DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 33

1. CLOSURES (ANONYMOUS FUNCTIONS)

2. ITERATORS

3. ITERATOR + CLOSURE MAGIC

4. HOW ABOUT PYTHON?

1. CLOSURES (ANONYMOUS FUNCTIONS)

2. ITERATORS

3. ITERATOR + CLOSURE MAGIC

4. HOW ABOUT PYTHON?

CLOSURES (ANONYMOUS FUNCTIONS)

We have seen them before in Python (as lambda functions):

```
lambda a b: a * b
```

CLOSURES (ANONYMOUS FUNCTIONS)

We have seen them before in Python (as lambda functions):

```
lambda a b: a * b
```

In Rust (with implicit or explicit type specification):

```
|a, b| a * b
|a: i32, b: i32| -> i32 {a * b}
```

CLOSURES (ANONYMOUS FUNCTIONS)

We have seen them before in Python (as lambda functions):

```
lambda a b: a * b
```

In Rust (with implicit or explicit type specification):

```
|a, b| a * b
|a: i32, b: i32| -> i32 {a * b}
```

In [2]:
```
{
  let f = |a, b| a * b;
  let x = 10;
  let y = 20;
  println!("{}",f(x,y));
};
```

```
200
```

SAMPLE APPLICATION: LAZY EVALUATION OF A VALUE

Compute a value only if needed

In [3]:
```
// What does it compute?
fn expensive_function(i:u32) -> u128 {
    if i <= 1 {
        i as u128
    } else {
        expensive_function(i-1) + expensive_function(i-2)
    }
}
```

In [4]:
```
expensive_function(44)
```

Out[4]: 701408733

SAMPLE APPLICATION: LAZY EVALUATION OF A VALUE

Compute a value only if needed

In [3]:
```
// What does it compute?
fn expensive_function(i:u32) -> u128 {
    if i <= 1 {
        i as u128
    } else {
        expensive_function(i-1) + expensive_function(i-2)
    }
}
```

In [4]:
```
expensive_function(44)
```

Out[4]: 701408733

In [5]:
```
// This function always computes expensive_function(44), even if not needed.
// Method unwrap_or takes a default value as a parameter.
fn value_or_fib44(input:Option<u128>) -> u128 {
    input.unwrap_or(expensive_function(44))
}
```

In [6]:
```
// slow
value_or_fib44(None)
```

Out[6]: 701408733

In [7]:
```
// slow
value_or_fib44(Some(123))
```

Out[7]: 123

SAMPLE APPLICATION: LAZY EVALUATION OF A VALUE

Compute a value only if needed

In [8]:
```
// This function computes expensive_function(44) only if needed.
// Method unwrap_or_else's parameter is a function that computes
// the default value, not the default value itself.
fn value_or_fib44_version_2(input:Option<u128>) -> u128 {
    input.unwrap_or_else(|| expensive_function(44))
}
```

In [9]:
```
// slow
value_or_fib44_version_2(None)
```

Out[9]: 701408733

In [10]:
```
// fast
value_or_fib44_version_2(Some(1))
```

Out[10]: 1

SAMPLE APPLICATION: LAZY EVALUATION OF A VALUE

Compute a value only if needed

In [8]:
```
// This function computes expensive_function(44) only if needed.
// Method unwrap_or_else's parameter is a function that computes
// the default value, not the default value itself.
fn value_or_fib44_version_2(input:Option<u128>) -> u128 {
    input.unwrap_or_else(|| expensive_function(44))
}
```

In [9]:
```
// slow
value_or_fib44_version_2(None)
```

Out[9]: 701408733

In [10]:
```
// fast
value_or_fib44_version_2(Some(1))
```

Out[10]: 1

- This programing pattern appears in many places.
- Another example: default value for an entry in HashMap

In [11]:
```
let mut map = std::collections::HashMap::<i32,i32>::new();
map.insert(1, 1);
*map.entry(1).or_insert_with(|| expensive_function(44) as i32) *= -1;
*map.entry(2).or_insert_with(|| expensive_function(44) as i32) *= -1;
println!("{}:{:?}    {}:{:?}",1,map.get(&1),2,map.get(&2));
```

```
1:Some(-1)    2:Some(-701408733)
```

1. CLOSURES (ANONYMOUS FUNCTIONS)

2. ITERATORS

3. ITERATOR + CLOSURE MAGIC

4. HOW ABOUT PYTHON?

ITERATORS

- provide values one by one
- method `next` provides next one
- `Some(value)` or `None` if no more available

