Rust for Python Programmers

written on Wednesday, May 27, 2015

Now that Rust 1.0 is out and quite stable, I thought it might be interesting to write an introduction to Rust for Python programmers. This guide goes over the basics of the language and compares different constructs and how they behave.

Rust language wise is a completely different beast compared to Python. Not just because one is a compiled language and the other one is interpreted, but also because the principles that go into them are completely different. However as different as the languages might be at the core, they share a lot in regards to ideas for how APIs should work. As a Python programmer a lot of concepts should feel very familiar.

Syntax

The first difference you will notice as a Python programmer is the syntax. Unlike Python, Rust is a language with lots of curly braces. However this is for a good reason and that is that Rust has anonymous functions, closures and lots of chaining that Python cannot support well. These features are much easier to understand and write in a non indentation based language. Let's look at the same example in both languages.

First a Python example of printing “Hello World” three times:

```
def main():
    for count in range(3):
        print "{}. Hello World!".format(count)
```

And here is the same in Rust:

```
fn main() {
    for count in 0..3 {
        println!("{}. Hello World!", count);
    }
}
```

As you can see, quite similar. `def` becomes `fn` and colons become braces. The other big difference syntax wise is that Rust requires type information for parameters to function which is not something you do in Python. In Python 3 type annotations are available which share the same syntax as in Rust.

One new concept compared to Python are these functions with exclamation marks at the end. Those are macros. A macro expands at compile time into something else. This for instance is used for string formatting and printing because this way the compiler can enforce correct format strings at compile time. It does not accidentally happen that you mismatch the types or number of arguments to a print function.

Traits vs Protocols

The most familiar yet different feature is object behavior. In Python a class can opt into certain behavior by implementing special methods. This is usually called "conforming to a protocol". For instance to make an object iterable it implements the `__iter__` method that returns an iterator. These methods must be implemented in the class itself and cannot *really* be changed afterwards (ignoring monkeypatching).

In Rust the concept is quite similar but instead of special

methods, it uses traits. Traits are a bit different in that they accomplish the same goal but the implementation is locally scoped and you can implement more traits for types from another module. For instance if you want to give integers a special behavior you can do that without having to change anything about the integer type.

To compare this concept let's see how to implement a type that can be added to itself. First in Python:

```
class MyType(object):

    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        if not isinstance(other, MyType):
            return NotImplemented
        return self.__class__(self.value + other.value)
```

And here is the same in Rust:

```
use std::ops::Add;

struct MyType {
    value: i32,
}

impl MyType {
    fn new(value: i32) -> MyType {
        MyType { value: value }
    }
}

impl Add for MyType {
    type Output = MyType;

    fn add(self, other: MyType) -> MyType {
        MyType { value: self.value + other.value }
    }
}
```

Here the Rust example looks a bit longer but it also comes with automatic type handling which the Python example does

not do. The first thing you notice is that in Python the methods live on the class, whereas in Rust the data and the operations live independently. The `struct` defines the data layout and the `impl MyType` define methods the type itself has, whereas `impl Add for MyType` implements the `Add` trait for that type. For the `Add` implementation we also need to define the result type of our add operations, but we avoid the extra complexity of having to check the type at runtime like we have to do in Python.

Another difference is that in Rust the constructor is explicit whereas in Python it's quite magical. When you create an instance of an object in Python it will eventually call `__init__` to initialize the object, whereas in Rust you just define a static method (by convention called `new`) which allocates and constructs the object.

Error Handling

Error handling in Python and Rust is completely different. Whereas in Python errors are thrown as exceptions, errors in Rust are passed back in the return value. This might sound strange at first but it's actually a very nice concept. It's pretty clear from looking at a function what error it returns.

This works because a function in Rust can return a *Result*. A *Result* is a parametrized type which has two sides: a success and a failure side. For instance `Result<i32, MyError>` means that the function either returns a 32bit integer in the success case or `MyError` if an error happens. What happens if you need to return more than one error? This is where things differ from a philosophical point of view.

