Rusty Notes

Option Pricing - Monte Carlo

link: https://play.rust-lang.org/?gist=1ba216a0728252be16c45bd8d9d7b50d

Topics covered:

1. standard normal rv generation
2. math functions

```
extern crate rand;
use rand::Rng;
use rand_distr::StandardNormal;
use std::f32;

fn call(s: f32, x: f32) -> f32 {
    return f32::max(s - x, 0.0);
}

fn main() {
    let k: f32 = 1.0;
    let vol: f32 = 0.20;
    let rf: f32 = 0.03;
    let t = 250;

    let dt: f32 = 1.0 / 252.0;

    let mu = (rf - 0.5 * vol * vol) * dt;
    let sigma = vol * f32::sqrt(dt);

    let mut p: f32 = 0.0;
    for _i in 1..1000 {
        let logs0: f32 = 0.0;
        let mut logst: f32 = logs0;
        for _j in 1..=t {
            let z: f32 = rand::thread_rng().sample(StandardNormal);
            logst += mu + sigma * z;
        }
        p += call(f32::exp(logst), k);
    }
    p /= 1000.0;
    println!("price: {}", p);
}
```

Enums

Enums are special types that can take finite set of values, called variance of an enum.

Rust enums differ from enums in C/C++. They are similar to algebraic data types of Haskell.

To create an enum:

```
enum MyEnum {
    VALUE1,
    VALUE2,
}
let myvar = MyEnum::VALUE1;
```

Rust enum variants can take different types of data. For example,

```
enum MyEnum {
    VALUE1(String),
    VALUE2(u8),
}

let s = String::from("abc");
let u: u8 = 123;
let v1 = MyEnum::VALUE1(s);
let v2 = MyEnum::VALUE2(u);
```

Returning Custom Errors

link: https://play.rust-lang.org/?gist=1690e89f38dd1d19b726c3644ca04192

Topic: Result enum.

```
fn sqrt(a: f32) -> Result<f32, String> {
    if a >= 0.0 {
        return Ok(a.sqrt());
    }
    Err("negative number".to_string())
}

fn main() {
    match sqrt(-4.0) {
        Ok(x) => println!("{:?}", x),
        Err(s) => println!("{:?}", s),
    }
}
```

Guessing game

link: https://play.rust-lang.org/?gist=9c8840c989e884eb3fbab4698796fc45

Topics:

1. Reading from stdin
2. Parsing string to int
3. Random int generation
4. Comparing with cmp::Ordering
5. Match

```
extern crate rand;
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    let secret = rand::thread_rng().gen_range(1..=100);
    println!("guess the secret number:");
    loop {
        let mut guess = String::new();
        io::stdin()
            .read_line(&mut guess)
            .expect("error reading input");
        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        match guess.cmp(&secret) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
        println!("guess again:");
    }
}
```

Option Enum

link: https://play.rust-lang.org/?gist=ae5963045e1d2eaf501f06ea8afe24e9

Topics: Option Enum

Option Enum for a type T is defined as follows:

```
enum Option<T> {
    Some(T),
    None,
}
```

It is a type representing two possible states - valid value or none (or null). Using this type forces checking and handling of null values.

```
fn add(a: Option<i32>, b: Option<i32>) -> Option<i32> {
    if a.is_some() && b.is_some() {
        let c : i32 = a.unwrap() + b.unwrap();
        return Some(c);
    }
    None
}

fn main() {
    let a : Option<i32> = Some(5);
    let b : Option<i32> = None;
    let c = add(a, b);
    println!("{:?}", c);
}
```

Match Expression

Match is an expression, so it can be assigned to a variable. In the example below, if parse does not return an error, then s takes the value of the result of the statement block corresponding to the Ok variant. Otherwise, it takes the result of the statement block corresponding to the Err variant, in this case 99. Like function body the last statement should be without semi-colon for it to be returned.

```
fn main() {
    let guess = "123\n".to_string();
    let guess: u32 = match guess.trim().parse() {
        Ok(num) => {
            println!("well done!");
            num
        }
        Err(_) => {
            println!("bad case");
            99
        }
    };
```

```
    println!("{}", guess)
}
```

Ownership

When a variable is passed to a function, the function takes ownership of the memory referenced by the variable.

```
#[derive(Debug)]
struct Some_struct {
    x: i32,
}
fn foo(s : Some_struct) {
    println!("{:?}", s)
}
fn main() {
    let s = Some_struct{5};
    foo(s);
    foo(s);
}
```

This results in an error because the first foo owns s and the compiler does not allow s to be moved again.

One way is to use clone(). This copies the struct when passing as argument. Clone needs to be derived like Debug for custom types.

```
#[derive(Debug, Clone)]
struct Some_struct {
    x: i32,
}
fn foo(&s : Some_struct) {
    println!("{:?}", s)
}
fn main() {
    let s = Some_struct{x: 5};
    foo(s.clone());
    foo(s);
}
```

Another way is to use Copy, which is implicit.

```
#[derive(Debug, Clone, Copy)]
struct SomeStruct {
    x: i32,
}
fn foo(s : SomeStruct) {
    println!("{:?}", s)
}
```

```
fn main() {
    let s = SomeStruct{ x: 5 };
    foo(s);
    foo(s);
}
```

Copying structs for every function call may not be performant.

