DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 18

1. STRUCTS
2. MEMORY MANAGEMENT: STACK AND HEAP

STRUCTS

Last time: tuples, e.g., `(12, 1.7, true)`

Structs compared to tuples:

- **Similar:** can hold a few items of different types
- **Different:** the items have names

STRUCTS

Last time: tuples, e.g., `(12, 1.7, true)`

Structs compared to tuples:

- **Similar:** can hold a few items of different types
- **Different:** the items have names

In [2]:

```
// Definition: list items (called fields)
//             and their types

struct Person {
    name: String,
    year_born: u16,
    time_100m: f64,
    likes_ice_cream: bool,
}
```

STRUCTS

Last time: tuples, e.g., `(12, 1.7, true)`

Structs compared to tuples:

- **Similar:** can hold a few items of different types
- **Different:** the items have names

In [2]:

```
// Definition: list items (called fields)
//             and their types

struct Person {
    name: String,
    year_born: u16,
    time_100m: f64,
    likes_ice_cream: bool,
}
```

In [3]:

```
// Instantiation: replace types with values

let mut cartoon_character = Person {
    name: String::from("Tasmanian Devil"),
    year_born: 1954,
    time_100m: 7.52,
    likes_ice_cream: true,
};
```

STRUCTS

Last time: tuples, e.g., `(12, 1.7, true)`

Structs compared to tuples:

- **Similar:** can hold a few items of different types
- **Different:** the items have names

In [2]:

```
// Definition: list items (called fields)
//             and their types

struct Person {
    name: String,
    year_born: u16,
    time_100m: f64,
    likes_ice_cream: bool,
}
```

In [3]:

```
// Instantiation: replace types with values

let mut cartoon_character = Person {
    name: String::from("Tasmanian Devil"),
    year_born: 1954,
    time_100m: 7.52,
    likes_ice_cream: true,
};
```

In [4]:

```
// Accessing fields: use ".field_name"
println!("{} was born in {}",
    cartoon_character.name,
    cartoon_character.year_born);
cartoon_character.year_born = 2022;
println!("{} was born in {}",
    cartoon_character.name,
    cartoon_character.year_born);
```

```
Tasmanian Devil was born in 1954
Tasmanian Devil was born in 2022
```

STRUCTS

Last time: tuples, e.g., `(12, 1.7, true)`

Structs compared to tuples:

- **Similar:** can hold a few items of different types
- **Different:** the items have names

In [2]:
```
// Definition: list items (called fields)
//             and their types

struct Person {
    name: String,
    year_born: u16,
    time_100m: f64,
    likes_ice_cream: bool,
}
```

In [3]:
```
// Instantiation: replace types with values

let mut cartoon_character = Person {
    name: String::from("Tasmanian Devil"),
    year_born: 1954,
    time_100m: 7.52,
    likes_ice_cream: true,
};
```

In [4]:
```
// Accessing fields: use ".field_name"
println!("{} was born in {}",
    cartoon_character.name,
    cartoon_character.year_born);
cartoon_character.year_born = 2022;
println!("{} was born in {}",
    cartoon_character.name,
    cartoon_character.year_born);
```

```
Tasmanian Devil was born in 1954
Tasmanian Devil was born in 2022
```

Structs vs tuples:
Which are better?

TUPLE STRUCTS

Named tuples to impose more meaning and delineate a different type.

Example: both `(f64,f64,f64)`

- box size (e.g., 8.5 in × 11 in × 6 in)
- Euclidean coordinates of a point in 3D

TUPLE STRUCTS

Named tuples to impose more meaning and delineate a different type.

Example: both `(f64,f64,f64)`

- box size (e.g., 8.5 in $\times$ 11 in $\times$ 6 in)
- Euclidean coordinates of a point in 3D

In [5]:
```
struct BoxSize(f64,f64,f64);
struct Point(f64,f64,f64);
```

TUPLE STRUCTS

Named tuples to impose more meaning and delineate a different type.

