hy-files Documentation

Release 0.2

Tuukka Turto

Jul 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.

Source code for the book is available for anyone to view and offer their feedback or even contributions.

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.

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.

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

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.

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.

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[2] 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.

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

CHAPTER 2

Variables

Scope

fill in

Types

CHAPTER 3

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 4

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 *fn*, which creates a new function. In order to use the defined function, it usually needs to be bound to a name. This pattern is so common that a special shortcut called *defn* has been added. It takes name and definition of a function and then creates and binds the function into the name. Following two examples will have identical results:

```
(setv hello (fn [person] (print "hello " person)))

(defn hello [person]
  (print "hello " person))
```

Below is another example of using *defn*. 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. Unlike many other languages, Hy doesn't have explicit return keyword. As you can see, the value of last expression executed is also the return value of the function. 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

Higher-order functions are just ordinary functions that have functions as their formal parameters or return value (or even both). In essence, they are functions that deal with other functions, hence the name. They are useful in many situations, allowing one to write generic code that can be easily adapted to handle specific cases. For example, here is an example of making an alchemy potion. Each potion has dry *ingredients* and one or more *liquids*. Dry ingredients are simply mixed together, while liquids might need different approach depending on what kind of potion is being made. The choice of how to *prepare* liquids is left to the discretion of the alchemist and they need to supply *mix-potion* with the function that they would like to use to prepare liquids.

```
(defn mix-potion [ingredients liquids prepare]
  (setv mixture (mix ingredients))
  (setv liquid (prepare liquids))
  (combine liquid mixture))
```

Lets pretend that some other alchemist has defined different ways of preparing mixtures for us as show below:

```
(defn stir [liquids]
   ...)

(defn slosh [liquids]
   ...)

(defn carefully-mix [liquids]
   ...)
```

On a superficial level, each function looks same. They might have different names, but they have same amount of parameters and return similar things (mixture of liquids). To use them in potion making, our alchemist can do something like this:

```
(mix-potion ["pixie dust", "fly wings"]
            ["water", "juice"]
            slosh)

(mix-potion ["olive"]
            ["gin", "vermouth"]
            stir)

(mix-potion ["newt eyes", "dragon nail", "basilisk scale"]
            ["nitric acid", "hydrochloric acid"]
            carefully-mix)
```

Each of these would create a new potion, using the specified ingredients, liquids and method of combining liquids. Such way of programming lets us to write general code, which is not interested on the tiny details, but in the overall process of how to do something. While working with such code, the programmer can concentrate on problem at the hand and defer details to another time or even have somebody else to help writing them.

It is also possible to write functions that create new function when called. While it is possible to use *defn* to do so, often it is simpler to use *fn*. These functions are sometimes called anonymous, as they are not bound to a name.

To illustrate this, lets look a different kind of problem. Our friends in gnomish bank are handling deposits of customers from various different regions. While gold is easy to handle, it is the letters that are causing teller gnomes headache. Even simple things like greetings in the beginning of a letter are hard to keep in order as elfs, humans and orcs all have different customs that gnomes try to observe. In order to alleviate this problem, one particularly crafty gnome has designed an automatic letter writing system. Like everything that gnomes do, the system is very ornate and flexible. It consists of very many pieces that can be combined together in myriad ways. One such part is greeter-crafter. When given a culture, this device will construct another device which will know how to greet a person of that culture.

```
(defn greeter-crafter [culture]
  (if (= culture "elven") (fn [person]
                            (+ "The most illustrious " person.name))
      (= culture "human") (fn [person]
                            (+ "Greetings " person.name))
      (= culture "orcish") (fn [person]
                             (+ "Saluations " person.name))
      (fn [person]
        "Dear sir or madam")))
```

Heart of the routine is a case study. *culture* parameter is examined and corresponding branch of if statement is executed. There are three special cases, each corresponding to a specific culture and a generic one that is used when unknown culture is given as an argument. Each of the branches will create a new function and return it. Following piece of code highlights how greeter-crafter could be used to personalize monthly report letter.

```
(defn handle-monthly-letter [person]
  (setv greeter (greeter-crafter (culture-of person)))
  (setv letter (+ (greeter person)
                  (write-body person)
                  (in-closing person)))
  (send letter))
```

First a *greeter* is constructed by using greeter-crafter. Then a letter consisting of greeting, body of text and closing statement is crafted and finally the letter is sent. In case gnomes would like to send yearly letters too, they could reuse the greeter-crafter and would only need to create new gadget that knows how to write body of the yearly letter. And if

later a new culture would start doing business with the gnomes, they would add this culture to greeter-crafter and all different types of letters would automatically start greeting this new culture correctly.

