# Rust: PyO3

The following are simplified, abridged notes taken from the PyO3 user guide and other sources. The full user guide can be found here: PyO3 User Guide

## Introduction

PyO3 is a Rust library that allows Rust code to be written in a way that can be compiled into Python modules. PyO3 is specified as a Rust dependency in `Cargo.toml`:

```toml
[lib]
name = "my_rust_python_library"

[dependencies]
pyo3 = { version = "0.20.2", features = ["extension-module"] }
```

Maturin is then used as a build system to package the Rust code to Python. Maturin is installed as a Python package using `pip` and then initialized in the target folder:

```bash
$ pip install maturin
$ cd string_sum
$ maturin init
# Then choose pyo3 bindings
```

Once the Rust code is ready to be pushed to a Python package, it can be developed and then imported in the same Python environment:

```
$ maturin develop
$ python3
>>> import my_rust_python_library
```

## Writing Rust-Python Functions

Python functions are defined with the `#[pyfunction]` attribute:

```rust
#[pyfunction]
fn double_py(x: usize) -> usize {
    return x*2;
}
```

An additional `#[pyo3]` attribute can be applied underneath to modify the properties of the function. Below is an example of how the function's Python name can be manually specified:

```
#[pyfunction]
#[pyo3(name = "double")]
fn double_py(x: usize) -> usize {
    return x*2;
}
```

The result is a Python function named `double()`.
PyO3 signatures can specify Python function properties and their usage is explained in the sections below.

### Specifying Default Arguments

Default values are simply specified by the argument names followed by their default Rust-code values:

```
#[pyfunction]
#[pyo3(signature = (name="John Doe", age=0))]
fn print_age_and_name(name: &str, age: u8) {
    println!("My name is {}, and I am {} years old", name, age);
}
```

To give an optional argument a `None` default argument, the `Option<...>` enum can be used:

```
#[pyfunction]
fn increment(x: u64, amount: Option<u64>) -> u64 {
    return x + amount.unwrap_or(1);
}
// Only `x` is required and `amount` defaults to `None` if not provided
```

If the argument should be required but accept `None` as an input, the PyO3 signature annotation must be provided:

```
#[pyfunction]
#[pyo3(signature = (x, amount))] // <-- added this line
fn increment(x: u64, amount: Option<u64>) -> u64 {
    return x + amount.unwrap_or(1);
}
// Now, both arguments are required but `amount=None` is acceptable
```

### Positional & Keyword Arguments

To allow Python `*args` or `**kwargs` to be passed into a function, they can be specified in the PyO3 signature. Positional arguments must be of type `&PyTuple` and keyword arguments of type `Option<&PyDict>`. These can be unpacked with iterators:

```
use pyo3::types::{PyTuple, PyDict};

#[pyfunction]
```

```
#[pyo3(signature = (*args, **kwargs))]
fn my_args(args: &PyTuple, kwargs: Option<&PyDict>) {
    // Unpack *args
    for arg in args.iter() {
        println!("arg: {:?}", arg);
    }

    // Unpack **kwargs
    if let Some(kw) = kwargs {
        for (key, value) in kw.iter() {
            println!("{}: {:?}", key, value);
        }
    }
}
```

## Rust-to-Python Error Handling

Rust code uses the generic `Result<T, E>` enum to propagate errors, in which the error type `E` describes the possible errors that can happen. However, PyO3 has the `PyErr` type which represents a Python exception. If a Python exception could raised, the type can be specified with `PyResult<T>`, which is simply an alias for `Result<T, PyErr>`.
Default Python exceptions, such as ValueError, can be implemented: [List of Supported PyO3 Exceptions]

```
use pyo3::exceptions::PyValueError;

#[pyfunction]
fn check_positive(x: i32) -> PyResult<()> {
    if x < 0 {
        return Err(PyValueError::new_err("x is negative"));
    } else {
        return Ok(());
    }
}
```

## Writing Rust-Python Classes

Python classes are defined with the `#[pyclass]` attribute over structs or fieldless enums. These Rust types cannot have lifetime parameters, generic parameters, and must implement `Send` regarding threading:

```
#[pyclass]
struct Animal {
    name: String,
}
```

```
#[pyclass]
enum HttpResponse {
    Ok = 200,
    NotFound = 404,
    Teapot = 418,
}
```

Methods can be specified by placing the `#[pymethods]` over implementation blocks. Constructors can be implemented with the `#[new]` attribute, which replicates Python's `__new__` method (an `__init__` equivalent is not available):

```
#[pymethods]
impl Animal {
    #[new]
    fn new(name: String) -> Self {
        return Animal { name };
    }

    fn speak(&self) -> String {
        return format!("{} spoke!", self.name);
    }
}
```

Class attributes/variables can be set on a constant or a method without any arguments, annotated with the `#[classattr]` attribute. Class variables and methods can be accessed by setting `cls` to `&PyType` and then writing `cls.getattr("...")?;`. Then result is a `PyObject` where it can then be converted to a Rust value with `.extract()`.
Class methods can be specified with the `#[classmethod]` annotation and static methods with the `#[staticmethod]` annotation:

```
use pyo3::types::PyType;

#[pymethods]
impl MyClass {
    #[classattr]
    const MY_CONST_ATTRIBUTE: &'static str = "foobar";

    #[classattr]
    fn my_attribute() -> String {    // MyClass.my_attribute = "hello"
        return "hello".to_string();
    }

    #[classmethod]
    fn cls_method(cls: &PyType) -> PyResult<String> {
        // This gets the class variable `MY_CONST_ATTRIBUTE` as a PyObject
        // then, converts it to a String
        let local_attribute = cls.getattr("MY_CONST_ATTRIBUTE")?.extract::
<String>()?;
        return Ok(local_attribute);
    }

    #[staticmethod]
    fn static_method() {
        println!("Hello!");
    }
}
```

For detailed instructions on modifying class magic methods, please refer to the PyO3 User Guide.