Example: both `(f64,f64,f64)`

- box size (e.g., 8.5 in × 11 in × 6 in)
- Euclidean coordinates of a point in 3D

In [5]:
```
struct BoxSize(f64,f64,f64);
struct Point(f64,f64,f64);
```

In [6]:
```
let mut my_box = BoxSize(3.2,6.0,2.0);
let mut p : Point = Point(-1.3,2.1,0.0);
```

TUPLE STRUCTS

Named tuples to impose more meaning and delineate a different type.

Example: both `(f64,f64,f64)`

- box size (e.g., 8.5 in × 11 in × 6 in)
- Euclidean coordinates of a point in 3D

In [5]:
```
struct BoxSize(f64,f64,f64);
struct Point(f64,f64,f64);
```

In [6]:
```
let mut my_box = BoxSize(3.2,6.0,2.0);
let mut p : Point = Point(-1.3,2.1,0.0);
```

In [7]:
```
// won't work
my_box = p;

// Impossible to accidentally confuse different
// types of triples.
// No runtime penalty! Verified at compilation.
```

```
my_box = p;
         ^ expected struct `BoxSize`, found struct `Poi
nt`
mismatched types
```

TUPLE STRUCTS

Named tuples to impose more meaning and delineate a different type.

Example: both `(f64,f64,f64)`

- box size (e.g., 8.5 in × 11 in × 6 in)
- Euclidean coordinates of a point in 3D

In [5]:
```
struct BoxSize(f64,f64,f64);
struct Point(f64,f64,f64);
```

In [6]:
```
let mut my_box = BoxSize(3.2,6.0,2.0);
let mut p : Point = Point(-1.3,2.1,0.0);
```

In [7]:
```
// won't work
my_box = p;

// Impossible to accidentally confuse different
// types of triples.
// No runtime penalty! Verified at compilation.
```

```
my_box = p;
         ^ expected struct `BoxSize`, found struct `Poi
nt`
mismatched types
```

In [8]:
```
// Acessing via index
println!("{} {} {}",p.0,p.1,p.2);
p.0 = 17.2;

// Destructuring
let Point(first,second,third) = p;
println!("{} {} {}", first, second, third);
```

```
-1.3 2.1 0
17.2 2.1 0
```

NAMED STRUCTS IN ENUMS

Structs with braces and exchangable with tuples in many places

In [9]:
```
enum LPSolution {
    None,
    Point{x:f64,y:f64}
}

let example = LPSolution::Point{x:1.2, y:4.2};
```

NAMED STRUCTS IN ENUMS

Structs with braces and exchangable with tuples in many places

In [9]:
```
enum LPSolution {
    None,
    Point{x:f64,y:f64}
}

let example = LPSolution::Point{x:1.2, y:4.2};
```

In [10]:
```
if let LPSolution::Point{x:first,y:second} = example {
    println!("coordinates: {} {}", first, second);
};
```

```
coordinates: 1.2 4.2
```

MEMORY MANAGEMENT: STACK VS. HEAP

- Two different places where space for data can be allocated
- We will discuss them one by one

STACK

- FILO (first in last out) memory allocation
- Stores current local variables and additional information such as:
  - function arguments
  - function output
  - where to continue when a function terminates
- Fast memory allocation
- Usually small fraction of the memory
- Often: size of the allocated memory has to be known in advance (compilation time)

STACK

- FILO (first in last out) memory allocation
- Stores current local variables and additional information such as:
  - function arguments
  - function output
  - where to continue when a function terminates
- Fast memory allocation
- Usually small fraction of the memory
- Often: size of the allocated memory has to be known in advance (compilation time)

Almost everything you saw so far allocated on stack

- Exception: data in `String` allocated on heap

STACK EXAMPLE (IDEALIZED)