ITERATORS

- provide values one by one
- method `next` provides next one
- `Some(value)` or `None` if no more available

Some ranges are iterators:

- `1..100`
- `0..`

First value has to be known (so `..` and `..123` are not)

ITERATORS

- provide values one by one
- method `next` provides next one
- `Some(value)` or `None` if no more available

Some ranges are iterators:

- `1..100`
- `0..`

First value has to be known (so `..` and `..123` are not)

```
In [12]: let mut iter = 1..3; // must be mutable

In [13]: iter.next()

Out[13]: Some(1)
```

ITERATOR FROM SCRATCH: IMPLEMENT TRAIT `Iterator`

In [14]:

```
struct Fib {
    current: u128,
    next: u128,
}

impl Fib {
    fn new() -> Fib {
        Fib{current: 0, next: 1}
    }
}

impl Iterator for Fib {
    type Item = u128;

    fn next(&mut self) -> Option<Self::Item> {
        let now = self.current;
        self.current = self.next;
        self.next = now + self.current;
        Some(now)
    }
}
```

ITERATOR FROM SCRATCH: IMPLEMENT TRAIT `Iterator`

In [14]:
```
struct Fib {
    current: u128,
    next: u128,
}

impl Fib {
    fn new() -> Fib {
        Fib{current: 0, next: 1}
    }
}

impl Iterator for Fib {
    type Item = u128;

    fn next(&mut self) -> Option<Self::Item> {
        let now = self.current;
        self.current = self.next;
        self.next = now + self.current;
        Some(now)
    }
}
```

In [15]:
```
let mut fib = Fib::new();
for _ in 0..10 {
    print!("{:?} ",fib.next().unwrap());
}
println!();
```

```
0 1 1 2 3 5 8 13 21 34
```

ITERATORS COME WITH MANY USEFUL FUNCTIONS IMPLEMENTED

`collect` can be used to put elements of an iterator into a vector:

In [16]:
```
let small_numbers : Vec<_> = (1..=10).collect();
small_numbers
```

Out[16]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

ITERATORS COME WITH MANY USEFUL FUNCTIONS IMPLEMENTED

`collect` can be used to put elements of an iterator into a vector:

In [16]:
```
let small_numbers : Vec<_> = (1..=10).collect();
small_numbers
```

Out[16]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

`take` turns an iterator into an iterator that provides at most a specific number of elements

In [17]:
```
let small_numbers : Vec<_> = (1..).take(10).collect();
small_numbers
```

Out[17]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

ITERATORS COME WITH MANY USEFUL FUNCTIONS IMPLEMENTED

`collect` can be used to put elements of an iterator into a vector:

```
In [16]: let small_numbers : Vec<_> = (1..=10).collect();
         small_numbers

Out[16]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

`take` turns an iterator into an iterator that provides at most a specific number of elements

```
In [17]: let small_numbers : Vec<_> = (1..).take(10).collect();
         small_numbers

Out[17]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

`cycle` creates an iterator that repeats itself forever:

```
In [18]: let cycle : Vec<_> = (1..4).cycle().take(20).collect();
         cycle

Out[18]: [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2]
```

1. CLOSURES (ANONYMOUS FUNCTIONS)

2. ITERATORS

3. ITERATOR + CLOSURE MAGIC

4. HOW ABOUT PYTHON?

ITERATOR + CLOSURE MAGIC

- Operate on entire sequence, sometimes lazily by creating a new iterator
- Allows for concise expression of many concepts

ITERATOR + CLOSURE MAGIC

- Operate on entire sequence, sometimes lazily by creating a new iterator
- Allows for concise expression of many concepts

`for_each` applies a function to each element

```
In [19]: (0..5).for_each(|x| println!("{}",x));
```

```
0
1
2
3
4
```

ITERATOR + CLOSURE MAGIC

- Operate on entire sequence, sometimes lazily by creating a new iterator
- Allows for concise expression of many concepts

`for_each` applies a function to each element

```
In [19]: (0..5).for_each(|x| println!("{}",x));
```

```
0
1
2
3
4
```

`filter` creates a new iterator that has elements for which the given function is true

```
In [20]: let not_divisible_by_3 : Vec<_> = (0..10).filter(|x| x % 3 != 0).collect();
         not_divisible_by_3
```

```
Out[20]: [1, 2, 4, 5, 7, 8]
```

ITERATOR + CLOSURE MAGIC

- Operate on entire sequence, sometimes lazily by creating a new iterator
- Allows for concise expression of many concepts

`map` creates a new iterator in which values are processed by a function

```
In [21]: let fibonacci_squared : Vec<_> = Fib::new().take(10).map(|x| x*x).collect();
         fibonacci_squared

Out[21]: [0, 1, 1, 4, 9, 25, 64, 169, 441, 1156]
```

PRIMES

`any` is true if the passed function is true on some element

Is a number prime?

