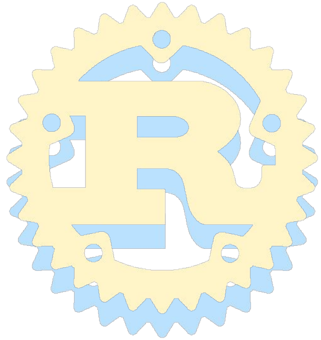
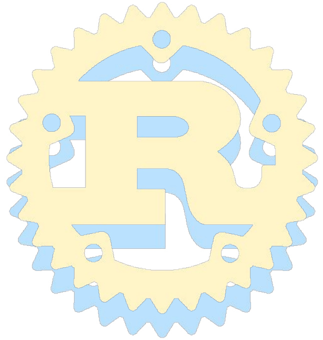
From Python to Rust

Chipy Talk



What is Rust?

- A low level language developed by Mozilla
- Relatively new (around 2007)



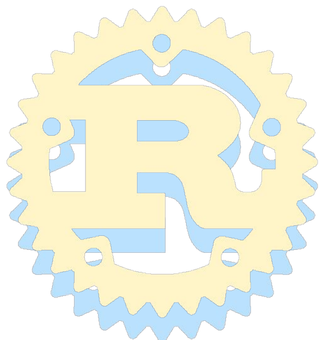
A Hello World

```python
x = 1
y = 2
print("Hello {} {}".format(x, y))
```

```
Hello 1 2
```

```rust
let x: i32 = 1;
let y: i32 = 2;
println!("Hello {} {}", x, y);
```

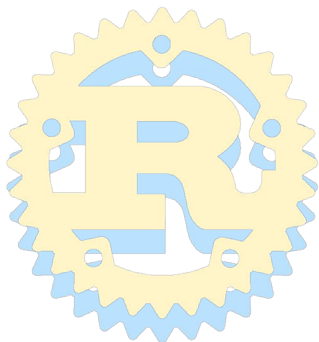
```
Hello 1 2
```



What is a low level language?

- Closer to the hardware
- Less abstraction

```
SOFTWARE (more abstraction)

 -> Python
    Go
    Java
 -> Rust
    C++
    C

HARDWARE (less abstraction)
```



Why use a low level language?

Advantages

- Faster
- Low Memory
- Explicit rather than implicit
- Maximum control
- Learning experience

Disadvantages

- More code
- Slow development time
- Use numpy / low level wrappers instead?

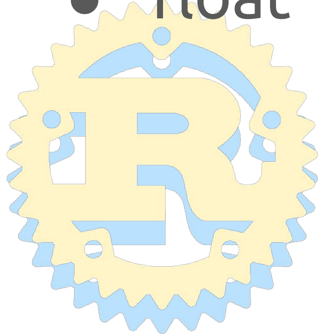
Data Types

Python

- str
- bool
- int
- float

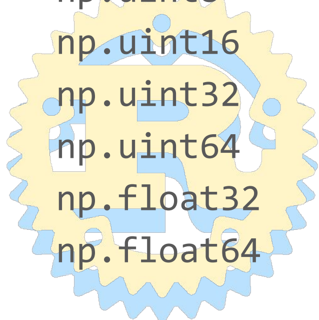
Rust

- String
- bool
- i32, i64, u32, u16, etc..
- f32, f64



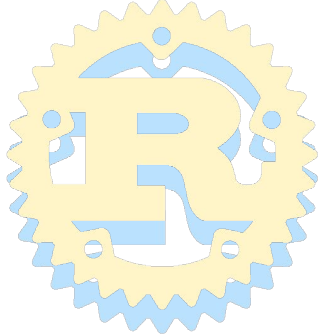
Complete List

str	String	
list	Vector	
np.array	[]	
np.charray	str	
np.int8	i8	-128 to 127
np.int16	i16	-32768 to 32767
np.int32	i32	-2147483648 to 2147483647
np.int64	i64	-9223372036854775808 to 9223372036854775807
np.uint8	u8	0 to 255
np.uint16	u16	0 to 65535
np.uint32	u32	0 to 4294967295
np.uint64	u64	0 to 18446744073709551615
np.float32	f32	
np.float64	f64	



Data Types cont.