In [11]:

```
fn main() {
    let mut x = 3;
    let mut y = 8;
    println!("x = {}, y = {}",x,y);
    x = add_or_subtract(x,y,true); // x = x + y
    y = add_or_subtract(x,y,false); // y = x - y
    x = add_or_subtract(x,y,false); // x = x - y
    println!("x = {}, y = {}",x,y);
}

fn add_or_subtract(x:i32, y:i32, add:bool) -> i32 {
    let second_arg = if add {y} else {negate(y)};
    x + second_arg
}

fn negate(x:i32) -> i32 {
    -x
}

main();
```

```
x = 3, y = 8
x = 8, y = 3
```

STACK EXAMPLE (IDEALIZED)

```
In [11]: fn main() {
             let mut x = 3;
             let mut y = 8;
             println!("x = {}, y = {}",x,y);
             x = add_or_subtract(x,y,true); // x = x + y
             y = add_or_subtract(x,y,false); // y = x - y
             x = add_or_subtract(x,y,false); // x = x - y
             println!("x = {}, y = {}",x,y);
         }

         fn add_or_subtract(x:i32, y:i32, add:bool) -> i32 {
             let second_arg = if add {y} else {negate(y)};
             x + second_arg
         }

         fn negate(x:i32) -> i32 {
             -x
         }

         main();
```

```
x = 3, y = 8
x = 8, y = 3
```

STEP 1: CALL `main`

- `x` and `y` allocated on stack and initiated
- Stack: `main` (`x`, `y`)

STACK EXAMPLE (IDEALIZED)

```
In [11]: fn main() {
    let mut x = 3;
    let mut y = 8;
    println!("x = {}, y = {}",x,y);
    x = add_or_subtract(x,y,true); // x = x + y
    y = add_or_subtract(x,y,false); // y = x - y
    x = add_or_subtract(x,y,false); // x = x - y
    println!("x = {}, y = {}",x,y);
}

fn add_or_subtract(x:i32, y:i32, add:bool) -> i32 {
    let second_arg = if add {y} else {negate(y)};
    x + second_arg
}

fn negate(x:i32) -> i32 {
    -x
}

main();
```

```
x = 3, y = 8
x = 8, y = 3
```

STEP 1: CALL `main`

- `x` and `y` allocated on stack and initiated
- Stack: `main` (`x`, `y`)

STEP 2: CALL `add_or_subtract` (1ST TIME)

- arguments for `add_or_subtract` put on stack
- space for solution allocated on stack
- space for `second_arg` allocated as well
- Stack: `main` (`x`, `y`), `add_or_subtract` (all the above + auxiliary information)

STACK EXAMPLE (IDEALIZED)

In [ ]:
```
fn main() {
    let mut x = 3;
    let mut y = 8;
    println!("x = {}, y = {}",x,y);
    x = add_or_subtract(x,y,true);
    y = add_or_subtract(x,y,false);
    x = add_or_subtract(x,y,false);
    println!("x = {}, y = {}",x,y);
}

fn add_or_subtract(x:i32, y:i32, add:bool) -> i32 {
    let second_arg = if add {y} else {negate(y)};
    x + second_arg
}

fn negate(x:i32) -> i32 {
    -x
}

main();
```

STEP 3: `add_or_subtract` TERMINATES

- process and remove all information about the call
- Stack: `main` (`x`, `y`)

STACK EXAMPLE (IDEALIZED)

In [ ]:
```
fn main() {
    let mut x = 3;
    let mut y = 8;
    println!("x = {}, y = {}",x,y);
    x = add_or_subtract(x,y,true);
    y = add_or_subtract(x,y,false);
    x = add_or_subtract(x,y,false);
    println!("x = {}, y = {}",x,y);
}

fn add_or_subtract(x:i32, y:i32, add:bool) -> i32 {
    let second_arg = if add {y} else {negate(y)};
    x + second_arg
}

fn negate(x:i32) -> i32 {
    -x
}

main();
```

STEP 3: `add_or_subtract` TERMINATES

- process and remove all information about the call
- Stack: `main` (`x`, `y`)

STEP 4: CALL `add_or_subtract` (2ND TIME)

