Structs and Such

Structs and such

By QuentinUK (Own work), via Wikimedia Commons

Shop till you drop!

Declaring a structure

```
struct Store {
    name: String,
    prices: Vec<Item>,
}
```

Declaring a structure

```
struct Store {
    name: String,
    prices: Vec<Item>,
}
```

Declaring a structure

```
struct Store {
    name: String,
    prices: Vec<Item>,
}
```

Declaring a structure

```
struct Store {
    name: String,
    prices: Vec<Item>,
}
```

Declaring a structure

```
struct Store {
    name: String,
    prices: Vec<Item>,
}
```

Declaring a structure

```
struct Store {
    name: String,
    prices: Vec<Item>,
}
```

Declaring a structure

```
struct Store {
    name: String,
    prices: Vec<Item>,
}
```

Declaring a structure

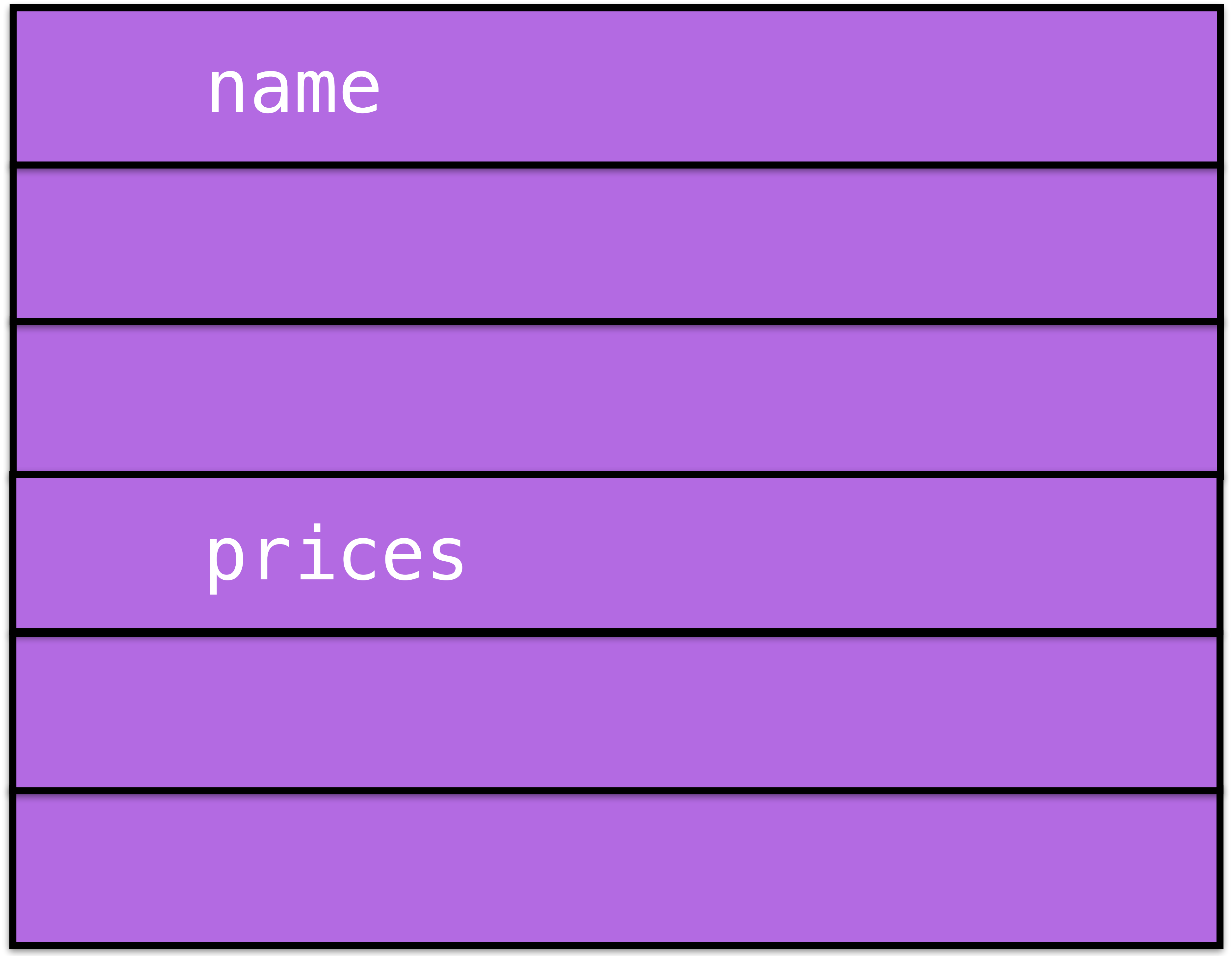
```
struct Store {
    name: String,
    prices: Vec<Item>,
}
```

Declaring a structure

```
struct Store {
    name: String,
    prices: Vec<Item>,
}
```

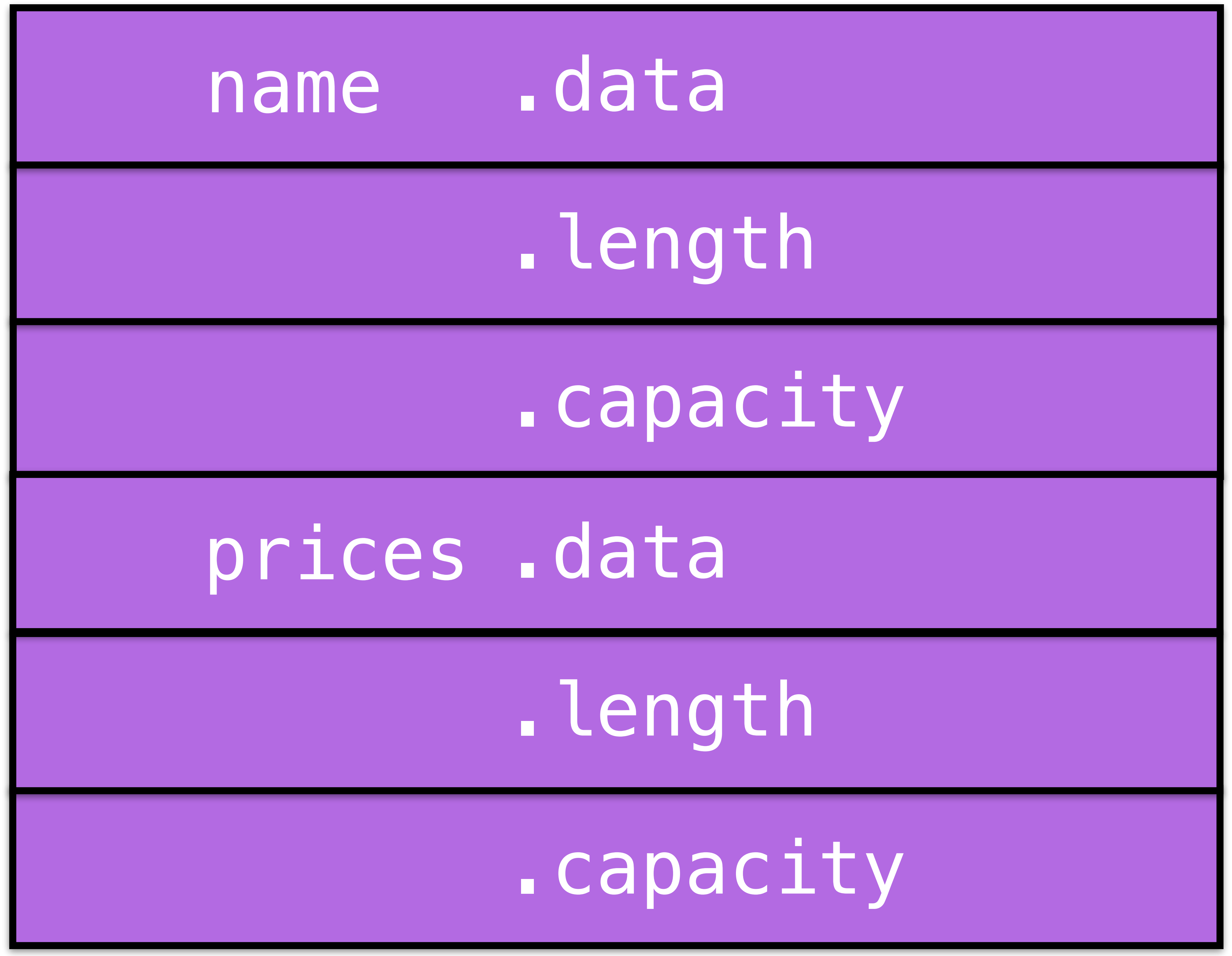
Declaring a structure

```
struct Store {
    name: String,
    prices: Vec<Item>,
}
```