In Python a function can fail with any error and there is nothing you can do about that. If you ever used the Python "requests" library and you caught down all request exceptions and then got annoyed that SSL errors are not caught by this,

you will understand the problem. There is very little you can do if a library does not document what it returns.

In Rust the situation is very different. A function signature includes the error. If you need to return two errors then the way to do this is to make a custom error type and to convert internal errors into a better one. For instance if you have an HTTP library and internally it might fail with Unicode errors, IO errors, SSL errors, what have you, you need to convert these errors into one error type specific to your library and users then only need to deal with that. Rust provides error chaining that such an error can still point back to the original error that created it if you need to.

You can also at any point use the `Box<Error>` type which any error converts into, if you are too lazy to make your own custom error type.

Where errors propagate invisibly in Python, errors propagate visibly in Rust. What this means is that you can see whenever a function returns an error even if you chose to not handle it there. This is enabled by the `try!` macro. This example demonstrates this:

```
use std::fs::File;

fn read_file(path: &Path) -> Result<String, io::Error> {
    let mut f = try!(File::open(path));
    let mut rv = String::new();
    try!(f.read_to_string(&mut rv));
    Ok(rv)
}
```

Both `File::open` and `read_to_string` can fail with an IO error. The `try!` macro will propagate the error upwards and cause an early return from the function and unpack the success side. When returning the result it needs to be wrapped in either `Ok` to indicate success or `Err` to indicate failure.

The `try!` macro invokes the `From` trait to allow conversion of

errors. For instance you could change the return value from `io::Error` to `MyError` and implement a conversion from `io::Error` to `MyError` by implementing the `From` trait and it would be automatically invoked there.

Alternatively you can change the return value from `io::Error` to `Box<Error>` and any error can be returned. This way however you can only reason about errors at runtime and no longer at compile time.

If you don't want to handle an error and abort the execution instead, you can `unwrap()` a result. That way you get the success value and if the result was an error, then the program aborts.

Mutability and Ownership

The part where Rust and Python become completely different languages is the concept of mutability and ownership. Python is a garbage collected language and as a result pretty much everything can happen with the objects at runtime. You can freely pass them around and it will "just work". Obviously you can still generate memory leaks but most problems will be resolved for you automatically at runtime.

In Rust however there is no garbage collector, yet the memory management still works automatically. This is enabled by a concept known as ownership tracking. All things you can create are owned by another thing. If you want to compare this to Python you could imagine that all objects in Python are owned by the interpreter. In Rust ownership is much more local. Function calls can have a list of objects in which case the objects are owned by the list and the list is owned by the function's scope.

More complex ownership scenarios can be expressed by lifetime annotations and the function signatures. For instance in the case of the `Add` implementation in the previous example

the receiver was called `self` like in Python. However unlike in Python the value is "moved" into the function whereas in Python the method is invoked with a mutable reference. What this means is that in Python you could do something like this:

```
leaks = []

class MyType(object):
    def __add__(self, other):
        leaks.append(self)
        return self

a = MyType() + MyType()
```

Whenever you add an instance of *MyType* to another object you also leak out self to a global list. That means if you run the above example you have two references to the first instance of `MyType`: one is in `leaks` the other is in `a`. In Rust this is impossible. There can only ever be one owner. If you would append `self` to `leaks` the compiler would "move" the value there and you could not return it from the function because it was already moved elsewhere. You would have to move it back first to return it (for instance by removing it from the list again).

So what do you do if you need to have two references to an object? You can borrow the value. You can have an unlimited number of immutable borrows but you can only ever have one mutable borrow (and only if no immutable borrows were given out).

Functions that operate on immutable borrows are marked as `&self` and functions that need a mutable borrow are marked as `&mut self`. You can only loan out references if you are the owner. If you want to move the value out of the function (for instance by returning it) you cannot have any outstanding loans and you cannot loan out values after having moved ownership away from yourself.