- arguments for `add_or_subtract` put on stack
- space for solution allocated on stack
- space for `second_arg` allocated as well
- Stack: `main` (`x`, `y`), `add_or_subtract` (all the above + auxiliary information)

STACK EXAMPLE (IDEALIZED)

In [ ]:
```
fn main() {
    let mut x = 3;
    let mut y = 8;
    println!("x = {}, y = {}",x,y);
    x = add_or_subtract(x,y,true);
    y = add_or_subtract(x,y,false);
    x = add_or_subtract(x,y,false);
    println!("x = {}, y = {}",x,y);
}

fn add_or_subtract(x:i32, y:i32, add:bool) -> i32 {
    let second_arg = if add {y} else {negate(y)};
    x + second_arg
}

fn negate(x:i32) -> i32 {
    -x
}

main();
```

STEP 5: CALL `negate` (1ST TIME)

- the argument for `negate` put on stack
- space for solution allocated on stack
- Stack: `main` (`x`, `y`), `add_or_subtract` (...), `negate` (all of the above + auxiliary information)

STACK EXAMPLE (IDEALIZED)

In [ ]:
```
fn main() {
    let mut x = 3;
    let mut y = 8;
    println!("x = {}, y = {}",x,y);
    x = add_or_subtract(x,y,true);
    y = add_or_subtract(x,y,false);
    x = add_or_subtract(x,y,false);
    println!("x = {}, y = {}",x,y);
}

fn add_or_subtract(x:i32, y:i32, add:bool) -> i32 {
    let second_arg = if add {y} else {negate(y)};
    x + second_arg
}

fn negate(x:i32) -> i32 {
    -x
}

main();
```

STEP 5: CALL `negate` (1ST TIME)

- the argument for `negate` put on stack
- space for solution allocated on stack
- Stack: `main` (`x`, `y`), `add_or_subtract` (…), `negate` (all of the above + auxiliary information)

STEP 6: `negate` TERMINATES

- process and remove all information about the call
- Stack: `main` (`x`, `y`), `add_or_subtract` (…)

STACK EXAMPLE (IDEALIZED)

In [ ]:
```
fn main() {
    let mut x = 3;
    let mut y = 8;
    println!("x = {}, y = {}",x,y);
    x = add_or_subtract(x,y,true);
    y = add_or_subtract(x,y,false);
    x = add_or_subtract(x,y,false);
    println!("x = {}, y = {}",x,y);
}

fn add_or_subtract(x:i32, y:i32, add:bool) -> i32 {
    let second_arg = if add {y} else {negate(y)};
    x + second_arg
}

fn negate(x:i32) -> i32 {
    -x
}

main();
```

STEP 7: `add_or_subtract` TERMINATES

- [...]
- Stack: `main` (`x`, `y`)

STACK EXAMPLE (IDEALIZED)

In [ ]:
```
fn main() {
    let mut x = 3;
    let mut y = 8;
    println!("x = {}, y = {}",x,y);
    x = add_or_subtract(x,y,true);
    y = add_or_subtract(x,y,false);
    x = add_or_subtract(x,y,false);
    println!("x = {}, y = {}",x,y);
}

fn add_or_subtract(x:i32, y:i32, add:bool) -> i32 {
    let second_arg = if add {y} else {negate(y)};
    x + second_arg
}

fn negate(x:i32) -> i32 {
    -x
}

main();
```

STEP 7: `add_or_subtract` TERMINATES

- [...]
- Stack: `main` (`x`, `y`)

STEP 8: CALL `add_or_subtract` (3RD TIME)

- [...]
- Stack: `main` (`x`, `y`), `add_or_subtract` (...)