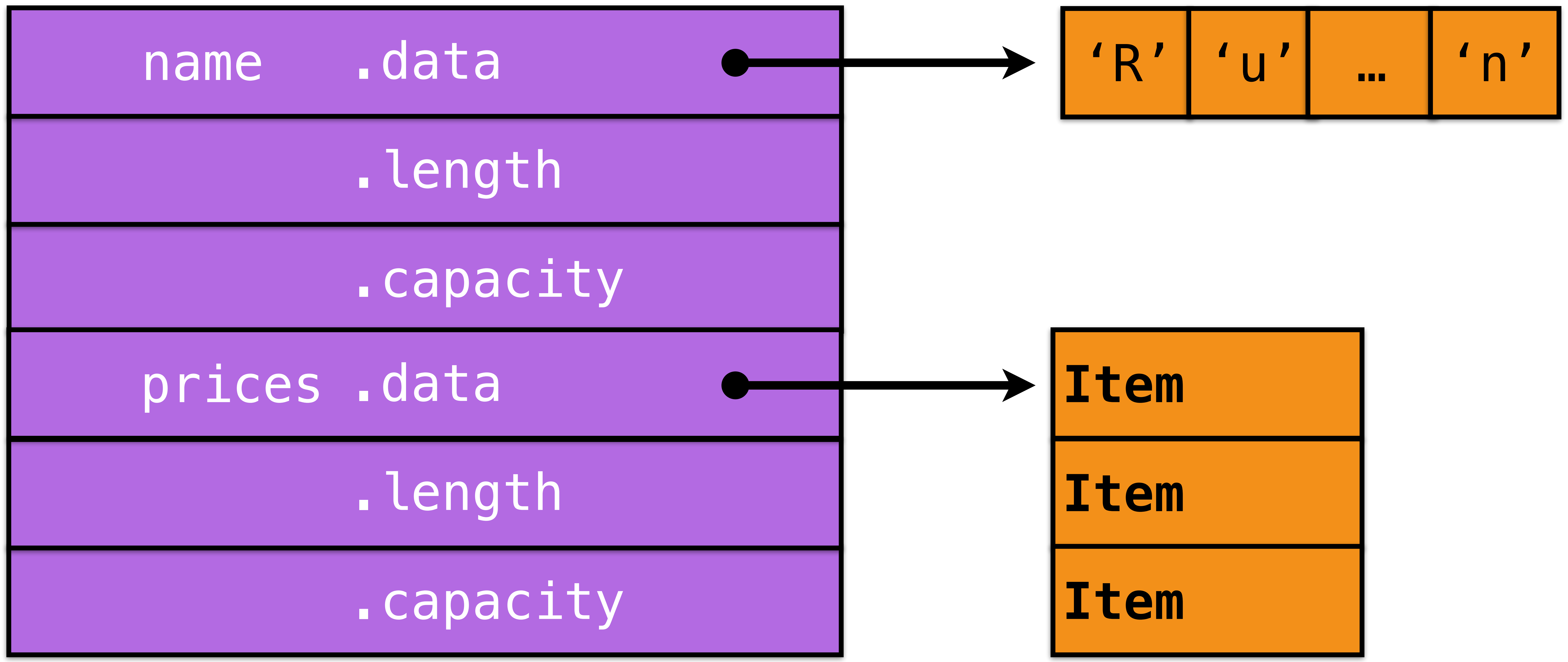
Declaring a structure

```
struct Store {
    name: String,
    prices: Vec<Item>,
}
```



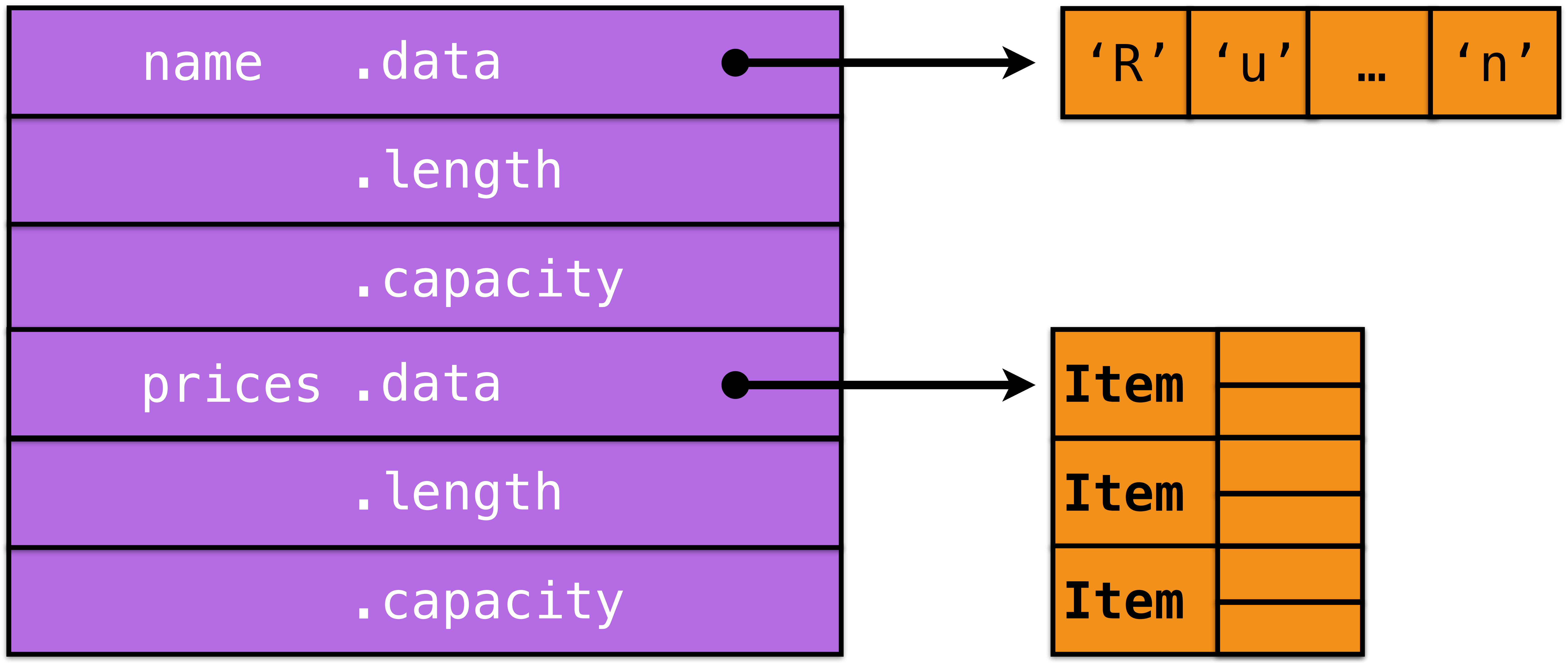
Declaring a structure

```
struct Store {
    name: String,
    prices: Vec<Item>,
}
```



Declaring a structure

```
struct Store {
    name: String,
    prices: Vec<Item>,
}
```



```rust
struct Item {
    name: String,
    price: f32,
}
```

Other fundamental types

`f32`	`i8`	`u8`	`&str`
`f64`	`i16`	`u16`	`&[T]`
	`i32`	`u32`	
	`i64`	`u64`	
	`isize`	`usize`	
floats	signed	unsigned	slices

```
struct Item {
    name: String,
    price: f32,
}
```

Other fundamental types

floats	signed	unsigned	slices
`f32`	`i8`	`u8`	`&str`
`f64`	`i16`	`u16`	`&[T]`
	`i32`	`u32`	
	`i64`	`u64`	
	`isize`	`usize`	

```rust
struct Item {
    name: String,
    price: f32,
}
```

Other fundamental types

`f32`	`i8`	`u8`	`&str`
`f64`	`i16`	`u16`	`&[T]`
	`i32`	`u32`	
	`i64`	`u64`	
	`isize`	`usize`	
floats	signed	unsigned	slices

```rust
struct Item {
    name: String,
    price: f32,
}
```

Other fundamental types

`f32`	`i8`	`u8`	`&str`
`f64`	`i16`	`u16`	`&[T]`
	`i32`	`u32`	
	`i64`	`u64`	
	`isize`	`usize`	
floats	signed	unsigned	slices

```
struct Item {
    name: String,
    price: f32,
}
```

Other fundamental types

`f32`	`i8`	`u8`	`&str`
`f64`	`i16`	`u16`	`&[T]`
	`i32`	`u32`	
	`i64`	`u64`	
	`isize`	`usize`	
floats	signed	unsigned	slices

```
struct Item {
    name: String,
    price: f32,
}
```

Other fundamental types

floats	signed	unsigned	slices
`f32`	`i8`	`u8`	`&str`
`f64`	`i16`	`u16`	`&[T]`
	`i32`	`u32`	
	`i64`	`u64`	
	`isize`	`usize`	

```
struct Item {
    name: String,
    price: f32,
}
```

Other fundamental types

`f32`	`i8`	`u8`	`&str`
`f64`	`i16`	`u16`	`&[T]`
	`i32`	`u32`	
	`i64`	`u64`	
	`isize`	`usize`	
floats	signed	unsigned	slices

```
struct Item {
    name: String,
    price: f32,
}
```

Other fundamental types

f32	i8	u8	&str
f64	i16	u16	&[T]
	i32	u32	
	i64	u64	
	isize	usize	
floats	signed	unsigned	slices

```rust
struct Item {
    name: String,
    price: f32,
}
```

Other fundamental types

`f32`	`i8`	`u8`	`&str`
`f64`	`i16`	`u16`	`&[T]`
	`i32`	`u32`	
	`i64`	`u64`	
	`isize`	`usize`	
floats	signed	unsigned	slices

```
struct Item {
    name: String,
    price: f32,
}
```

Other fundamental types

`f32`	`i8`	`u8`	`&str`
`f64`	`i16`	`u16`	`&[T]`
	`i32`	`u32`	
	`i64`	`u64`	
	`isize`	`usize`	
floats	signed	unsigned	slices

```
struct Item {
    name: String,
    price: f32,
}
```

Other fundamental types

`f32`	`i8`	`u8`	`&str`
`f64`	`i16`	`u16`	`&[T]`
	`i32`	`u32`	
	`i64`	`u64`	
	`isize`	`usize`	
floats	signed	unsigned	slices

Methods

```
struct Store { .. }
struct Item { .. }

impl Store {
  fn add_item(&mut self, item: Item) {
    self.items.push(item);
  }

  fn price(&self, item_name: &str) -> f32 {
    … // see upcoming slide
  }
}
```

Methods

```
struct Store { .. }
struct Item { .. }

impl Store {
  fn add_item(&mut self, item: Item) {
    self.items.push(item);
  }

  fn price(&self, item_name: &str) -> f32 {
    … // see upcoming slide
  }
}
```