This is a big change in how you think about programs but you

will get used to it.

Runtime Borrows and Mutible Owners

So far pretty much all this ownership tracking was verified at compile time. But what if you cannot verify ownership at compile time? There you have multiple options to your disposal. One example is that you can use a mutex. A mutex allows you to guarantee at runtime that only one person has a mutable borrow to an object but the mutex itself owns the object. That way you can write code that access the same object but only ever once thread can access it at the time.

As a result of this this also means that you cannot accidentally forget to use a mutex and cause a data race. It would not compile.

But what if you want to program like in Python and you can't find an owner for memory? In that case you can put an object into a referenced counted wrapper and loan it out at runtime this way. That way you get very close to Python behavior just that you can cause cycles. Python breaks up cycles in it's garbage collector, Rust does not have an equivalent.

To show this in a better way, let's go with a complex Python example and the Rust equivalent:

```
from threading import Lock, Thread

def fib(num):
    if num < 2:
        return 1
    return fib(num - 2) + fib(num - 1)

def thread_prog(mutex, results, i):
    rv = fib(i)
    with mutex:
        results[i] = rv
```

```
def main():
    mutex = Lock()
    results = {}

    threads = []
    for i in xrange(35):
        thread = Thread(target=thread_prog, args=(mutex, results, i))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    for i, rv in sorted(results.items()):
        print "fib({}) = {}".format(i, rv)
```

So what we do here is spawn 35 threads and make them compute in a very terrible manner increasing Fibonacci numbers. Then we join the threads and print the sorted results. One thing you immediately notice here is that there is no intrinsic relationship between the mutex (the lock) and the results array.

Here is the Rust example:

```
use std::sync::{Arc, Mutex};
use std::collections::BTreeMap;
use std::thread;

fn fib(num: u64) -> u64 {
    if num < 2 { 1 } else { fib(num - 2) + fib(num - 1) }
}

fn main() {
    let locked_results = Arc::new(Mutex::new(BTreeMap::new()));
    let threads : Vec<_> = (0..35).map(|i| {
        let locked_results = locked_results.clone();
        thread::spawn(move || {
            let rv = fib(i);
            locked_results.lock().unwrap().insert(i, rv);
        })
    }).collect();
    for thread in threads { thread.join().unwrap(); }
    for (i, rv) in locked_results.lock().unwrap().iter() {
        println!("fib({}) = {}", i, rv);
```

```
    }
}
```

The big differences to the Python version here is that we use a B-tree map instead of a hash table and we put that into an Arc'ed mutex. What's that? First of all we use a B-tree because it sorts automatically which is what we want here. Then we put it into a mutex so that we can at runtime lock it. Relationship established. Lastly we put it into an Arc. An Arc reference counts what it encloses. In this case the mutex. This means that we can make sure the mutex gets deleted only after the last thread finished running. Neat.

So here is how the code works: we count to 20 like in Python, and for each of those numbers we run a local function. Unlike in Python we can use a closure here. Then we make a copy of the Arc into the local thread. This means that each thread sees it's own version of the Arc (internally this will increment the refcount and decrement automatically when the thread dies). Then we spawn the thread with a local function. The `move` tells us to move the closure into the thread. Then we run the Fibonacci function in each thread. When we lock our Arc we get back a result we can *unwrap* and the insert into. Ignore the *unwrap* for a moment, that's just how you convert explicit results into panics. However the point is that you can only ever get the result map when you unlock the mutex. You cannot accidentally forget to lock!

Then we collect all threads into a vector. Lastly we iterate over all threads, join them and then print the results.

Two things of note here: there are very few visible types. Sure, there is the *Arc* and the Fibonacci function takes unsigned 64bit integers, but other than that, no types are visible. We can also use the B-tree map here instead of a hashtable because Rust provides us with such a type.

Iteration works exactly the same as in Python. The only

difference there is that in Rust in this case we need to acquire the mutex because the compiler cannot know that the threads finished running and the mutex is not necessary. However there is an API that does not require this, it's just not stable yet in Rust 1.0.