And if gnomes would require more intricate system, nothing would stop them from creating *greeter-crafter-creator*, a device that can build greeter-crafters which know how to build greeters that know how to address members of a specific culture. Very sophisticated, intricate and maybe even confusing system.

Closures

Closures are functions accessing symbols outside their scope (we talked about scope earlier in *Scope*). When such a function is defined, it *captures* symbols that it refers to, but are outside of its scope. These symbols must have been defined in the outer scope of the function. An example will clarify this:

```
=> (defn create-adder [number]
...   (fn [n] (+ n number)))

=> (setv add-1 (create-adder 1))
=> (add-1 5)
6

=> (setv add-5 (create-adder 5))
=> (add-1 (add-5 2))
8
```

create-adder is a higher-order function (we talked about these just recently at *Higher-order functions* that takes parameter *number* and returns a new function that takes parameter *n*. When called, this new function will add *n* and *number* together. It has captured the value of *number* when it was initially created.

This useful technique can be used to cut down amount of classes (We will go over them in detail later at *Classes and Objects*, but now it is enough to know that they are a way of packaging data and functions that operate on that data together). As always, an example will hopefully clarify the idea.

A new smithy has been opened by a drawf. It is small, but has latest automated tools developed by gnomes, which helps the smith to get their work done neatly and efficiently. There's a device from creating swords, another for shoe nails and third one for iron keys:

```
(defn create-sword [] ...)

(defn create-shoenail [] ...)

(defn create-key [] ...)
```

Each of these tools create a basic item that the smith can then continue work on and customize according to their client's needs. However, as fame and client base of the smith grows, they soon find themself unable to take all the jobs that are offered to them. The smith considers hiring another smith to work for them, but that would require building a bigger smithy and splitting the profits. Instead of that, the smith asks gnomes to build them more tools for different kinds of items. The first batch of such tools is for swords only:

```
(defn create-short-sword [] ...)

(defn create-long-sword [] ...)

(defn create-claymore [] ...)
```

While the approach works, the drawf is a bit unhappy as now they have lots and lots of very specialized tools all over the smithy. What used to be nice and tidy smithy is now very cramped and untidy place. Something needs to be done before the smith accidentally steps on one of the tools that are now laying on every possible surface. Ingenious gnomes

quickly come up with a solution. They design a special sword maker machine, that can make all kinds of swords. User only needs to supply it with a dictionary (covered in more detail in *Dictionaries*) that describes what kind of sword should be created:

```
(setv short-sword { :blade-length 'short
                    :blade-width 'medium
                    :hilt 'standard })

(create-sword short-sword)
```

Business was booming and smith was really happy with his reduced amount of tools. Smithy was neat and tidy again. Sure, they had to keep track of little metal discs that contained dictionaries for preparing different kinds of items. While the smith tried to be careful and pay attention to item makers and discs, sometimes they still managed to use wrong type of device with a disc. Usually disc and device were so incompatible that nothing happened, but from time to time he ended up with tiny daggers or sword sized nails that were simply unusable. Discs were clearly labeled, but the devices were hard to keep track of. Like always, gnomes had a solution for this problem too. Each dictionary would have information that clearly indicated what kind of item it would create. And instead of multiple devices, there were only one device that was needed.

```
(setv short-sword { :type 'sword
                    :blade-length 'short
                    :blade-width 'medium
                    :hilt 'standard })

(create-item short-sword)
```

However, this omni-maker was very complex device and the smith could only afford one of them. Suddenly they had to spend lot of time waiting for the omni-device to finish, so that they could load next dictionary and start making the next item. Especially frustrating this was when multiple similar items had been ordered. But again, the gnomes has a solution. Omni-maker was modified to create not items, but devices for creating those items. This higher-order maker could then used to right tool for right job when needed and creating three similar swords was easy. They could even be engraved with the owner's name:

```
(setv sword-creator (omni-maker sword-dictionary))
(sword-creator "Pete the Adventurer")
(sword-creator "Uglak the Goblin")
(sword-creator "Jaska the Conqueror")
```

When device was no longer needed, it could be melted down in forge and used to create different device later when needed again.