Methods

```
struct Store { .. }
struct Item { .. }

impl Store {
  fn add_item(&mut self, item: Item) {
    self.items.push(item);
  }

  fn price(&self, item_name: &str) -> f32 {
    … // see upcoming slide
  }
}
```

Methods

```
struct Store { .. }
struct Item { .. }

impl Store {
  fn add_item(&mut self, item: Item) {
    self.items.push(item);
  }

  fn price(&self, item_name: &str) -> f32 {
    … // see upcoming slide
  }
}
```

```
store.add_item(…); // must be let mut
store.price(…);    // let OR let mut
```

Methods

```
struct Store { .. }
struct Item { .. }

impl Store {
  fn add_item(&mut self, item: Item) {
    self.items.push(item);
  }

  fn price(&self, item_name: &str) -> f32 {
    … // see upcoming slide
  }
}
```

```
store.add_item(…); // must be let mut
store.price(…);    // let OR let mut
```

Methods

```
struct Store { .. }
struct Item { .. }

impl Store {
  fn add_item(&mut self, item: Item) {
    self.items.push(item);
  }

  fn price(&self, item_name: &str) -> f32 {
    … // see upcoming slide
  }
}
```

itself an &mut method

```
store.add_item(…); // must be let mut
store.price(…);    // let OR let mut
```

Methods

```
struct Store { .. }
struct Item { .. }

impl Store {
  fn add_item(&mut self, item: Item) {
    self.items.push(item);
  }

  fn price(&self, item_name: &str) -> f32 {
    … // see upcoming slide
  }
}
```

```
store.add_item(…); // must be let mut
store.price(…);    // let OR let mut
```

Methods

```
struct Store { .. }
struct Item { .. }

impl Store {
  fn add_item(&mut self, item: Item) {
    self.items.push(item);
  }

  fn price(&self, item_name: &str) -> f32 {
    … // see upcoming slide
  }
}
```

```
store.add_item(…); // must be let mut
store.price(…);    // let OR let mut
```

Methods

```
struct Store { .. }
struct Item { .. }

impl Store {
  fn add_item(&mut self, item: Item) {
    self.items.push(item);
  }

  fn price(&self, item_name: &str) -> f32 {
    … // see upcoming slide
  }
}
```

```
store.add_item(…); // must be let mut
store.price(…);    // let OR let mut
```

Methods

```
struct Store { .. }
struct Item { .. }

impl Store {
  fn add_item(&mut self, item: Item) {
    self.items.push(item);
  }

  fn price(&self, item_name: &str) -> f32 {
    … // see upcoming slide
  }
}
```

```
store.add_item(…); // must be let mut
store.price(…);    // let OR let mut
```

Methods

```
struct Store { .. }
struct Item { .. }

impl Store {
  fn add_item(&mut self, item: Item) {
    self.items.push(item);
  }

  fn price(&self, item_name: &str) -> f32 {
    … // see upcoming slide
  }
}
```

```
store.add_item(…); // must be let mut
store.price(…);    // let OR let mut
```

Methods

```
struct Store { .. }
struct Item { .. }

impl Store {
  fn add_item(&mut self, item: Item) {
    self.items.push(item);
  }

  fn price(&self, item_name: &str) -> f32 {
    … // see upcoming slide
  }
}
```

```
store.add_item(…); // must be let mut
store.price(…);    // let OR let mut
```

Methods

```
struct Store { .. }

impl Store {
  fn new(name: String) -> Store {
    return Store {
      name: name,
      items: vec![],
    };
  }
}
```

Methods

```
struct Store { .. }

impl Store {
  fn new(name: String) -> Store {
    return Store {
      name: name,
      items: vec![],
    };
  }
}
```

Methods

```
struct Store { .. }

impl Store {
  fn new(name: String) -> Store {
    return Store {
      name: name,
      items: vec![],
    };
  }
}
```

```
Store::new(some_name)
```

Methods

```
struct Store { .. }

impl Store {
  fn new(name: String) -> Store {
    return Store {
      name: name,
      items: vec![],
    };
  }
}
```

```
Store::new(some_name)
```

Methods

```
struct Store { .. }

impl Store {
  fn new(name: String) -> Store {
    return Store {
      name: name,
      items: vec![],
    };
  }
}
```

```
Store::new(some_name)
```

Methods

```
struct Store { .. }

impl Store {
  fn new(name: String) -> Store {
    return Store {
      name: name,
      items: vec![],
    };
  }
}
```

```
Store::new(some_name)
```

Methods

```
struct Store { .. }

impl Store {
  fn new(name: String) -> Store {
    return Store {
      name: name,
      items: vec![],
    };
  }
}
```

```
Store::new(some_name)
```

Methods

```
struct Store { .. }

impl Store {
  fn new(name: String) -> Store {
    return Store {
      name: name,
      items: vec![],
    };
  }
}
```

```
Store::new(some_name)
```

Methods

```
struct Store { .. }

impl Store {
  fn new(name: String) -> Store {
    return Store {
      name: name,
      items: vec![],
    };
  }
}
```

```
Store::new(some_name)
```

Methods

```
struct Store { .. }

impl Store {
  fn new(name: String) -> Store {
    return Store {
      name: name,
      items: vec![],
    };
  }
}
```

```
Store::new(some_name)
```

Methods

```
struct Store { .. }

impl Store {
  fn new(name: String) -> Store {
    return Store {
      name: name,
      items: vec![],
    };
  }
}
```

```
Store::new(some_name)
```

Methods

```
struct Store { .. }

impl Store {
  fn new(name: String) -> Store {
    return Store {
      name: name,
      items: vec![],
    };
  }
}
```

```
Store::new(some_name)
```

Return is optional

```
struct Store { .. }

impl Store {
  fn new(name: String) -> Store {
    Store {
      name: name,
      items: vec![],
    }
  }
}
```

Return is optional

```
struct Store { .. }

impl Store {
  fn new(name: String) -> Store {
    Store {
      name: name,
      items: vec![],
    }
  }
}
```

No `;` on last expression:
“return this value”

Options and Enums

```
enum Option<T> {
  Some(T),
  None
}
fn main() {
  use Option::*;
  let v: Option<i32> = Some(22);
  match v {
    Some(x) => println!(“v = {}”, x),
    None => println!(“v = None”),
  }
  println!(“v = {}”, v.unwrap()); // risky
}
```

Options and Enums

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

```
fn main() {
  use Option::*;
  let v: Option<i32> = Some(22);
  match v {
    Some(x) => println!(“v = {}”, x),
    None => println!(“v = None”),
  }
  println!(“v = {}”, v.unwrap()); // risky
}
```

Options and Enums

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

fn main() {
  use Option::*;
  let v: Option<i32> = Some(22);
  match v {
    Some(x) => println!(“v = {}”, x),
    None => println!(“v = None”),
  }
  println!(“v = {}”, v.unwrap()); // risky
}
```

Options and Enums

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

fn main() {
  use Option::*;
  let v: Option<i32> = Some(22);
  match v {
    Some(x) => println!(“v = {}”, x),
    None => println!(“v = None”),
  }
  println!(“v = {}”, v.unwrap()); // risky
}
```

Options and Enums

```
enum Option<T> {
  Some(T),
  None
}
fn main() {
  use Option::*;
  let v: Option<i32> = Some(22);
  match v {
    Some(x) => println!(“v = {}”, x),
    None => println!(“v = None”),
  }
  println!(“v = {}”, v.unwrap()); // risky
}
```

Options and Enums

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

fn main() {
  use Option::*;
  let v: Option<i32> = Some(22);
  match v {
    Some(x) => println!("v = {}", x),
    None => println!("v = None"),
  }
  println!("v = {}", v.unwrap()); // risky
}
```

Options and Enums

```
enum Option<T> {
  Some(T),
  None
}
fn main() {
  use Option::*;
  let v: Option<i32> = Some(22);
  match v {
    Some(x) => println!(“v = {}”, x),
    None => println!(“v = None”),
  }
  println!(“v = {}”, v.unwrap()); // risky
}
```

Options and Enums

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

fn main() {
  use Option::*;
  let v: Option<i32> = Some(22);
  match v {
    Some(x) => println!("v = {}", x),
    None => println!("v = None"),
  }
  println!("v = {}", v.unwrap()); // risky
}
```

Options and Enums

```
enum Option<T> {
  Some(T),
  None
}
fn main() {
  use Option::*;
  let v: Option<i32> = Some(22);
  match v {
    Some(x) => println!(“v = {}”, x),
    None => println!(“v = None”),
  }
  println!(“v = {}”, v.unwrap()); // risky
}
```

Options and Enums

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

fn main() {
  use Option::*;
  let v: Option<i32> = Some(22);
  match v {
    Some(x) => println!(“v = {}”, x),
    None => println!(“v = None”),
  }
  println!(“v = {}”, v.unwrap()); // risky
}
```

Options and Enums