STACK EXAMPLE (IDEALIZED)

In [ ]:
```
fn main() {
    let mut x = 3;
    let mut y = 8;
    println!("x = {}, y = {}",x,y);
    x = add_or_subtract(x,y,true);
    y = add_or_subtract(x,y,false);
    x = add_or_subtract(x,y,false);
    println!("x = {}, y = {}",x,y);
}

fn add_or_subtract(x:i32, y:i32, add:bool) -> i32 {
    let second_arg = if add {y} else {negate(y)};
    x + second_arg
}

fn negate(x:i32) -> i32 {
    -x
}

main();
```

STEP 7: `add_or_subtract` TERMINATES

- [...]
- Stack: `main` (`x`, `y`)

STEP 8: CALL `add_or_subtract` (3RD TIME)

- [...]
- Stack: `main` (`x`, `y`), `add_or_subtract` (...)

...

LIMITED SPACE ON STACK!

In [12]:

```
fn same_number(x:u32) -> u32 {
    match x {
        0 => 0,
        _ => 1 + same_number(x - 1),
    }
}
```

LIMITED SPACE ON STACK!

In [12]:

```
fn same_number(x:u32) -> u32 {
    match x {
        0 => 0,
        _ => 1 + same_number(x - 1),
    }
}
```

In [13]:

```
same_number(7)
```

Out[13]: 7

LIMITED SPACE ON STACK!

In [12]:

```
fn same_number(x:u32) -> u32 {
    match x {
        0 => 0,
        _ => 1 + same_number(x - 1),
    }
}
```

In [13]:

```
same_number(7)
```

Out[13]: 7

In [14]:

```
same_number(123_456)
```

Out[14]: 123456

LIMITED SPACE ON STACK!

In [12]:

```
fn same_number(x:u32) -> u32 {
    match x {
        0 => 0,
        _ => 1 + same_number(x - 1),
    }
}
```

In [13]:

```
same_number(7)
```

Out[13]: 7

In [14]:

```
same_number(123_456)
```

Out[14]: 123456

In [15]:

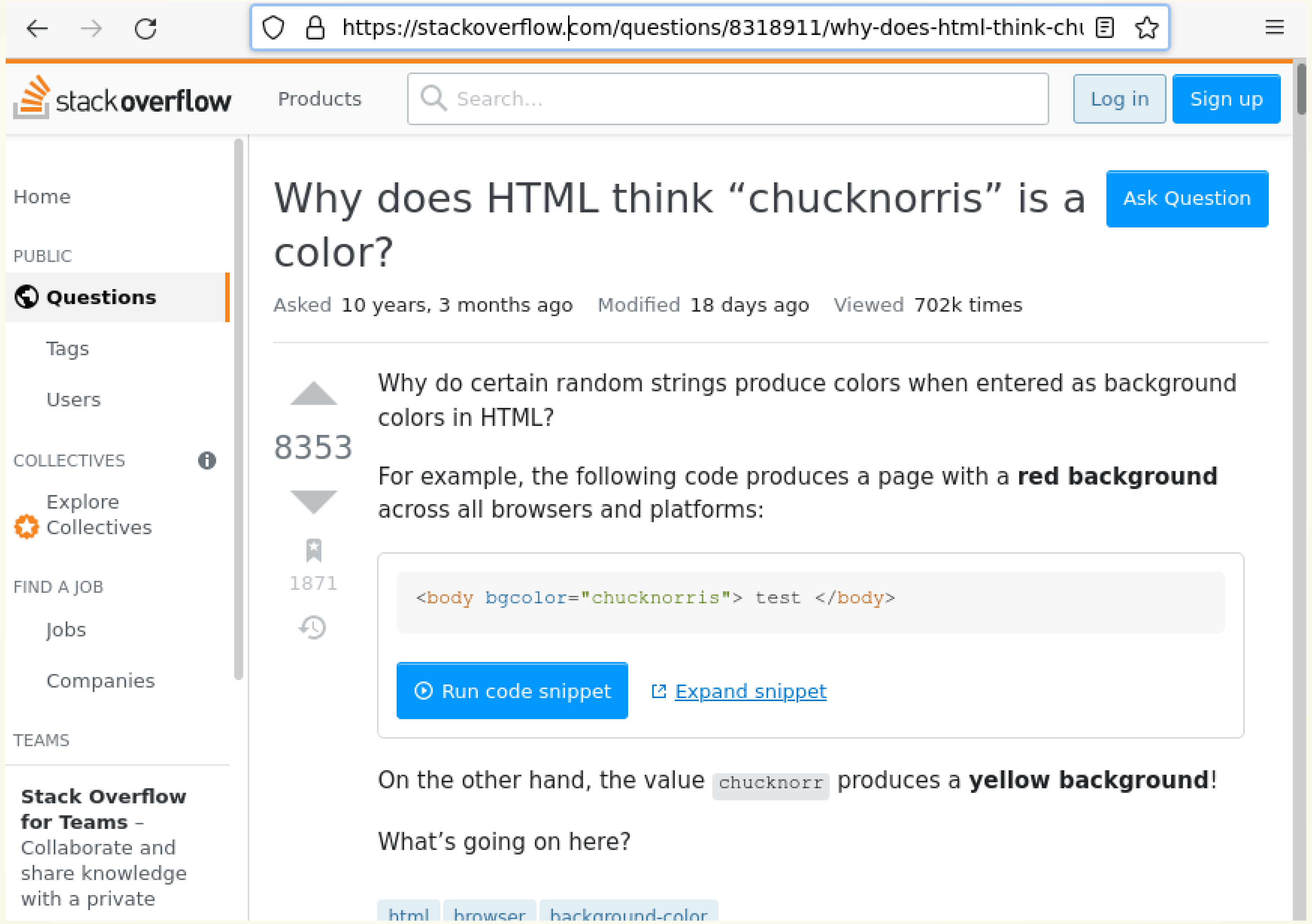
```
same_number(1_000_000)
```

```
Child process terminated with status: signal: 11 (core dumped)
```