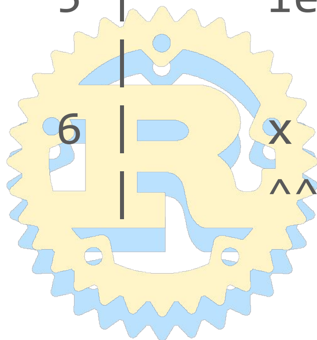
- No constants in Python
  - "We're all adults here" - Guido Van Rossum
  - Python (safety by convention)
  - Rust (enforced safety by default)
- Mutable (can be changed)
- Immutable (constant and cannot be changed)



mut Keyword

```
let x: i32 = 5;
x = 6;
```

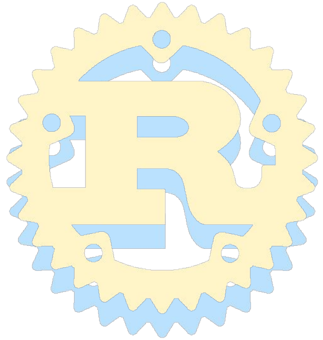
```
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:6:5
  |
5 |     let x: i32 = 5;
  |         - first assignment to `x`
6 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable
```



mut Keyword cont.

```
let mut x: i32 = 5;
```

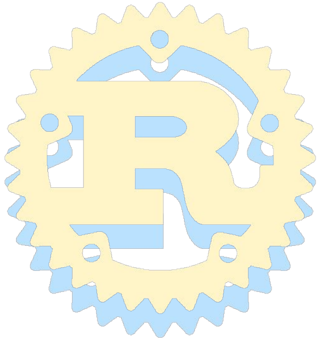
```
x = 6;
```



Lists aka Vectors

```python
items = [0, 2, 4, 6]
items.append(7)
len(items)

for value in items:
  print("The value is {}", value)
```

```rust
let mut items = vec![0, 2, 4, 6];
items.push(7);
items.len();

for value in &items {
  println!("The value is {}", value);
}
```



Functions

```python
# Python
def hello(x, y):
```

```rust
// Rust without return value
fn hello(x: i32, y:i32) {
```

```rust
// Rust with return value
fn hello2(x: i32, y:i32) -> bool {
```

pip aka cargo

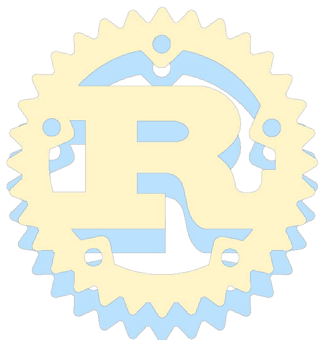
```
python hello.py

# In Rust no virtual envs and
must have requirements file
(Cargo.toml) for every project

pandas==0.9.4
portmidi==0.2.4
```

```
cargo new hello
cd hello
cargo run

[dependencies]
rand = "0.5.5"
portmidi = "0.2.4"
```

Dictionaries aka Hashes

```
id_table = {}
id_table["Alex"] = 17473
id_table["Carla"] = 13543
```

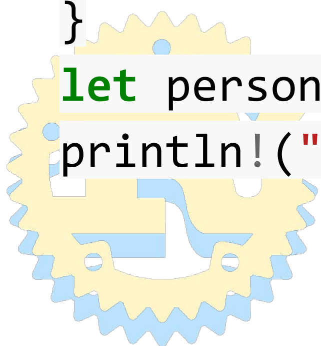
```
let mut id_table: <String, i32> = HashMap::new();
id_table.insert("Alex".to_string(), 17473);
id_table.insert("Carla".to_string(), 13543);
```