```
enum Option<T> {
  Some(T),
  None
}
fn main() {
  use Option::*;
  let v: Option<i32> = Some(22);
  match v {
    Some(x) => println!(“v = {}”, x),
    None => println!(“v = None”),
  }
  println!(“v = {}”, v.unwrap()); // risky
}
```

Options and Enums

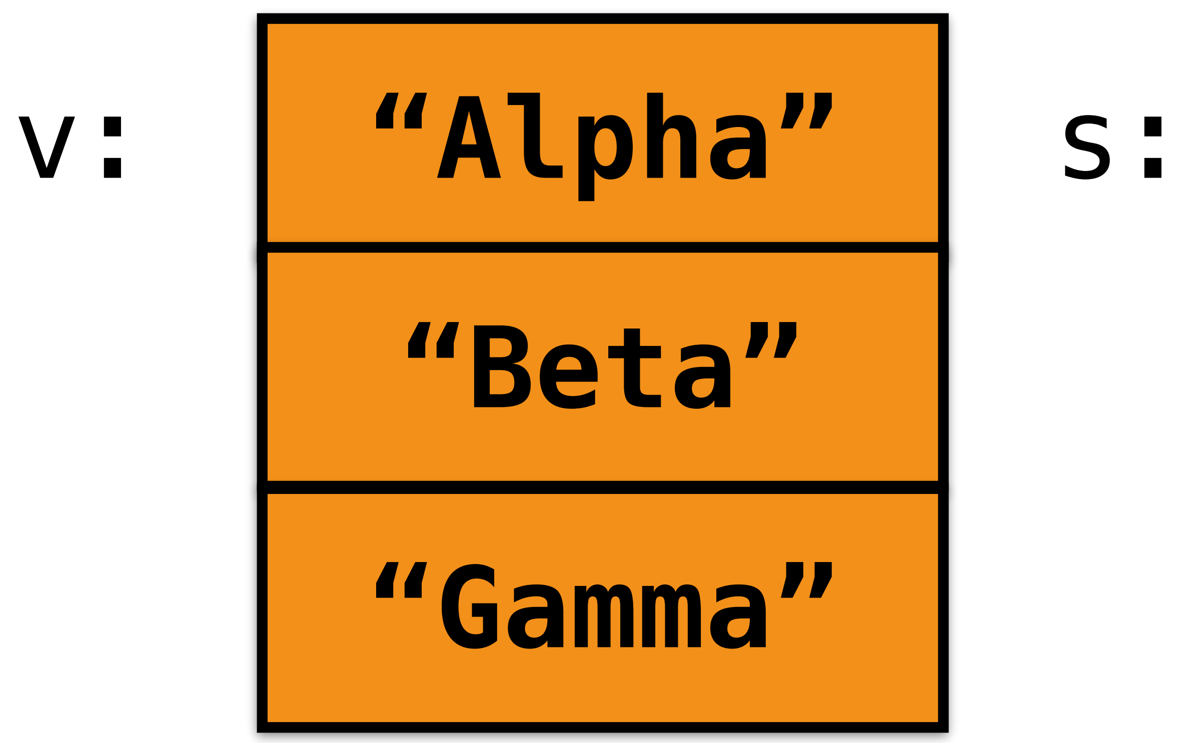
```
enum Option<T> {
  Some(T),
  None
}
fn main() {
  use Option::*;
  let v: Option<i32> = Some(22);
  match v {
    Some(x) => println!(“v = {}”, x),
    None => println!(“v = None”),
  }
  println!(“v = {}”, v.unwrap()); // risky
}
```

Options and Enums

```
enum Option<T> {
  Some(T),
  None
}
fn main() {
  use Option::*;
  let v: Option<i32> = Some(22);
  match v {
    Some(x) => println!(“v = {}”, x),
    None => println!(“v = None”),
  }
  println!(“v = {}”, v.unwrap()); // risky
}
```

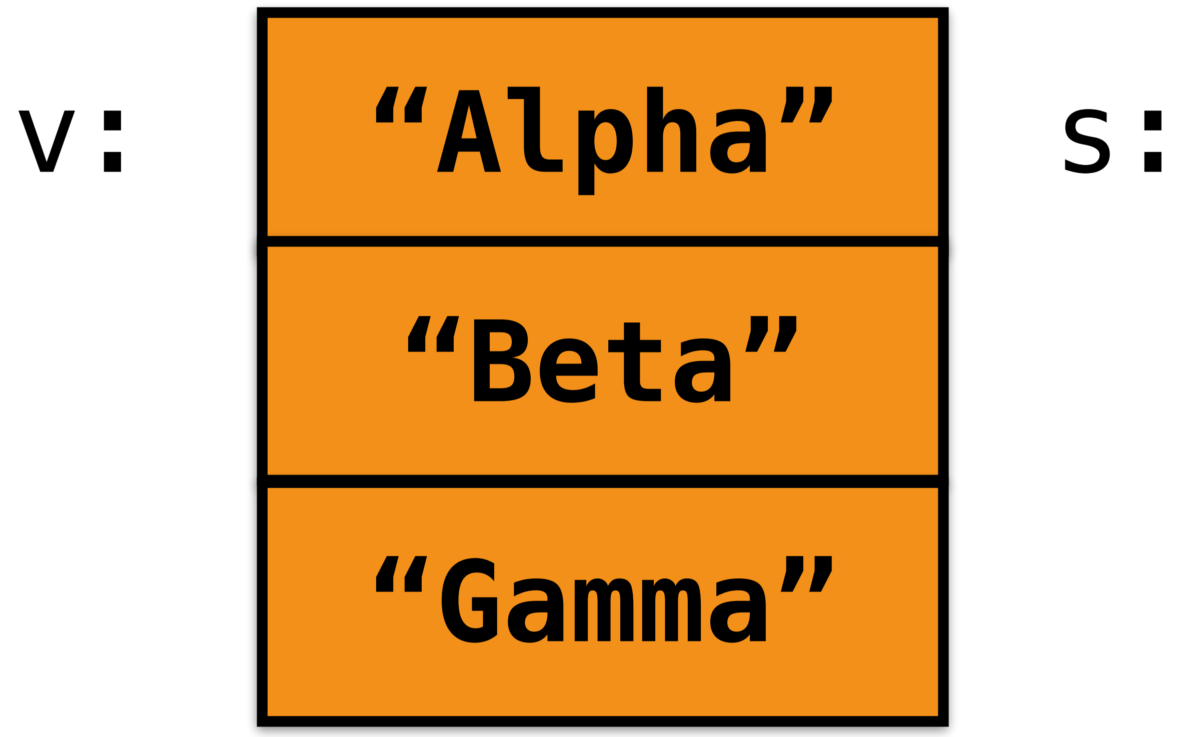
For Loops

```
fn main() {
  let v = vec![format!(“Alpha”),
               format!(“Beta”),
               format!(“Gamma”)];
  for s in v {
    println!(“{:?}”, s);
  }
}
```



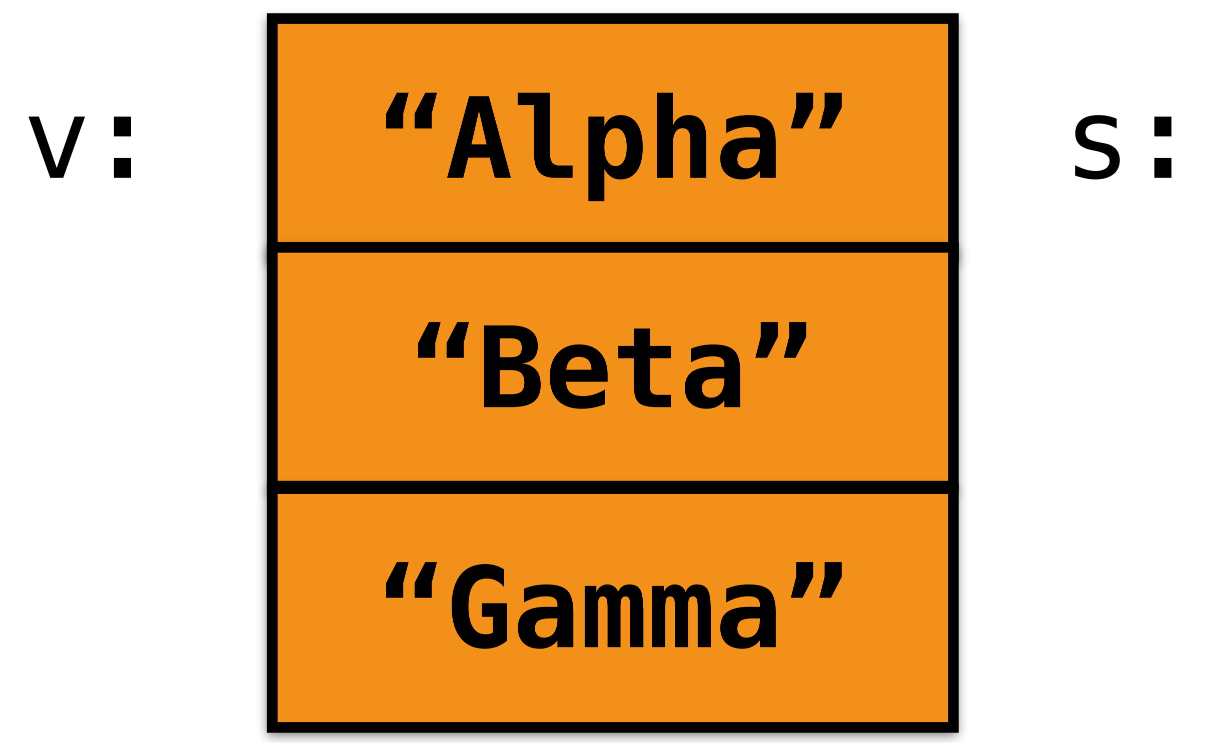
For Loops