## Data Type Conversion

PyO3 provides tools to convert between Python and Rust types. The PyO3 User Guide provides a conversion table for simple, native data types: PyO3 Type Conversions

For example, a Python list object can be represented in PyO3 as `&PyList` and the conversion results in a Rust Vector:

```
use pyo3::types::PyList;

#[pyfunction]
fn print_pylist_values(my_list: &PyList) {
    for item in my_list.iter() {
        println!("My Vector: {:?}", item);
    }
}
```

The resulting function takes in a Python List and PyO3 converts it to a Rust Vector.

To convert Python objects to Rust structs or enums, see the sections below.

### Python Dictionaries to Rust Structs

Python dictionaries can be converted to Rust structs using the `#[derive(FromPyObject)]` attribute, specifying the fields of the dictionary to include with `#[pyo3(item)]`:

```
#[derive(Debug, FromPyObject)]
struct MyDictionary {
    #[pyo3(item)]
    A: String,
    #[pyo3(item)]
    B: String,
    #[pyo3(item)]
    C: String
}

#[pyfunction]
fn print_struct(my_dict: MyDictionary) {
    println!("{:?}", my_dict);
}
```

The resulting Python function requires a dictionary with keys 'A', 'B', 'C' to be passed in.
A short-cut to placing the `#[pyo3(item)]` over every field is to apply the `#[pyo3(from_item_all)]` attribute on the struct:

```
#[derive(Debug, FromPyObject)]
#[pyo3(from_item_all)]
struct MyDictionary {
    A: String,
    B: String,
    C: String
}
```

If the struct fields and dictionary keys are different, they can be mapped by specifying the Python key name in the PyO3 annotation:

```
#[derive(FromPyObject)]
struct MyDictionary {
    #[pyo3(item("A_in_dict"))]
    A_in_struct: String,
    #[pyo3(item("B_in_dict"))]
    B_in_struct: String,
    #[pyo3(item("C_in_dict"))]
    C_in_struct: String
}
```

Now, the resulting Python function requires a dictionary with keys 'A_in_dict', 'B_in_dict', 'C_in_dict' to be passed in and will be mapped to the corresponding Rust struct fields. This can still be used with and overrides the `#[pyo3(from_item_all)]` attribute.

### Python Objects to Rust Structs

Similar to dictionaries, Python objects with attributes can be converted to Rust structs using the `#[derive(FromPyObject)]` attribute, specifying the corresponding object's attribute name with `#[pyo3(attribute("..."))]`:

```rust
#[derive(Debug, FromPyObject)]
struct MyPythonObject {
    #[pyo3(attribute("A"))]
    A: String,
    #[pyo3(attribute("B"))]
    B: String,
    #[pyo3(attribute("C"))]
    C: String
}

#[pyfunction]
fn print_object(my_object: MyPythonObject) {
    println!("{:?}", my_object);
}
```

The resulting Python function requires an object with attributes 'A', 'B', 'C' to be passed in.

### Python Data Types to Rust Enums

For fieldless enums, PyO3 can automatically match to Rust variants:

```rust
#[derive(FromPyObject)]
enum MyEnum {
    VariantOne,
    VariantTwo,
}

#[pyfunction]
fn take_enum(value: MyEnum) -> String {
    match value {
        MyEnum::VariantOne => "Variant One".to_string(),
        MyEnum::VariantTwo => "Variant Two".to_string(),
    }
}
```

The resulting function can now take in the string argument `take_enum("VariantOne")`.

If a Python object is provided, enum fields can be mapped to the Python object's attribute names:

```rust
#[derive(FromPyObject)]
enum MyEnum {
    #[pyo3(attribute("attr_one"))]
    VariantOne,
    #[pyo3(attribute("attr_two"))]
    VariantTwo,
}
```

The resulting function can now take in a Python object with attributes 'attr_one' and 'attr_two', which will be mapped to the corresponding Rust enum variants.

PyO3 will also attempt match to variants based on the provided data type:

```
#[derive(FromPyObject)]
enum MyEnum {
    Int(usize),                     // Input is a positive int
    String(String),                 // Input is a string
    IntTuple(usize, usize),         // Input is a 2-tuple with positive ints
    StringIntTuple(String, usize), // Input is a 2-tuple with string and int
}
```

## Setting Python Modules

Python functions and classes written in Rust can be added to a module using the `#[pymodule]` attribute. The `add_function()` and `add_class()` methods are then used to add classes and methods. If desired, the module can take a custom name with an additional `$[pyo3(name = "...")]` attribute:

```
#[pymodule]
#[pyo3(name = "custom_module_name")]
fn pyo3_example(_py: Python<'_>, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(my_function, m)?)?;
    m.add_class::<MyClass>()?;
    Ok(())
}
```

A hierarchy of modules can be created by adding sub-modules to the parent module. The pattern below can serve as a model in how to do so:

```
#[pymodule]
fn parent_module(py: Python<'_>, m: &PyModule) -> PyResult<()> {
    register_child_module(py, m)?;
    Ok(())
}

fn register_child_module(py: Python<'_>, parent_module: &PyModule) ->
PyResult<()> {
    let child_module = PyModule::new(py, "child_module")?;
    child_module.add_function(wrap_pyfunction!(my_function,
child_module)?)?;
```

```
    parent_module.add_submodule(child_module)?;
    Ok(())
}
```