Dictionaries vs. Structs

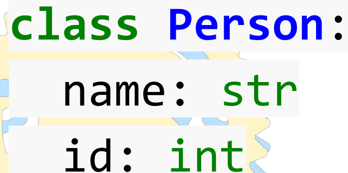
```python
person = {"name": "Alex", "id":17473}
print("The name is {}", person["name"])
```

```rust
struct Person {
 name: String,
 id: u32,
}
let person = Person {name: "Alex".to_string(), id: 17473};
println!("The name is {}", person.name);
```



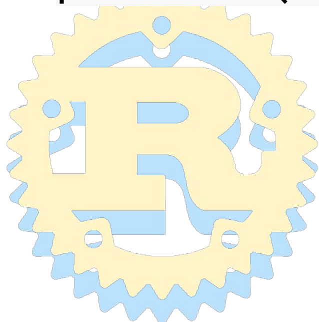
Other ways in Python

```
class Person:
  def __init__(self, name, id):
    self.name = name
    self.id = id
person = Person("Alex", 17473)

@dataclass #Will implement some methods for you wraps around class
class Person:
  name: str
  id: int
```



‘Decorators’ (a bit different)

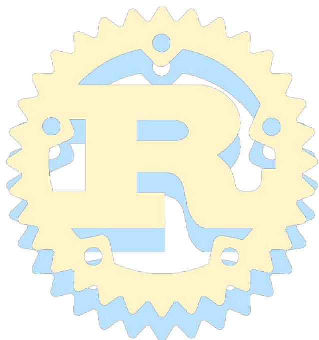
```
#[derive(Debug)] // ‘Debug’ automatically implements print method
struct Person {  // and does not wrap around functions
   name: String,
   id: u32,
}
println!("The person is {}", person);
```



‘Classes’

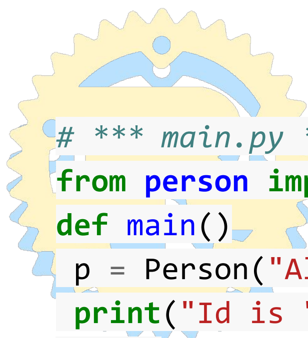
```python
class Person:
  def __init__(self, name, id):
    self.name = name
    self.id = id



  def ride_bike(self):
    print("{} is riding bike".format(self.name))
```

```rust
struct Person {

  name: String,
  id: u32,
}

impl Person {
  fn ride_bike(&self) {
    println!("{} is riding bike", self.name);
  }
}
```



Importing Modules

```python
# *** person.py ***
class Person:
 # Convention is to use
 # underscore for private
 def __init__(self, name, id):
   self._name = name
   self._id = id


def get_id(self):
   return self._id
```



```python
# *** main.py ***
from person import Person
def main()
 p = Person("Al", 3)
 print("Id is " + p.id())
main()
```

```rust
// **** person.rs ****
struct Person {
  name: String, // All variables are
  id: u32,      // private by default
}
impl Person { // Must use pub for public
  pub fn new(name: String, id: u32) -> Person {
    return Person {name: name, id: id};
  }
  pub fn get_id(&self) -> u32 {
    return self.id;
  }
}
// **** main.rs ****
mod person;
use person::Person;
fn main() { // . for value and :: for method / module
 let p = Person::new("Al".to_string(), 3);
 println!("Id is {}", p.get_id());
}
```

No exceptions in Rust!

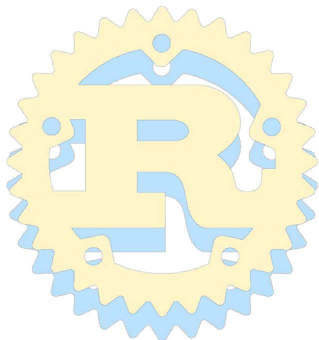
```python
def divide(x, y):
  if y == 0.0:
    raise Exception("Divide by zero!")
  else:
    return x / y

def main():
  try:
    value = divide(3.0,0.0)
    print("Value is {}".format(value))
  except Exception as e:
    print("Error oh no!")
    raise e

main()
```

```rust
// Returns Result type which contains a string if an
// error. If good then result type contains a number
fn divide(x: f32, y: f32) -> Result<f32, String> {
 if y == 0.0 {
   return Err("Divide by zero!".to_string())
  } else {
   return Ok(x / y);
  }
}
fn main() {
  let r = divide(3.0, 0.0);
  // We need to get the value from the Result
  let value: f32 = r.expect("Error oh no!");
  // The .expect() method returns the value
  // only if Ok() else prints and exits
  println!("Value is {}", value)
}
```