```
fn main() {
  let v = vec![format!(“Alpha”),
               format!(“Beta”),
               format!(“Gamma”)];
  for s in v {
    println!(“{:?}”, s);
  }
}
```



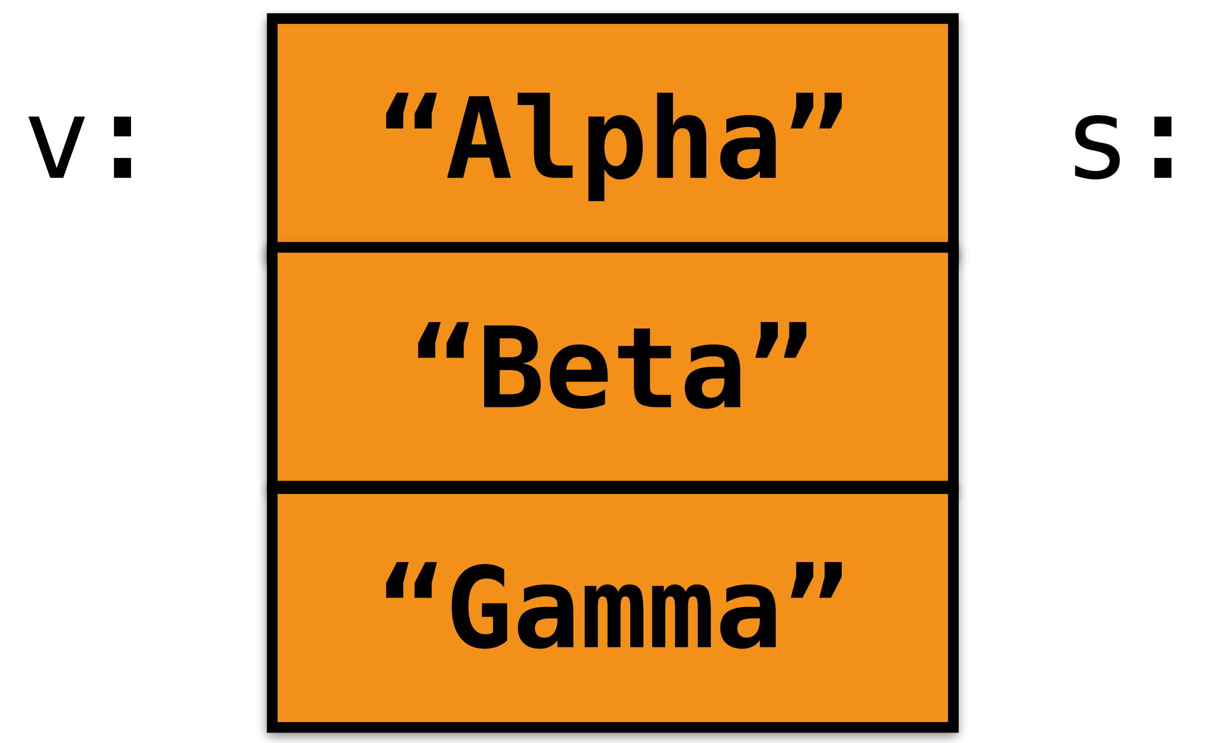
For Loops

```
fn main() {
  let v = vec![format!("Alpha"),
               format!("Beta"),
               format!("Gamma")];
  for s in v {
    println!("{:?}", s);
  }
}
```



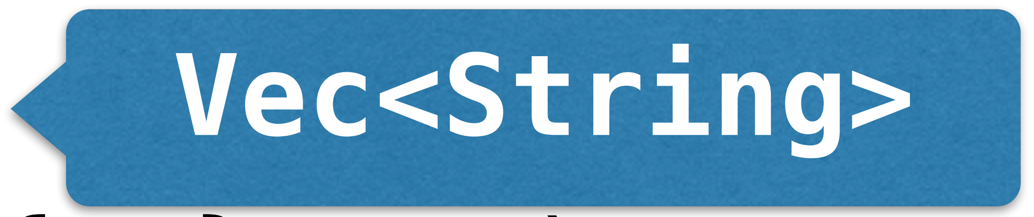
For Loops

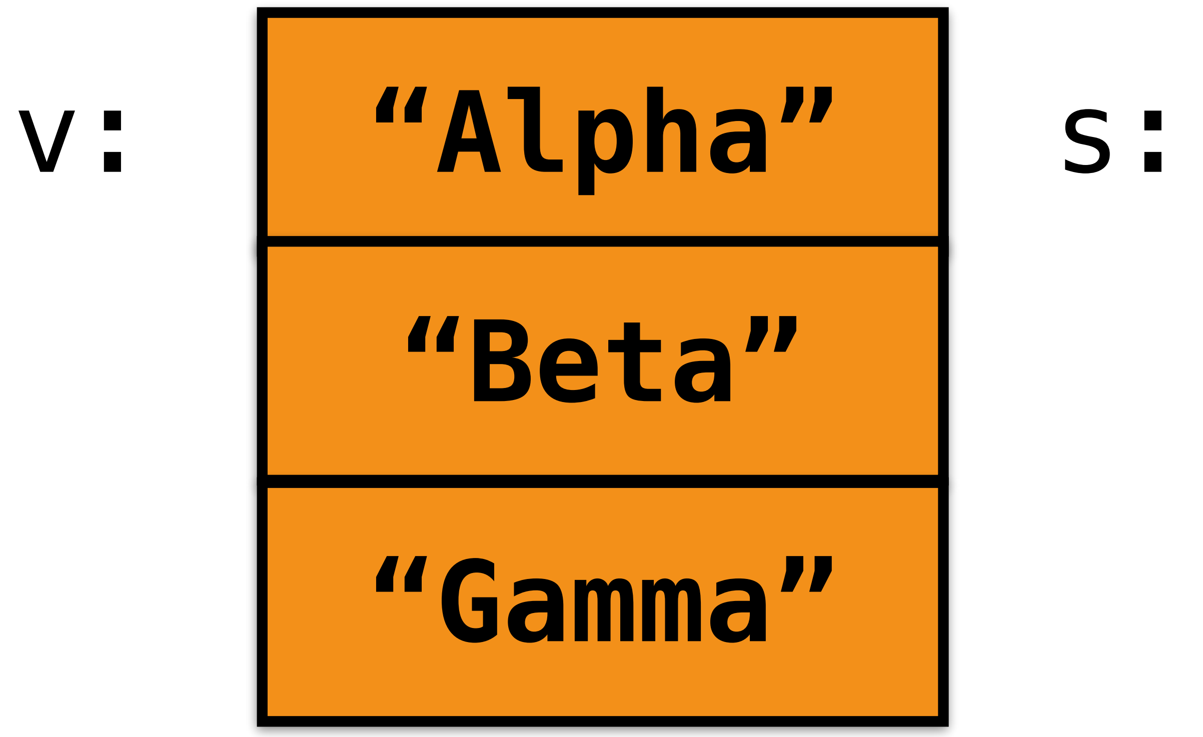
```
fn main() {
  let v = vec![format!(“Alpha”),
               format!(“Beta”),
               format!(“Gamma”)];
  for s in v {
    println!(“{:?}”, s);
  }
}
```



For Loops

```
fn main() {
  let v = vec![format!(“Alpha”),
               format!(“Beta”),
               format!(“Gamma”)];
  for s in v {
    println!(“{:?}”, s);
  }
}
```



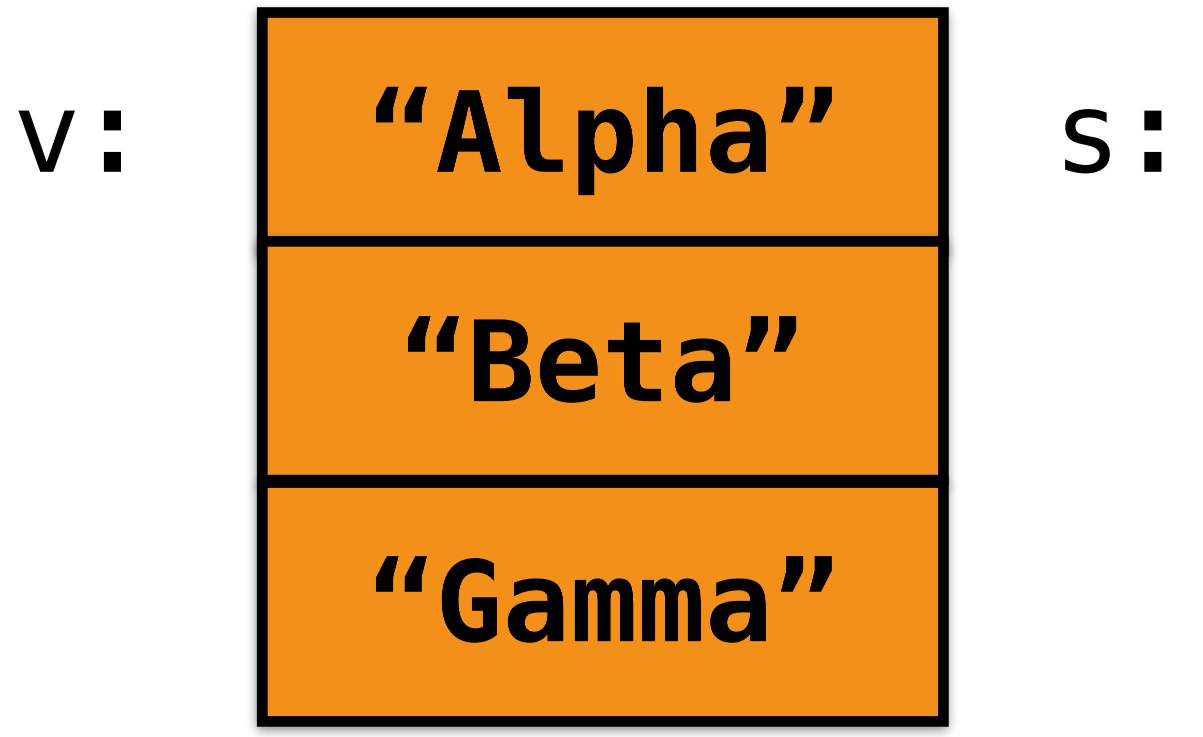


For Loops