In [22]:
```
fn is_prime(k:u32) -> bool {
    !(2..k).any(|x| k % x == 0)
}
```

PRIMES

`any` is true if the passed function is true on some element

Is a number prime?

In [22]:
```
fn is_prime(k:u32) -> bool {
    !(2..k).any(|x| k % x == 0)
}
```

In [23]:
```
is_prime(6)
```

Out[23]: false

PRIMES

`any` is true if the passed function is true on some element

Is a number prime?

In [22]:
```
fn is_prime(k:u32) -> bool {
    !(2..k).any(|x| k % x == 0)
}
```

In [24]:
```
is_prime(17)
```

Out[24]: true

PRIMES

`any` is true if the passed function is true on some element

Is a number prime?

In [22]:
```
fn is_prime(k:u32) -> bool {
    !(2..k).any(|x| k % x == 0)
}
```

In [25]:
```
is_prime(31)
```

Out[25]: true

PRIMES

`any` is true if the passed function is true on some element

Is a number prime?

In [22]:
```
fn is_prime(k:u32) -> bool {
    !(2..k).any(|x| k % x == 0)
}
```

In [25]:
```
is_prime(31)
```

Out[25]: true

Create infinite iterator over primes:

In [26]:
```
{
  let primes = (2..).filter(|k| !(2..*k).any(|x| k % x == 0));
  let v : Vec<_> = primes.take(20).collect();
  v
}
```

Out[26]: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]

FUNCTIONAL PROGRAMMING CLASSICS: `fold` AND `reduce`

`iterator.fold(init,f)` equivalent to

```
let mut accumulator = init;
while let Some(x) = iterator.next() {
    accumulator = f(accumulator,x);
}
accumulator
```

FUNCTIONAL PROGRAMMING CLASSICS: `fold` AND `reduce`

`iterator.fold(init,f)` equivalent to

```
let mut accumulator = init;
while let Some(x) = iterator.next() {
    accumulator = f(accumulator,x);
}
accumulator
```

Example: compute $\sum_{i=1}^{10} x^2$

```
In [27]: let sum_of_squares: i32 = (1..=10).fold(0,|a,x| a + x * x);
         sum_of_squares

Out[27]: 385
```

FUNCTIONAL PROGRAMMING CLASSICS: `fold` AND `reduce`

`iterator.fold(init,f)` equivalent to

```
let mut accumulator = init;
while let Some(x) = iterator.next() {
    accumulator = f(accumulator,x);
}
accumulator
```

Example: compute $\sum_{i=1}^{10} x^2$

In [27]:
```
let sum_of_squares: i32 = (1..=10).fold(0,|a,x| a + x * x);
sum_of_squares
```

Out[27]: 385

In [28]:
```
// Another approach: using `sum` (which can be implemented using `fold`)
let sum_of_squares: i32 = (1..=10).map(|x| x * x).sum();
sum_of_squares
```

Out[28]: 385

FUNCTIONAL PROGRAMMING CLASSICS: `fold` AND `reduce`

`iterator.reduce(f)` equivalent to

```
if let Some(x) = iterator.next() {
    let mut accumulator = x;
    while let Some(y) = iterator.next() { accumulator = f(accumulator,y}
    Some(accumulator)
} else {
    None
}
```

Differences from `fold`:

- no default value for an empty sequence
- output must be the same type as elements of input sequence
- output for length–one sequence equals the only element in the sequence

FUNCTIONAL PROGRAMMING CLASSICS: `fold` AND `reduce`

`iterator.reduce(f)` equivalent to

```
if let Some(x) = iterator.next() {
    let mut accumulator = x;
    while let Some(y) = iterator.next() { accumulator = f(accumulator,y}
    Some(accumulator)
} else {
    None
}
```

Example: computing the maximum number in $\{x^2 \bmod 7853 : x \in [123]\}$

```
In [29]: (1..=123).map(|x| (x*x) % 7853).reduce(|x,y| x.max(y)).unwrap()

Out[29]: 7744
```