In the silly example earlier, item makers were analoguous to functions. The smith started with set of specialized functions and kept adding more and more that were doing sort of similar tasks than the ones they already had. Gnomes then fixed this eventually coming up with a omni-maker, which in programming terms was higher-order function. It could create another function that performed the required task and could be reused as often as needed. The resulting function was also a closure, as it captured the dictionary passed to. We didn't look inside of these devices, but they might look something like the following code:

```
(defn omni-maker [config]
  (setv item-type (:type config))
  (if (= item-type 'sword)  (fn [engraving]
                              (setv item (new-sword))
                              (blade-length item (:blade-length config))
                              (blade-width item (:blade-width config))
                              (add-hilt (:hilt config))
                              (add-text engraving))
      (= item-type 'helmet) (fn [engraving]
```

```
                                        ..)))
```

Notice how argument passed into *config* parameter of omni-maker is later on used by anonymous function that was created by omni-maker.

Note: Closure does not create copies of values it captures, but uses them as they are. If you create closure that uses mutable variable, be extra mindful that you do not accidentally change it. Changed values will be visible to every closure using the original value.

Closures are useful when you want to have a group of functions that do a similar task, but slightly differently. In such case you can create a factory function that constructs specialized functions for you, which are using data they captured while being created. For example, a function that converts values between two different system (say, metric and imperial), could have the conversion factor fed to it by a factory function. The act of converting between two linearly related systems is always the same, regardless of the factor. You can represent the act of conversion in one part of the system and reuse it multiple times for converting between different systems.

```
=> (defn create-converter [factor]
...   (fn [value]
...     (* value factor)))

=> (setv feet-to-meters (create-converter 0.3048))
=> (feet-to-meters 5)
1.524

=> (setv kg-to-pounds (create-converter 2.2046))
=> (kg-to-pounds 5)
11.023
```

Decorators

Decorators add a whole new layer to functions, figuratively and literally.

Recursion

tco and all that

CHAPTER 5

Data Structures

Simple programs can be written that use just few variables to hold the data that they need. But as the programs grow more complex and they handle more data, single variables are not going to be enough. At that point you will need to use data structures to hold your data and keep it organized. Different data structures are suited for different kinds of tasks and there are lots of different structures, each with pros and cons. In this chapter we are going to have a look at some of the most common ones that you might need.

We will meet a new friend, apprentice filer Oseo, who has recently started working at the filing department of 4th Order. He will introduce several different ways of organizing data for us.

Mutability

Before we start exploring different kinds of data structures, we have to talk a bit about mutability. Some of the data structures covered here are mutable, while others are immutable. The difference between the two are that once you have created an immutable data structure, it can never be modified. Using immutable data structures can help you to minimize errors where program inadvertly changes some important data.

In terms of filing department, it means that after Oseo has created an filing entry, nobody is ever allowed to change it. If such information will need updating later, the only way to do so is to create a new entry and replace the old one with it.

Tuples

Our first structure is tuple. They're *ordered* (meaning that the values you put in have well defined and meaningful location within the data structure), *immutable* collection of zero or more values.

Creation

There are two ways of creating tuples: using , literal or *tuple* initializer (initializers are covered in more detail *later* when we talk about classes and objects).

The , literal takes a list of values that are placed into the tuple:

```
=> (, 1 2 3)
(, 1 2 3)

=> (, "one" 2 "III")
(, "one" 2 "III")
```

And nothing prevents you from creating a nested tuple, a tuple that has one or more tuples inside of it:

```
=> (setv address (, (, "John" "Smith") (, "Pyramid Road 1" "Cairo")))
(, (, "John" "Smith") (, "Pyramid Road 1" "Cairo"))
=> (first address)
(, "John" "Smith")
```

Todo

link to iterables here

tuple initializer take a single *iterable*, which items are used to initialize a new tuple. For example, one can take a list and use it to create a new tuple that has same elements in it:

```
=> (setv sizes ["small" "medium" "large"])
=> (tuple sizes)
(, "small" "medium" "large")
```

Common operations

While it is not possible to modify tuple once it has been created, there are still several other operations that can be done. Next we are going to look at some of the most common ones.