```
fn main() {
  let v = vec![format!("Alpha"),
               format!("Beta"),
               format!("Gamma")];
  for s in v {
    println!("{:?}", s);
  }
}
```

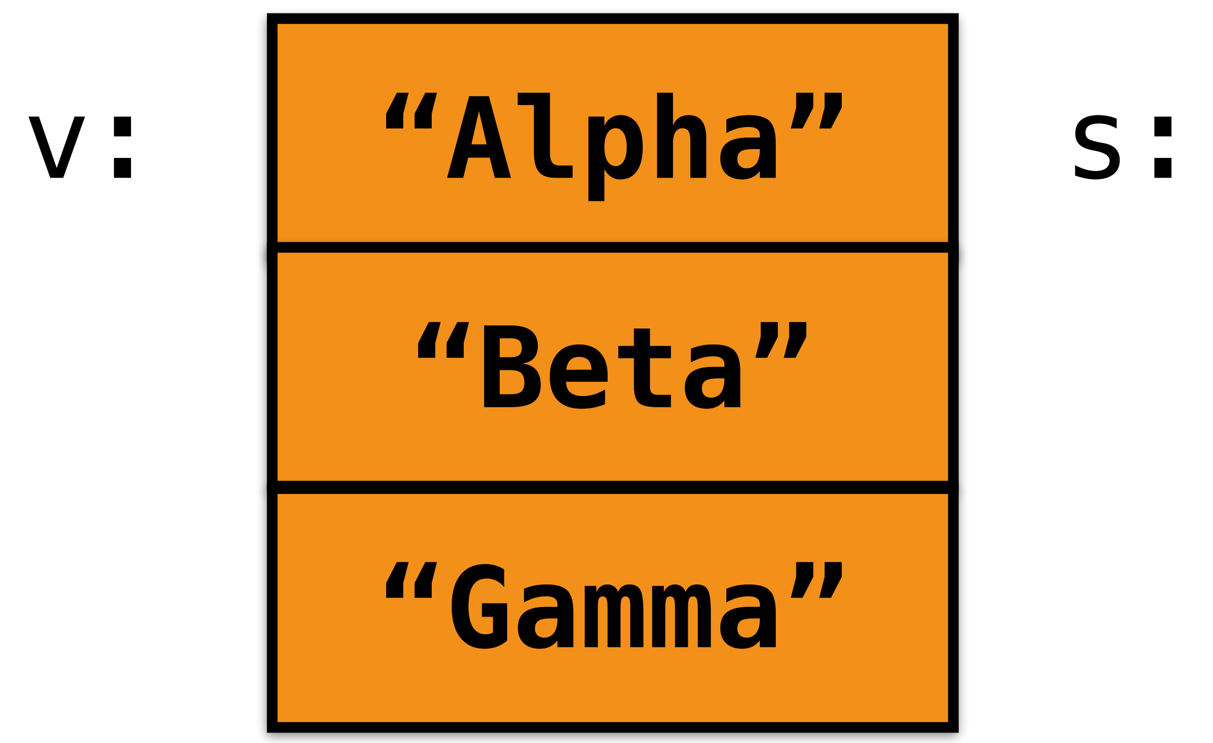
String

Vec<String>



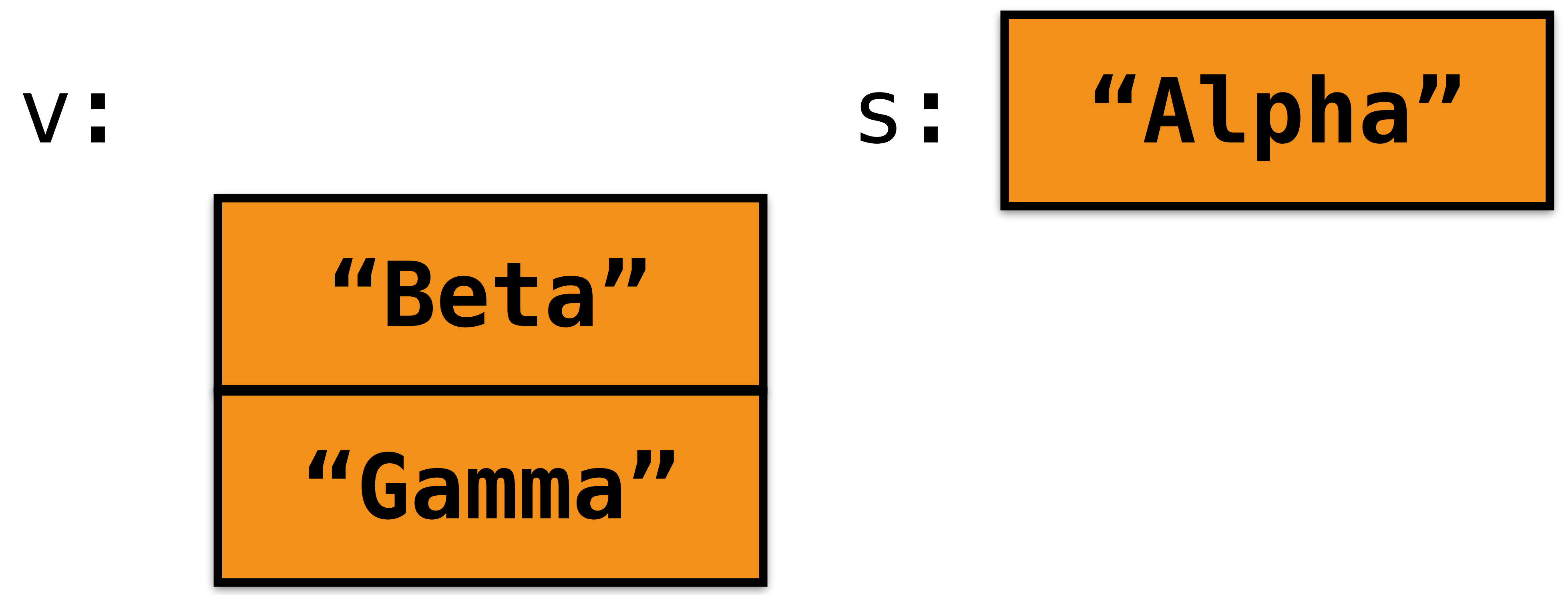
For Loops

```
fn main() {
  let v = vec![format!("Alpha"),
               format!("Beta"),
               format!("Gamma")];
  for s in v {
    println!("{:?}", s);
  }
}
```



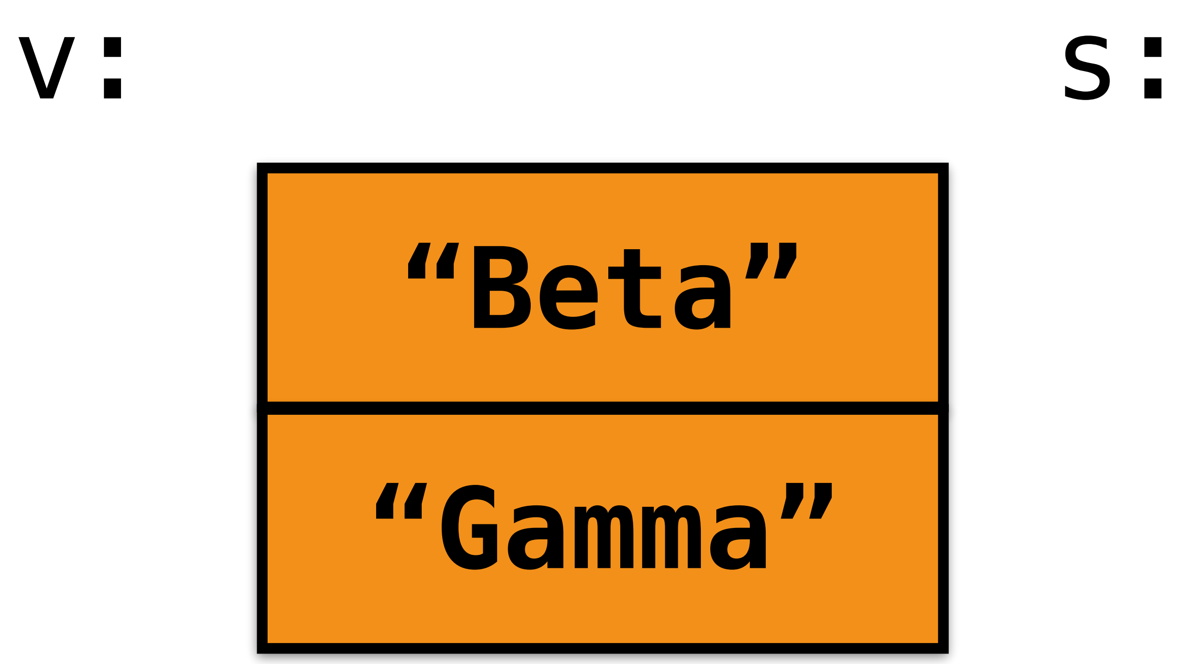
For Loops

