CHAPTER 16

Version

0.1

This is the initial version of the book. It starts with a preface at *Welcome to hy-files*, explaining goal of the book, some conventions and listing handy resources for where to find more information.

After the preface, the book continues into introductions in chapter *Introduction*, explaining more about goals of the book and how it has been organized.

Boolean algebra (save for xor) is covered in chapter *Control Structures*.

Functions has basic information on how to define new functions. Different types of parameters are explained also.

CHAPTER 17

License

CHAPTER 18

Authors

Initially, this book is one person project, but I hope that it might gather enough interest and contributions from other members of Hy community. Below is up to date list of people who have contributed to the book:

- Tuukka Turto <tuukka.turto@oktaeder.net>

CHAPTER 19

What you need for this book

Computer is a good start. Hy is built using Python, so an installed Python environment is a very good thing to have. While Hy works with several different kinds of Pythons, CPython 3.5 is recommended.

A special integrated development environment isn't neccessary, simple text editor like Notepad, Notepad++, vi or such will suffice. There are ways of setting up really nice Hy development environment using emacs or vim for example, but that's beyond the scope of this book.

CHAPTER 20

Who is this book for?

Previous experience with programming isn't neccessary, as the book tries to build from ground up. The book is aimed at people who haven't programmed much (if at all) in Hy. Occasionally the book might refer to other programming languages to highlight some specific aspect of Hy, but understanding of those languages isn't needed for reading this book.

The book is aimed for people who might have some programming experience with other languages and who are curious about Hy or Lisps in general. It should reasonably work for people who don't have any programming experience at all.

CHAPTER 21

Conventions

In this book, you'll find a number of text styles used to distinguish between different types of information. Here are some examples of them and their explanation.

A block of code is set as shown in listing below. This style is often used to highlight particular concept, structure or behaviour of code.

```
(defseq fibonacci [n]
  "infinite sequence of fibonacci numbers"
  (if (= n 0) 0
      (= n 1) 1
      (+ (get fibonacci (- n 1))
         (get fibonacci (- n 2)))))
```

Different kind of notation is used for code entered in interactive mode, as shown below. Code in such an example should work when entered in interactive mode (or REPL for short). => and ... at the beginning of the lines shouldn't be entered, they're there to denote that the example can be entered in the interactive mode. Lines without => or ... show output of commands you just entered.

```
=> (defn add-up [a b]
...   (+ a b))
=> (add-up 2 3)
5
```

New terms and important words are shown as: *Important new term*.

As this is a living book, it will evolve and change over time. At the beginning it is rather empty, but it will be filled eventually. Readers are encouraged to check *Version* for information what parts have recently been added or majorly edited.

CHAPTER 22

Resources

As this book can't possibly cover everything about Hy, some more handy resources are listed here in no particular order:

Code, open issues and other matters related to development of the language can be found at GitHub.

Up to date documenation is available online at Read the Docs. Be mindful that there are several versions of the documentation online, so be sure to select the most applicable one from the menu bottom left.

There's active community on IRC[1] at #hy on freenode. This is probably the best place to start asking questions as many core developers frequent the channel.

Semi-actively discussion is also held in hylang-discuss at Google groups.

[1] Albeit slowly falling out of favor, Internet Relay Chat is still commonly used in many open source projects

CHAPTER 23

Indices and tables

- genindex
- search

Index