USING TOO MUCH MEMORY ON STACK: *STACK OVERFLOW*

USING TOO MUCH MEMORY ON STACK: *STACK OVERFLOW*

This is where the name of the popular webpage for asking questions about programming comes from!



HEAP

- Memory allocated and freed in arbitrary order
- Arbitrary amount allocated
- The application knows a *pointer* = the address of assigned memory

Pros and cons?

HEAP

- Memory allocated and freed in arbitrary order
- Arbitrary amount allocated
- The application knows a *pointer* = the address of assigned memory

Pros and cons?

Pros:

- Arbitrary amount of data
- No copying to pass data around
  - Just share the pointer!

Cons:

- Slower allocation:
  - Possible request for more space to the operating system
- Possible memory fragmentation
- Slower access:
  - Have to follow the pointer to get to data

STACK VS. HEAP IN PYTHON

- Elementary pieces of data allocated on stack: integers, floats, Boolean values, ...
- Anything else allocated on the heap

STACK VS. HEAP IN PYTHON

- Elementary pieces of data allocated on stack: integers, floats, Boolean values, ...
- Anything else allocated on the heap

[SWITCH TO THE PYTHON NOTEBOOK]

Sample difference between stack and heap in Python

In [1]:
```python
# x on the stack, copied when passed to the function
# Modifying the copy doesn't modify the original.
def plus_one(x):
    x += 1

x = 3
print(x)
plus_one(x)
print(x)
```

```
3
3
```

Sample difference between stack and heap in Python

In [1]:

```python
# x on the stack, copied when passed to the function
# Modifying the copy doesn't modify the original.
def plus_one(x):
    x += 1

x = 3
print(x)
plus_one(x)
print(x)
```

```
3
3
```

In [2]:

```python
# Internally, a list is allocated on the heap.
# Passing a list to a function means copying
# its pointer, not a copy of the list. Modifying
# the list will modify the original.

def append_one(y):
    y.append(1)

y = [4,3,2]
print(y)
append_one(y)
print(y)
```

```
[4, 3, 2]
[4, 3, 2, 1]
```

Stack overflow in Python?