```
fn main() {
  let v = vec![format!(“Alpha”),
               format!(“Beta”),
               format!(“Gamma”)];
  for s in v {
    println!(“{:?}”, s);
  }
}
```



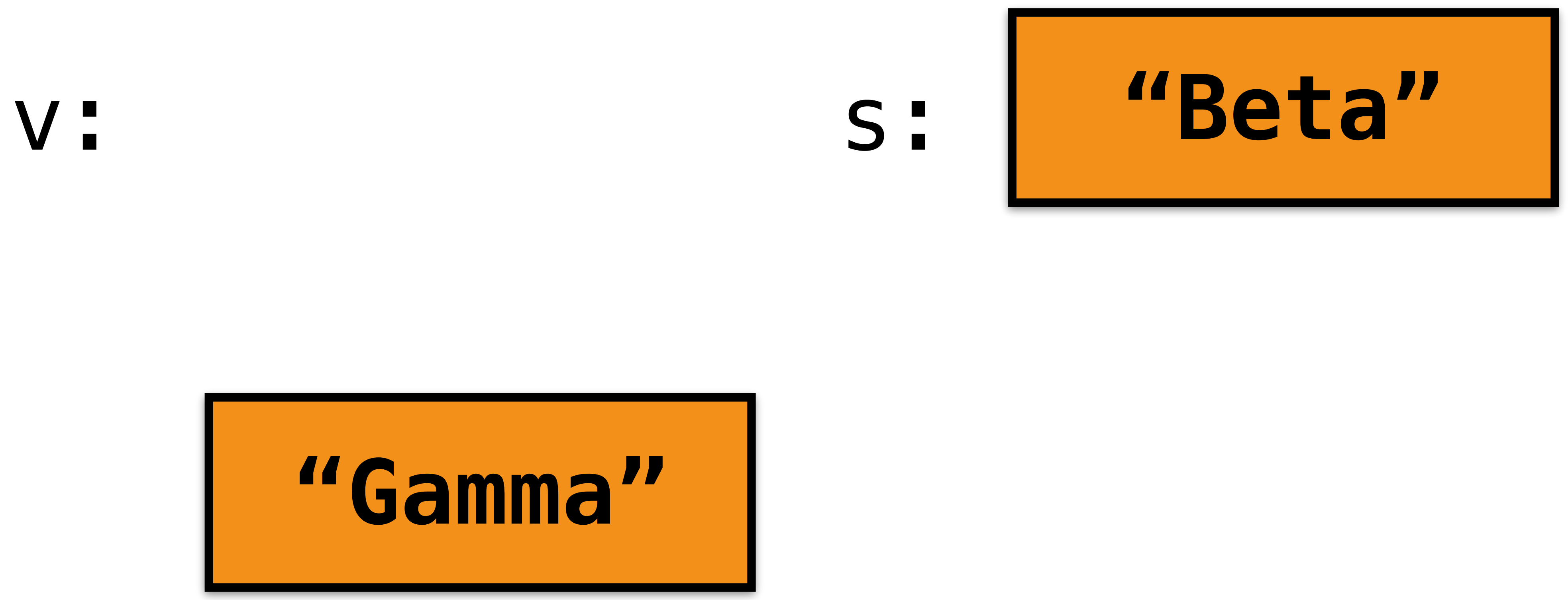
For Loops

```
fn main() {
  let v = vec![format!(“Alpha”),
               format!(“Beta”),
               format!(“Gamma”)];
  for s in v {
    println!(“{:?}”, s);
  }
}
```



For Loops

```
fn main() {
  let v = vec![format!("Alpha"),
               format!("Beta"),
               format!("Gamma")];
  for s in v {
    println!("{:?}", s);
  }
}
```



For Loops

```
fn main() {
  let v = vec![format!("Alpha"),
               format!("Beta"),
               format!("Gamma")];
  for s in v {
    println!("{:?}", s);
  }
}
```

v: s:

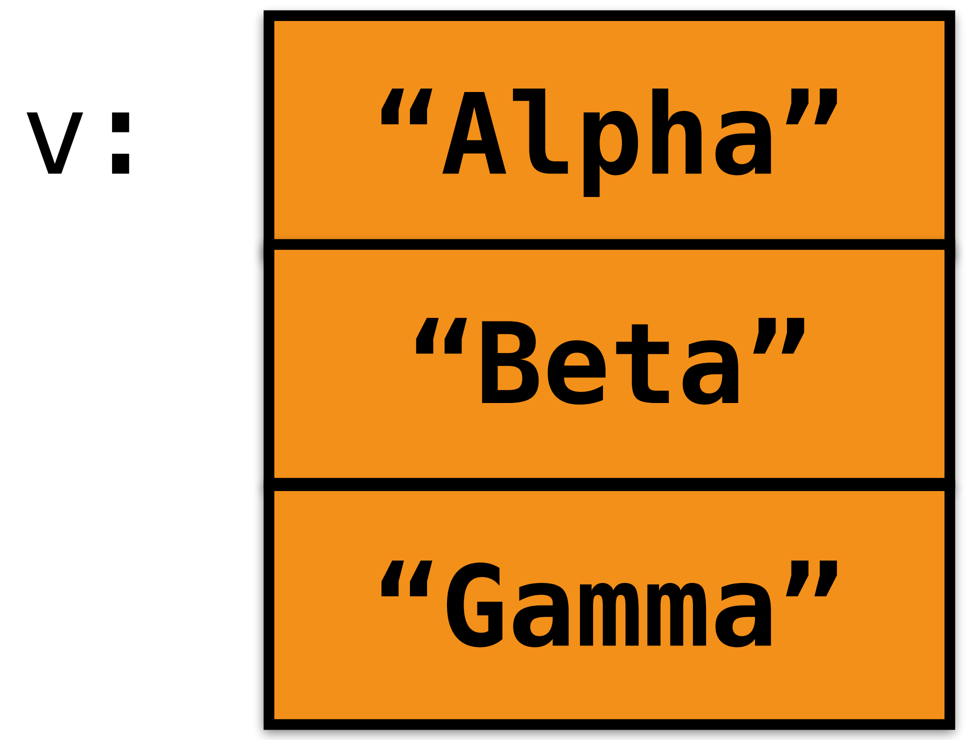
"Gamma"

For Loops

```
fn main() {
  let mut v = vec![format!(“Alpha”),
                   format!(“Beta”),
                   format!(“Gamma”)];

  for s in &v {
    println!(“{:?}”, s);
  }

  for s in &mut v {
    s.push_str(“.”);
  }
}
```

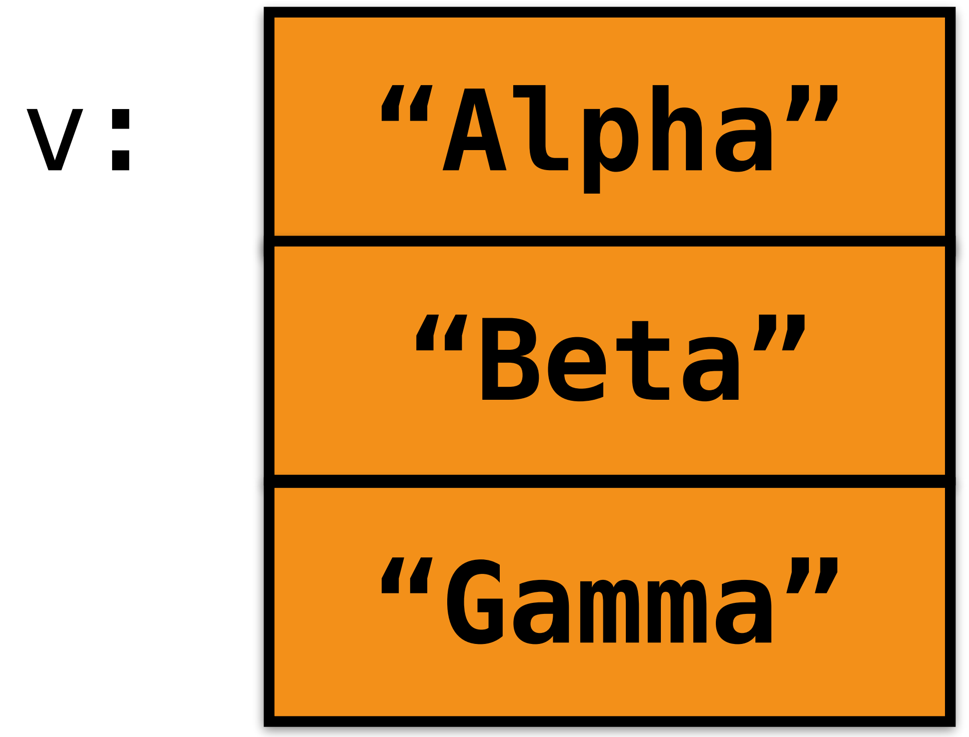


For Loops

```
fn main() {
  let mut v = vec![format!("Alpha"),
                   format!("Beta"),
                   format!("Gamma")];

  for s in &v {
    println!("{:?}", s);
  }

  for s in &mut v {
    s.push_str(".");
  }
}
```

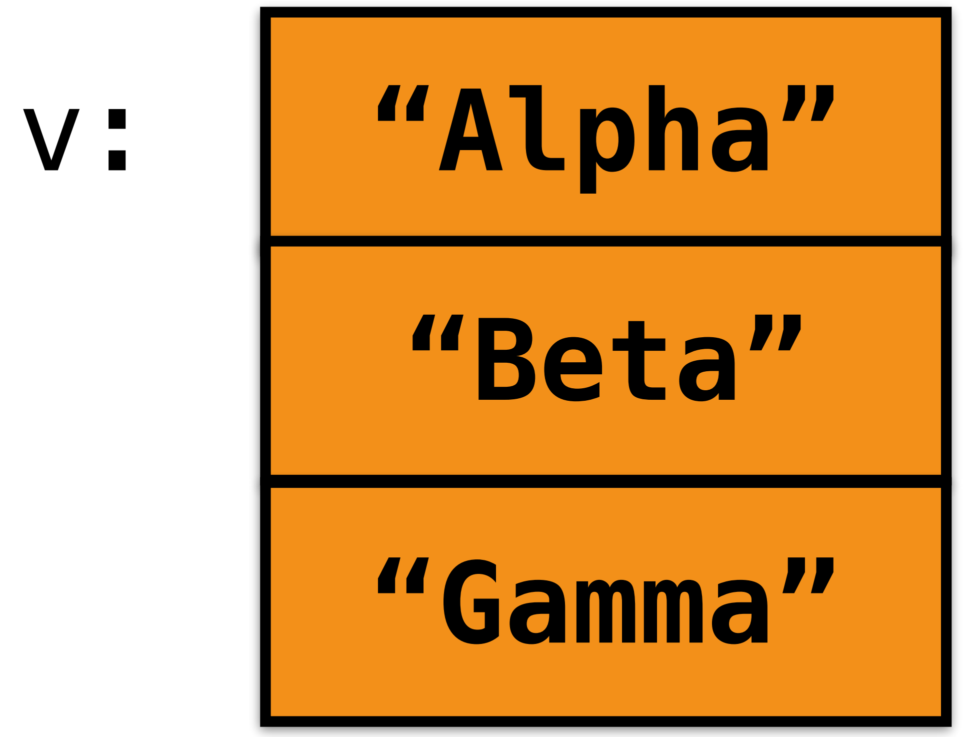


For Loops