Third way is to use read only reference. This is a way of handing over a variable without handing over ownership. Because it is read only, one can have as many read only references as needed. Here foo() does not modify the struct, so read only reference works.

```
#[derive(Debug)]
struct SomeStruct {
    x: i32,
}
fn foo(s : &SomeStruct) {
    println!("{:?}", s.x)
}
fn main() {
    let s = SomeStruct{ x: 5 };
    foo(&s);
    foo(&s);
}
```

What if the function modifies the struct passed? We use mutable reference. Rust allows only one mutable reference, however.

```
#[derive(Debug)]
struct SomeStruct {
    x: i32,
}
fn foo(s : &SomeStruct) {
    println!("{:?}", s.x)
}
fn bar(s: &mut SomeStruct) {
    s.x = 2;
}
fn main() {
    let s = SomeStruct{ x: 5 };
    foo(&mut s);
    foo(&s);
}
```

In the example below, s is borrowed immutable by u. Then a mutable borrow occurs in the call to bar(). However, the immutable borrow of u is still in scope, as it is used in the call to foo() after. This results in a compilation error.

```
fn main() {
    let mut s = SomeStruct{ x: 5 };
    let u: &SomeStruct = &s;
    bar(&mut s);
    foo(u);
}
```

Lifetime

Lifetime is relevant only to references. Essentially making sure reference to memory does not continue to exist, when the memory has been freed.

Example:

```
#[derive(Debug)]
struct SomeStruct {
    x: i32,
}
fn foo(s : &SomeStruct) {
    println!("{:?}", s.x)
}
fn max<'a>(a: &'a SomeStruct, b: &'a SomeStruct)-> &'a SomeStruct {
    if a.x > b.x {
        a
    } else {
        b
    }
}
fn main() {
    let s = SomeStruct{ x: 5 };
    let u = SomeStruct{ x: 2 };
    let t = max(&s, &u);
    foo(t);
}
```

Lifetimes are specified with a single quote and reference variable name. Here, the annotation ensures that the return reference exists as long as both input references exist.

Struct definitions also need lifetime annotation if it has reference type fields. For example,

```
#[derive(Debug)]
struct SomeStruct<'a> {
    x: &'a i32,
}
fn foo(s : &SomeStruct) {
    println!("{:?}", s)
```

```
}
fn main() {
    let n = 3;
    let s = SomeStruct{ x: &n };
    foo(&s);
}
```

Local Variables

Inside a function, local variables are not initialized when allocated. Local variables can be used only after they have been initialized.

```
fn foo() {
    let x: u8 = 0;
    let y: f32;
    println!("{}", x);
    println!("{}", y); //error[E0381]: used binding `y` isn't initialized
}
```

If is an Expression

```
let x = 5;
let y = if x == 5 { 10 } else { 15 }; // y: i32
```

Expressions return value and in this case the last expression in the final conditional.

Ranging Over Vector

A for loop takes ownership of the vector.

```
fn main() {
    let v = vec![0, 1, 2, 3, 4];
    for i in v {
        print!("{} ", i);
    }
    println!("{}", v[0]); // error because the loop above took ownership of v
}
```

The alternative is to use a reference.

```
fn main() {
    let v = vec![0, 1, 2, 3, 4];
    for i in &v {
        print!("{} ", i);
    }
    println!("{}", v[0]);
}
```

Vector

Constructor:

```
let vec = vec![1,2,3];
let vec: Vec<i32> = vec::new();
let vec = vec![0; 5]; // vector of length 5, filled with 0.
```

Attributes:

```
vec.len();
vec.is_empty();
```

Strings

Strings are sequences of UTf-8 bytes. There are two types: &str and String. &str (string slice) is a string literal and is statically allocated i.e. stored in the binary. &str is a reference to a sequence of UTF-8 bytes.

```
let s = "hello world";
```

String type is heap allocated and constructed from &str with to_string() method.

```
let s = "hello world".to_string();
```

String type is mutable while &str is immutable.

```
let s = "hello".to_string();
s.push_str(", world!");
```

A String will coerce into &str with a &.

```
fn takes_slice(slice: &str) {
    println!("Got: {}", slice);
}
fn main() {
    let s = "Hello".to_string();
    takes_slice(&s);
}
```

However, coercion does not happen for functions that accept a trait of &str.

Strings can be concatenated with a +.

```
let s = "hello".to_string();
let t = ", world!";
println!("{}", s + t);

let u = ", world!".to_string();
println!("{}", s + &u); //&u coerces to &str
```

Because of UTF-8 encoding (i.e. each character can take up to 4 bytes), indexing a String is tricky. Indexing produces byte indexing.

```rust
// prints h e l l o
let s = "hello".to_string;
for i in s.chars() {
    print!("{} ", i);
}

// prints bytes
for i in s.as_bytes {
    print!("{}", i);
}

// gets ith character
// nth() returns an Option<&str> enum
let c = s.chars().nth(1).unwrap();
println!("{:?}", c);
```

Generics

Parametric polymorphism. Couple of frequently used examples from standard library:

```rust
enum Option<T> {
    Some(T),
    None,
}

enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

We could use any symbol.