Just creating tuples is not going to be that useful, we want a way to get back the data that was originally put in. For that we have three operations at our disposal: *first*, *second* and *get*.

```
=> (setv data (, "quick" "brown" "fox"))
(, "quick" "brown" "fox")
=> (first data)
"quick"
=> (second data)
"brown"
=> (get data 2)
"fox"
```

Notice how *get* uses zero based indexing. Number of the first element is 0, second one is 1 and third one is 2 (and so on). Using negative indexes is possible too. In that case counting starts from the end of the tuple:

```
=> (get data -1)
"fox"
=> (get data -2)
"brown"
```

If you have two tuples, you can generally compare if they are equal or not (I say generally, because if they contain things that can not be compared, you can not compare two tuples). For the comparison, old friends of = and *!*= are used (we talked about them earlier in *Common predicates*).

```
=> (= (, 1 2 3) (, 1 2 3))
true
=> (!= (, 1 2 3) (, 1 2 3))
false
```

There are few functions for examining contents of a tuple. You can check if an element is in tuple using *in*, how many times certain element is in tuple using *.count* (note the dot at the beginning, it is significant as it denotes that count is a *method* of tuple. We will cover these in more detail when we talk about *methods*) or you can find first index the item appears in tuple using *.index*. Finally, to count how many items are in tuple, we use *len*.

```
=> (setv data (, 3 1 4 1 5 9 2 6))
=> (len data)
8
=> (in 5 data)
true
=> (.index data 5)
4
=> (.count data 1)
2
```

Finally, there are two ways of building larger tuples from smaller one: + creates a new tuple that has all the elements of two other tuples combined and * is used to create new tuple that has items of the original tuple repeatedly:

```
=> (+ (, 1 2 3) (, 4 5 6))
(, 1 2 3 4 5 6)
=> (* 2 (, 1 2))
(, 1 2 1 2)
```

Oseo's work

Within filing department, Oseo usually uses tuples for storing small pieces of data. Since they has to remember what each data in given index stands for, having large structures can be error prone. But for things like coordinates, measurements (value and unit together) and such tuples are good.

```
=> (defn make-measurement [value unit]
...   (, value unit))

=> (set stick-length (make-measurement 1.2 "meters"))
=> (first stick-length)
1.2
=> (second stick-length)
"meters"
```

Sometimes Oseo's work was related to mailing. Not the actual carrying the letters and parcels around though, but marking them based on which priority they were. Parcels were then moved forward to a sorting section, where they would be dispatched to correct direction, depending on the label Oseo had used.

```
(defn attach-label [label parcel]
  (, label parcel))

(defn set-priority [parcel]
  (if (paid-extra? parcel) (attach-label 'priority parcel)
      (vip-customer? parcel) (attach-label 'priority parcel)
      (no-postage-paid? parcel) (attach-label 'snail parcel)
      (attach-label 'regular parcel)))
```

```
(defn dispatch-by-priority [tagged-parcel]
  (setv (, speed parcel) tagged-parcel)
  (if (= speed 'priority) (send-immediately parcel)
      (= speed 'regular) (send-evening parcel)
      (send-later parcel)))
```

Notice how *dispatch-by-priority* deftly takes apart tuple of two items and assigns them to local variables with one *setv*. This is called destructuring and is useful technique. It allows you to take a tuple or list (which we will cover a bit later in *Oseo's work*) and assign each individual element to a variable. Amount of symbol names in first tuple have to match to the amount of elements in the second one, otherwise an error will be reported.

In some rare cases already labeled parcel is deemed to be more important or less important than what Oseo originally labeled it. Since labeling can not be altered after the creation, a completely new label has to be created that has new priority, but the old parcel. In such a case simply extracting the tuple and creating a new one is all that needs to be done.

```
(defn relabel-parcel [label tagged-parcel]
  (setv (, old-label parcel) tagged-parcel)
  (, label parcel))
```

Named tuples

Named tuples are quite similar to regular ones and offer all the same tools (and then some more). The advantage of named tuples over regular ones is that their fields can have descriptive names. Instead of having to remember what data is stored in which index, the programmer only has to remember what name fields have been given. This also makes it easier to read code written by someone else, as you do not have to guess what is stored in index 2 for example.