Performance wise pretty much what you expect would happen. (This example is intentionally terrible just to show how the threading works.)

Unicode

My favorite topic: Unicode :) This is where Rust and Python differ quite a bit. Python (both 2 and 3) have a very similar Unicode model which is to map Unicode data against arrays of characters. In Rust however Unicode strings are always stored as UTF-8. I have covered this in the past about why this is a much better solution than what Python or C# are doing (see also UCS vs UTF-8 as Internal String Encoding). What's however very interesting about Rust is how it deals with the ugly reality of our encoding world.

The first thing is that Rust is perfectly aware that operating system APIs (both in Windows Unicode and Linux non-Unicode land) are pretty terrible. Unlike Python however it does not try to force Unicode into these areas, instead it has different string types that can (within reason) convert between each other reasonably cheap. This works very well in practice and makes string operations very fast.

For the vast majority of programs there is no encoding/decoding necessary because they accept UTF-8, just need to run a cheap validation check, process on UTF-8 strings and then don't need an encode on the way out. If they need to integrate with Windows Unicode APIs they internally use the WTF-8 encoding which quite cheaply can convert to UCS2 like UTF-16 and back.

At any point can you convert between Unicode and bytes and munch with the bytes as you need. Then you can later run a validation step and ensure that everything went as intended. This makes writing protocols both really fast and really convenient. Compared this to the constant encoding and decoding you have to deal with in Python just to support `O(1)` string indexing.

Aside from a really good storage model for Unicode it also has lots of APIs for dealing with Unicode. Either as part of the language or on the excellent crates.io index. This includes case folding, categorization, Unicode regular expressions, Unicode normalization, well conforming URI/IRI/URL APIs, segmentation or just simple things as name mappings.

What's the downside? You can't do `"föo"[1]` and expect `'ö'` to come back. But that's not a good idea anyways.

As an example of how interaction with the OS works, here is an example application that opens a file in the current working directory and prints the contents and the filename:

```
use std::env;
use std::fs;

fn example() -> Result<(), Box<Error>> {
    let here = try!(env::current_dir());
    println!("Contents in: {}", here.display());
    for entry in try!(fs::read_dir(&here)) {
        let path = try!(entry).path();
        let md = try!(fs::metadata(&path));
        println!("  {} ({} bytes)", path.display(), md.len());
    }
    Ok(())
}

fn main() {
    example().unwrap();
}
```

All the IO operations use these `Path` objects that were also shown before, which encapsulate the operating system's

internal path properly. They might be bytes, unicode or whatever else the operating system uses but the can be formatted properly by calling `.display()` on them which return an object that can format itself into a string. This is convenient because it means you never accidentally leak out bad strings like we do in Python 3 for instance. There is a clear separation of concerns.

Distribution and Libraries

Rust comes with a combination of virtualenv+pip+setuptools called "cargo". Well, not entirely virtualenv as it can only work with one version of Rust by default, but other than that it works as you expect. Even better than in Python land can you depend on different versions of libraries and depend on git repositories or the crates.io index. If you get rust from the website it comes with the `cargo` command that does everything you would expect.

Rust as a Python Replacement?

I don't think there is a direct relationship between Python and Rust. Python shines in scientific computing for instance and I don't think that this is something that can Rust tackle in the nearest future just because of how much work that would be. Likewise there really is no point in writing shell scripts in Rust when you can do that in Python. That being said, I think like many Python programmers started to pick up Go, even more will start to look at Rust for some areas where they previously used Python for.

It's a very powerful language, standing on strong foundations, under a very liberal license, with a very friendly community and driving by a democratic approach to language evolution.

Because Rust requires very little runtime support it's very easy

to use via ctypes and CFFI with Python. I could very well envision a future where there is a Python package that would allow the distribution of a binary module written in Rust and callable from Python without any extra work from the developer needed.

This entry was tagged python and rust