```rust
enum Result<A, B> {
    Ok(A),
    Err(B),
}
```

Generic custom type example:

```rust
struct Point<T> {
    x: T,
    y: T,
}

let p1 = Point{x: 0, y: 0} // Point<i32>
let p2 = Point{x: 0.0, y: 0.0} // Point<f32>
```

struct fields are private by default, even if struct is marked pub.

Generic Functions

Similar to structs.

```
fn add<T>(x: T, y: T) -> T {
    x + y
}
```

If Expression

If is an expression.

```
let y = if val > 0.0 {
        println!("greater than 0");
        val
    } else {
        println!("less than 0");
        val
    };
    println!("{:?}", y);
```

Ownership

Every value is owned by a variables. And there can only be one owner. For example, in the below code, value owned by x is moved to y i.e. y takes the ownership.

When the owner goes out of scope its value will be dropped. This means when a function takes ownership of a value, it must return it for it to be used subsequently.

```
let x = String::from("hello world");
let y = x; //y owns x's value i.e. x's value moved to y.
println!("{}", x); // compile error; x is invalid after the move
```

However, x's value can be "borrowed". This is called referencing. There are two types of references: mutable and immutable.

Immutable reference is read only borrow. Since ownership integrity is not affected and all other immutable borrows refer to the same value, one can have any number of immutable references.

Only one mutable reference is allowed, since it is almost like a move.

```
let x = String::from("hello world");
let y = &x; //y borrows x's value immutably.
println!("{}", y);
println!("{}", x); // valid
let z = &x; // another immutable borrow
for c in z.chars() {
```

```
    println!("{}", c);
}
println!("{}", x); // valid
```

However, below results in compile error.

```
let mut x = String::from("hello world");
let y = &mut x; //mutable borrow of x, almost like a move
let z = &x; //immutable borrow of the same x; compile error as y can change x
println!("{}", y.to_uppercase());
println!("{}", z);
```

Stack Variables

Variables that live entirely in the stack memory are copied and not moved.

```
let x = [0; 5];
let n = 5;
let s = String::from("hello");
foo(x);
println!("{:?}, {}", x, n); // works, since x and n were copied and not moved to foo.
bar(s);
println!("{}", s); // compile error, s was moved to bar.
```

Printing Type of a variable

We can use the std::any::typename function.

```
fn print_type_of<T>(_: &T) {
    println!("{}", std::any::type_name::<T>())
}
```

println! & format! & push_str

Both borrow implicitly even if values are passed without explicit referencing.
Values are not moved.

```
let s = "abc".to_string();
println!("{}", s); //s is implicitly borrowed
let s2 = format!("{}{}", s, "def"); // s is implicitly borrowed
println!("{}", s); // works
let mut s3 = s; //s is moved and cannot be used after this
let s4 = "ghi";
s3.push_str(s4); // s4 is implicitly borrowed
s3.push_str(s4); // valid
```

Collections consume

Using a owned value to construct a vector, struct or hash map moves the value.

```rust
struct Foo {
    x: String,
    y: usize,
}

fn main() {
    let mut s = String::from("abc");
    let v = vec![s]; // s is moved to v
    s.push_str("def"); // error

    let mut s = String::from("abc");
    let mut h: HashMap<_, _> = HashMap::new();
    h.insert(s, 0); // s is moved to h
    s.push_str("def"); // error

    let mut s = String::from("abc");
    let foo = Foo { x: s, y: 0 }; // s is moved to foo
    s.push_str("def"); // error
}
```

Retrieving Values from Hash Map

The get function takes a ket value returns an Option enum. If the key does not exist, we get None.

```rust
let mut h: HashMap<_, _> = HashMap::new();
h.insert("djokovic", 1);
h.insert("federer", 2);
h.insert("nadal", 3);
println!("{:?}", h);

// get returns a mutable reference
let name = "nadal";
if let Some(rank) = h.get(&name) {
    println!("{} is seeded {}", name, rank);
} else {
    println!("{} is unseeded", name);
}
```

One can also iterate a hash map like a vector.

```rust
for (key, val) in &h {
    println!("{}: {}", key, val);
}
```

The order of output is unknown.

Updating Hash Map

If a key alreadey existings, inserting overwrites existing value.

```
h.insert("nadal", 4); // nadal's prev value of 3 is overwritten
```

The entry method provides more flexibility. It returns an Entry enum which has a or_insert() that returns a mutable reference to existing value (if it exists) or sets the value to 0 and returns a mutable reference to it.

```
// first insert
// inserts (wawrinka, 8) in h and returns mutable ref to 8
let mut v = h.entry("wawrinka").or_insert(8);

// "wawrinka" exists, so returns mutable ref to 3
let mut v = h.entry("wawrinka").or_insert(3);

// updating based on existing value
// example: counting occurences
let text = "quick brown fox jumped over the fence";
let mut map = HashMap::new();
for w in text.split_whitespace() {
    let mut count = map.entry(w).or_insert(0);
    *count += 1;
}
```