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);
}
```