In [3]:
```
def same_number(x):
    if x == 0:
        return 0
    else:
        return 1 + same_number(x-1)

same_number(123)
```

Out[3]: 123

Stack overflow in Python?

In [3]:
```
def same_number(x):
    if x == 0:
        return 0
    else:
        return 1 + same_number(x-1)

same_number(123)
```

Out[3]: 123

In [4]:
```
# overflow the stack
same_number(1230000)
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Input In [4], in <module>
      1 # overflow the stack
----> 2 same_number(1230000)

Input In [3], in same_number(x)
      3     return 0
      4 else:
----> 5     return 1 + same_number(x-1)

Input In [3], in same_number(x)
      3     return 0
      4 else:
----> 5     return 1 + same_number(x-1)

    [... skipping similar frames: same_number at line 5 (2969 times)]

Input In [3], in same_number(x)
      3     return 0
      4 else:
----> 5     return 1 + same_number(x-1)

Input In [3], in same_number(x)
      1 def same_number(x):
----> 2     if x == 0:
      3         return 0
      4     else:

RecursionError: maximum recursion depth exceeded in comparison
```

BONUS CONTENT: STACK OVERFLOW?

In [16]:

```
// an obfuscated way of computing 1 so the compiler
// does not realize :-)
fn return_one(x:u64) -> u64 {
    let x = (if x > 1000 {x-10} else {x}) as u128;
    let y = (x + 1) * (x + 1);
    (y - 2*x - x*x) as u64
}
```

BONUS CONTENT: STACK OVERFLOW?

In [16]:

```
// an obfuscated way of computing 1 so the compiler
// does not realize :-)
fn return_one(x:u64) -> u64 {
    let x = (if x > 1000 {x-10} else {x}) as u128;
    let y = (x + 1) * (x + 1);
    (y - 2*x - x*x) as u64
}
```

In [17]:

```
fn same_number_2(x:u64) -> u64 {
    fn same_number_aux(y:u64, accumulate:u64) -> u64 {
        match y {
            0 => accumulate,
            _ => same_number_aux(
                y - return_one(y),
                accumulate + 1),
        }
    }
    same_number_aux(x,0)
}
```

BONUS CONTENT: STACK OVERFLOW?

In [16]:

```
// an obfuscated way of computing 1 so the compiler
// does not realize :-)
fn return_one(x:u64) -> u64 {
    let x = (if x > 1000 {x-10} else {x}) as u128;
    let y = (x + 1) * (x + 1);
    (y - 2*x - x*x) as u64
}
```

In [17]:

```
fn same_number_2(x:u64) -> u64 {
    fn same_number_aux(y:u64, accumulate:u64) -> u64 {
        match y {
            0 => accumulate,
            _ => same_number_aux(
                y - return_one(y),
                accumulate + 1),
        }
    }
    same_number_aux(x,0)
}
```

In [18]:

```
same_number_2(1234)
```

Out[18]:

```
1234
```

?

BONUS CONTENT: STACK OVERFLOW?

In [16]:
```
// an obfuscated way of computing 1 so the compiler
// does not realize :-)
fn return_one(x:u64) -> u64 {
    let x = (if x > 1000 {x-10} else {x}) as u128;
    let y = (x + 1) * (x + 1);
    (y - 2*x - x*x) as u64
}
```

In [17]:
```
fn same_number_2(x:u64) -> u64 {
    fn same_number_aux(y:u64, accumulate:u64) -> u64 {
        match y {
            0 => accumulate,
            _ => same_number_aux(
                y - return_one(y),
                accumulate + 1),
        }
    }
    same_number_aux(x,0)
}
```

In [18]:
```
same_number_2(1234)
```

Out[18]: 1234

In [19]:
```
same_number_2(10_000_000_00)
```

Out[19]: 1000000000

- **No stack overflow!** Why? Look up **tail call** and **tail recursion**.
- Not guaranteed in Rust, but sometimes works.