Python Rules for Objects

- Objects are passed by reference
- You can have multiple references
- Which is the “owner”? It's both

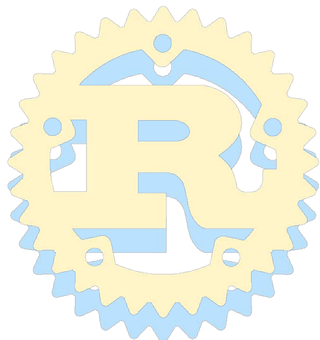
```
x = [1, 2, 3, 4, 5]
y = x
print(x)
```



Rust Rules for Objects

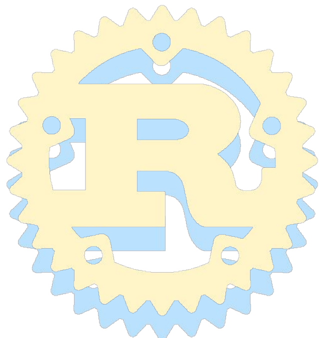
- Only one variable is the owner (one scope)
- It can be moved to different owner/scope
- Multiple references can exist, until owner goes out of scope

```
x = [1, 2, 3, 4, 5]
y = x  # x cannot be used anymore
z = &y
```



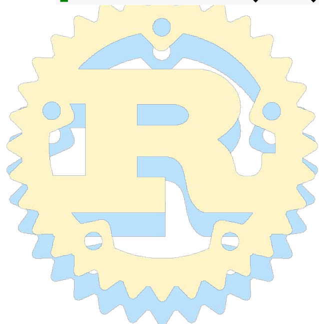
Using a variable after out of scope

```
# Pretend this is Rust code
x = [1,2,3,4,5]
write_to_file(x) # x is “moved” into write_to_file
print(x) # Compiler error here (x out of scope)
```



Always (almost) pass by reference!

```
# Pretend this is Rust code
x = [1,2,3,4,5]
write_to_file(&x) # write_to_file “borrows” x
print(x) # No more error! (x doesn’t get moved)
```



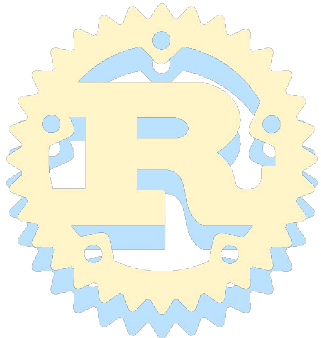
Compilers compared

Python

- Checks syntax
- Checks whitespace
- Variable names
- Generates Python bytecode

Rust

- Checks syntax
- Variable names
- Type checking
- Memory management/ errors
- Generates machine code



Why Rust instead of other low level?

Advantages

- Higher level features built into the standard library
- No garbage collection
- Memory safety at compile time

Disadvantages

- Slow compile time (complex compiler)

Q&A

- Why is the “hello”.to_string() method used instead of the string literal?
    - The to_string() converts it to String which has similar methods to Python strings. Without it, it is a str which is a fixed length array
- What happens if there is overflow of the max of an integer?
    - In most cases it will be a crash due to a runtime error, however if the compiler figures it out ahead of time, then a warning will be printed at compile time, and it will wrap around instead of crash

Further Learning

- Beginner
  - dcode's Rust Programming
  - Rust for Pythonistas
  - #rust-beginner IRC chat (mozilla)
- Intermediate
  - "The Book"
  - Rust By Example
- Expert
  - Programming Rust (O'reilly)
  - The Rustonomicon