Named tuples are not available by default, so you have to import the appropriate function as described in modules-and-packages-importing. This is needed at the beginning of every file where you want to use *namedtuple* function and in REPL before you use the function:

```
=> (import [collections [namedtuple]])
```

First step is to create a *type* to represent the specific named tuple and store it somewhere for future use:

```
(setv Measurement (namedtuple "Measurement" ["value" "unit"]))
```

After the type has been created, it can be used just like any other ordinary type. Attributes that we specified earlier are available and can be accessed:

```
=> (setv box-width (Measurement 5 "meters"))
=> (. box-width unit)
"meters"
=> (. box-width value)
5
```

Todo

keyword parameters link here

One can use keyword parameters to make creation of a tuple clear:

```
=> (setv box-height (Measurement :unit "meters" :value 2)
=> (. box-height value)
2
```

Indexed access and deconstructing works just like with regular tuples:

```
=> (first box-width)
5
=> (setv (, measurement unit) box-width)
(5, "meters")
=> unit
"meters"
```

Oseo's work

Oseo was really delighted when he learned about named tuples. No more second guessing what was stored in the 3rd element of a tuple. Names would help him and his colleaques at the filing department to keep things in neat order from now on. Instead of fixing the old labeling system for parcels (after all, it was working just fine), he was tasked with a new problem: interdepartment mail system.

Filing department was only one of the many departments of the 4th Order and sometimes they needed to communicate with each other. The problem was that the 4th Order had grown very organically, sprouting new departments here and there and was generally really convoluted mess. All communications between the departments was rather ad hoc solutions and it took considerable amount of time and energy to even track down how to send message to a specific department. There were just so many different systems in use.

To alleaviate the problem, Oseo decided that there needs to be one central location within filing department, where one can simply drop their message in a tiny metal cylinder, that then gets delivered to destination with appropriate means:

```
(setv Message (namedtuple "Message" [recipient department message-text]))

(create-message [recipient department text]
  (Message :recipient recipient
           :department department
           :message-text text))

(drop-off-message [message]
  (setv wrapped (wrap-message message))
  (dispatch-message wrapped))
```

Because there are many different ways of delivering message, Oseo built a system that wraps original message inside another metal tube. This metal tube is constructed specifically to the specifications of the target department. Some prefer plain tube, some require intricate carvings. They all have slightly different expectations on how addresses are written even. But all these details are taken care by `wrap-message` function.

`dispatch-message` is responsible for routing wrapped message to correct direction. It does this by examining wrapped message, which contains information what kind of system should be used in routing:

```
(defn dispatch-message [message]
  (setv system (. message routing-method))
  (if (= system 'pressure-tube) (route-pressure-tube message)
      (= system 'pigeon) (route-pigeon message)
      (= system 'courier) (route-courier message))
      (manual-routing message))
```

Each and every routing system has unique set of data that they need for correctly routing the wrapped message:

```
(defn route-pressure-tube [message]
  (setv tube (. message tube))
  (setv pressure (if (= (. message priority) 'high)
                     'extra-pressure
                     'regular-pressure))
  (insert-message tube message)
  (set-pressure tube pressure))

(defn route-pigeon [message]
  (setv destination (. message district))
  (setv pigeon-house (select-house destination))
  (setv pigeon (take-pigeon pigeon-house))
  (attach-message pigeon)
  (release-pigeon pigeon))
```

Here is a little trick that Oseo came up with: Since every wrapped message has some common information and some information specific to the routing method, Oseo defined multiple types of named tuples for the task. This way wrapped messages sent via pressure tube do not have to know anything about carrier pigeons and vice versa. Oseo anticipates that new methods will be added in the future and wanted to build a system that is easy enough to extend at that point:

```
(setv PigeonMessage (namedtuple "PigeonMessage"
                                ["routing-method" "district"]))

(setv TubeMessage (namedtuble "TubeMessage"
                              ["routing-method" "tube" "priority"]))
```

Sets

Oseo's work

Lists

Dictionaries

Oseo's work

CHAPTER 6

Classes and Objects

Initializing object

Methods

CHAPTER 7

Version

Current version

0.2

Version 0.2 is currently being worked on and list of major changes will be added here as they are being done.

- restructure index and introduction
- higher-order functions, closures

Past versions

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 8

License

CHAPTER 9

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 10

List of things to do

This is a laundry list of things needing to be addressed.

Todo

link to iterables here

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/hy-files/checkouts/latest/doc/source/data_structures.rst, line 74.)

Todo

keyword parameters link here

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/hy-files/checkouts/latest/doc/source/data_structures.rst, line 270.)

CHAPTER 11

Indices and tables

- genindex
- search

Index