FUNCTIONAL PROGRAMMING CLASSICS: `fold` AND `reduce`

`iterator.reduce(f)` equivalent to

```
if let Some(x) = iterator.next() {
    let mut accumulator = x;
    while let Some(y) = iterator.next() { accumulator = f(accumulator,y}
    Some(accumulator)
} else {
    None
}
```

Example: computing the maximum number in $\{x^2 \bmod 7853 : x \in [123]\}$

```
In [29]: (1..=123).map(|x| (x*x) % 7853).reduce(|x,y| x.max(y)).unwrap()

Out[29]: 7744

In [30]: // in this case one can use the builtin `max` method (which can be implemented, using `fold`)
         (1..=123).map(|x| (x*x) % 7853).max().unwrap()

Out[30]: 7744
```

Combining two iterators: `zip`

- Returns an iterator of pairs
- The length is the minimum of the lengths

COMBINING TWO ITERATORS: `zip`

- Returns an iterator of pairs
- The length is the minimum of the lengths

In [31]:
```
let v: Vec<_> = (1..10).zip(11..20).collect();
v
```

Out[31]:
```
[(1, 11), (2, 12), (3, 13), (4, 14), (5, 15), (6, 16), (7, 17), (8, 18), (9, 19)]
```

COMBINING TWO ITERATORS: `zip`

- Returns an iterator of pairs
- The length is the minimum of the lengths

In [31]:
```
let v: Vec<_> = (1..10).zip(11..20).collect();
v
```

Out[31]:
```
[(1, 11), (2, 12), (3, 13), (4, 14), (5, 15), (6, 16), (7, 17), (8, 18), (9, 19)]
```

Inner product of two vectors:

In [32]:
```
let x: Vec<f64> = vec![1.1,  2.2, -1.3,  2.2];
let y: Vec<f64>  = vec![2.7, -1.2, -1.1, -3.4];
let inner_product: f64 = x.iter().zip(y.iter()).map(|(a,b)| a * b).sum();
inner_product
```

Out[32]:
```
-5.72
```

1. CLOSURES (ANONYMOUS FUNCTIONS)

2. ITERATORS

3. ITERATOR + CLOSURE MAGIC

4. HOW ABOUT PYTHON?

[SWITCH TO THE PYTHON NOTEBOOK]

Quick look at a 20th century programming language

Generators:

- they can be defined like functions
- use `yield` instead of `return` to provide a sequence of values

Quick look at a 20th century programming language

Generators:

- they can be defined like functions
- use `yield` instead of `return` to provide a sequence of values

In [1]:
```
def fib(up_to):
    current, next = 0, 1
    while current <= up_to:
        yield current
        current, next = next, current + next
```

Quick look at a 20th century programming language

Generators:

- they can be defined like functions
- use `yield` instead of `return` to provide a sequence of values

In [1]:
```
def fib(up_to):
    current, next = 0, 1
    while current <= up_to:
        yield current
        current, next = next, current + next
```

In [2]:
```
# output Fibonacci numbers up to 1000
for x in fib(1000):
    print(x,end=" ")
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Example: `map` in Python

In [3]:
```
# * using `map` in Python
# * output squares of the same Fibonacci numbers
for x in map(lambda x: x*x, fib(1000)):
    print(x,end=" ")
```

```
0 1 1 4 9 25 64 169 441 1156 3025 7921 20736 54289 142129 372100 974169
```

Example: `map` in Python

In [3]:
```
# * using `map` in Python
# * output squares of the same Fibonacci numbers
for x in map(lambda x: x*x, fib(1000)):
    print(x,end=" ")
```

```
0 1 1 4 9 25 64 169 441 1156 3025 7921 20736 54289 142129 372100 974169
```

Compute the maximum of the squares of the same Fibonacci numbers modulo 789:

In [4]:
```
max(map(lambda x: (x*x) % 789, fib(1000)))
```

Out[4]:
```
658
```

List comprehensions and generator expressions

- more Pythonic
- often a great replacement for functional primitives

List comprehensions and generator expressions

- more Pythonic
- often a great replacement for functional primitives

```
In [5]: max((x*x) % 789 for x in fib(1000))

Out[5]: 658
```

List comprehensions and generator expressions

- more Pythonic
- often a great replacement for functional primitives

```
In [5]: max((x*x) % 789 for x in fib(1000))

Out[5]: 658
```

- Good overview of some of these topics: https://realpython.com/python-map-function/

- An attempt at eliminating functional features from Python 3.0 was not successful :-) https://www.artima.com/weblogs/viewpost.jsp?thread=98196