```
fn main() {
  let mut v = vec![format!(“Alpha”),
                   format!(“Beta”),
                   format!(“Gamma”)];

  for s in &v {   &Vec<String>
    println!(“{:?}”, s);
  }

  for s in &mut v {
    s.push_str(“.”);
  }
}
```



For Loops

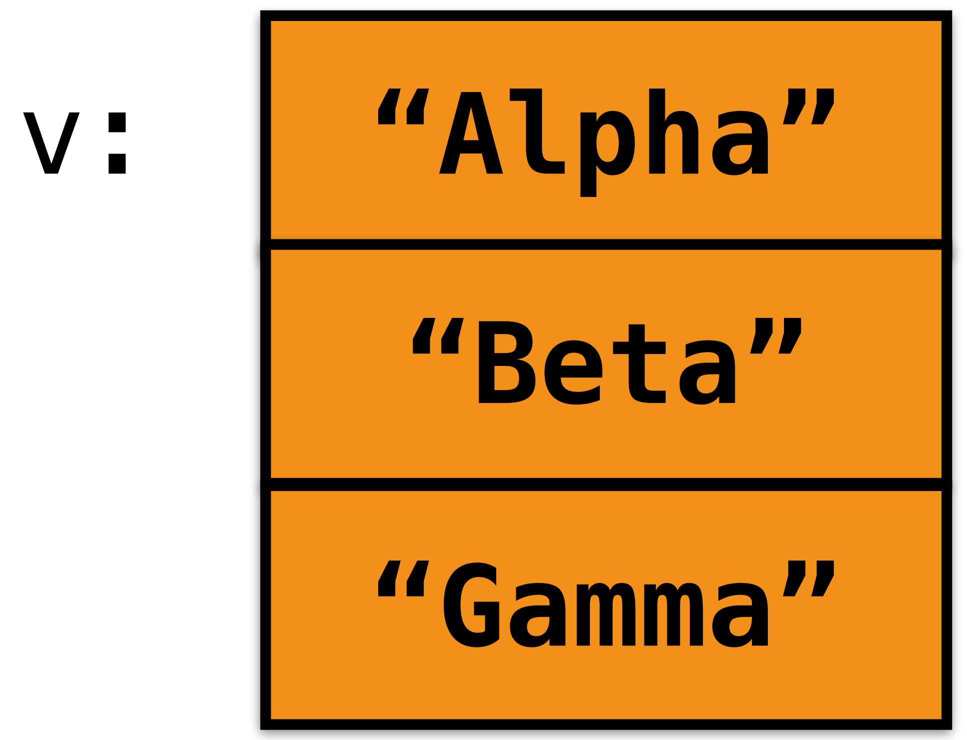
```
fn main() {
  let mut v = vec![format!("Alpha"),
                   format!("Beta"),
                   format!("Gamma")];

  for s in &v {
    println!("{:?}", s);
  }

  for s in &mut v {
    s.push_str(".");
  }
}
```

&String

&Vec<String>

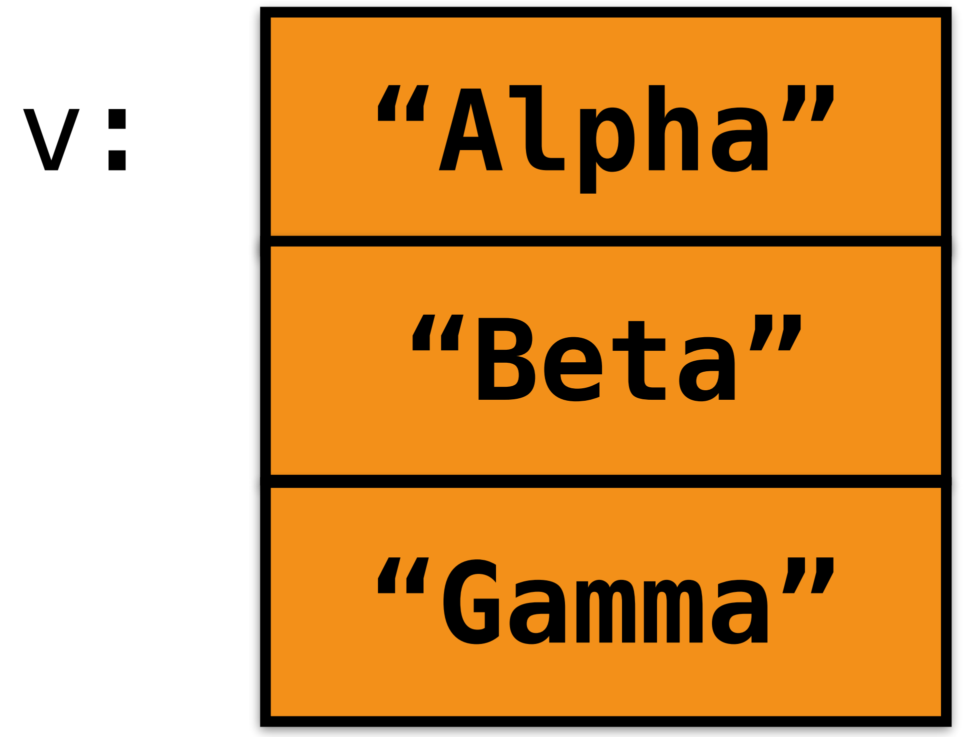


For Loops

```
fn main() {
  let mut v = vec![format!("Alpha"),
                   format!("Beta"),
                   format!("Gamma")];

  for s in &v {
    println!("{:?}", s);
  }

  for s in &mut v {
    s.push_str(".");
  }
}
```

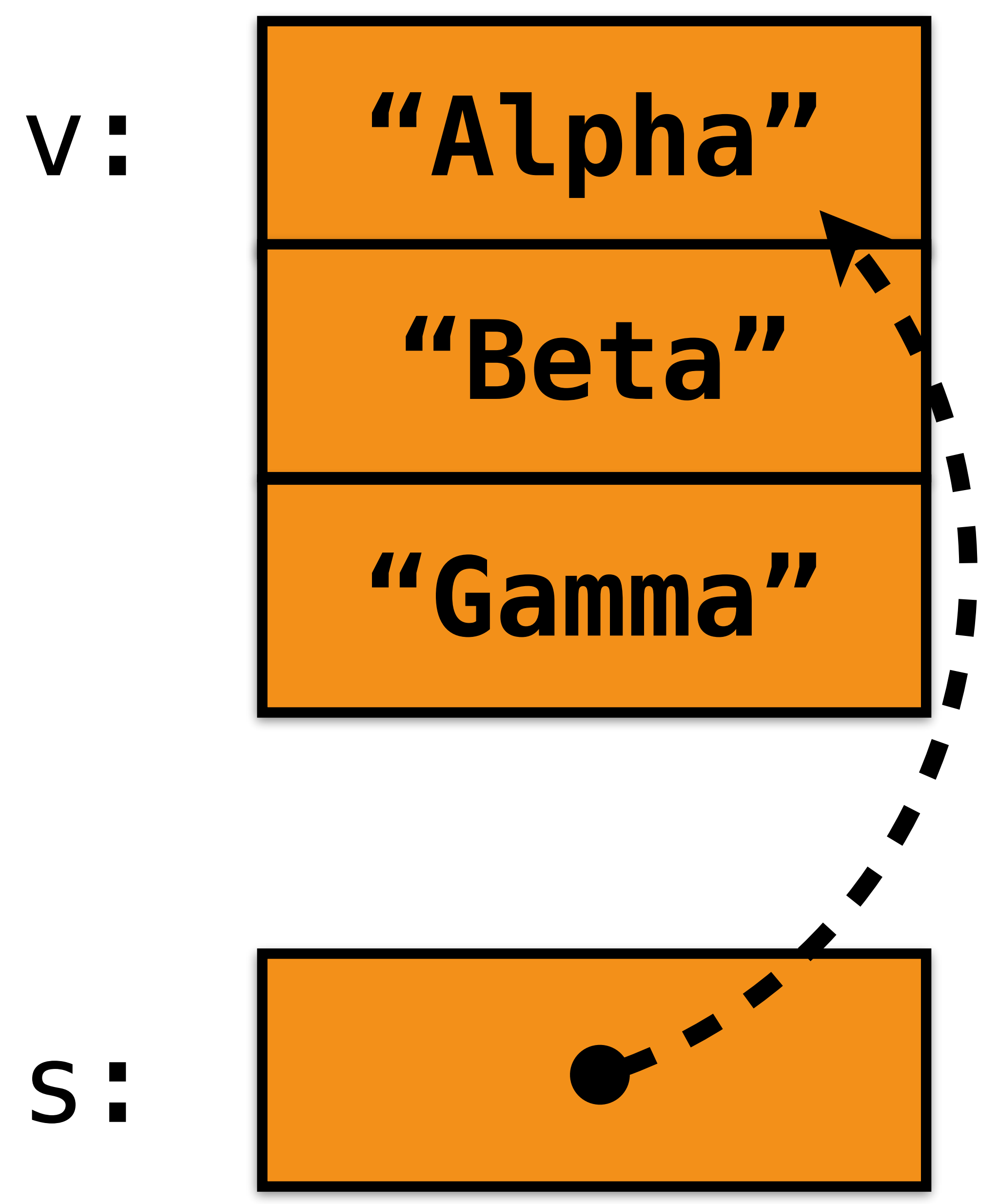


For Loops

```
fn main() {
  let mut v = vec![format!("Alpha"),
                   format!("Beta"),
                   format!("Gamma")];

  for s in &v {
    println!("{:?}", s);
  }

  for s in &mut v {
    s.push_str(".");
  }
}
```

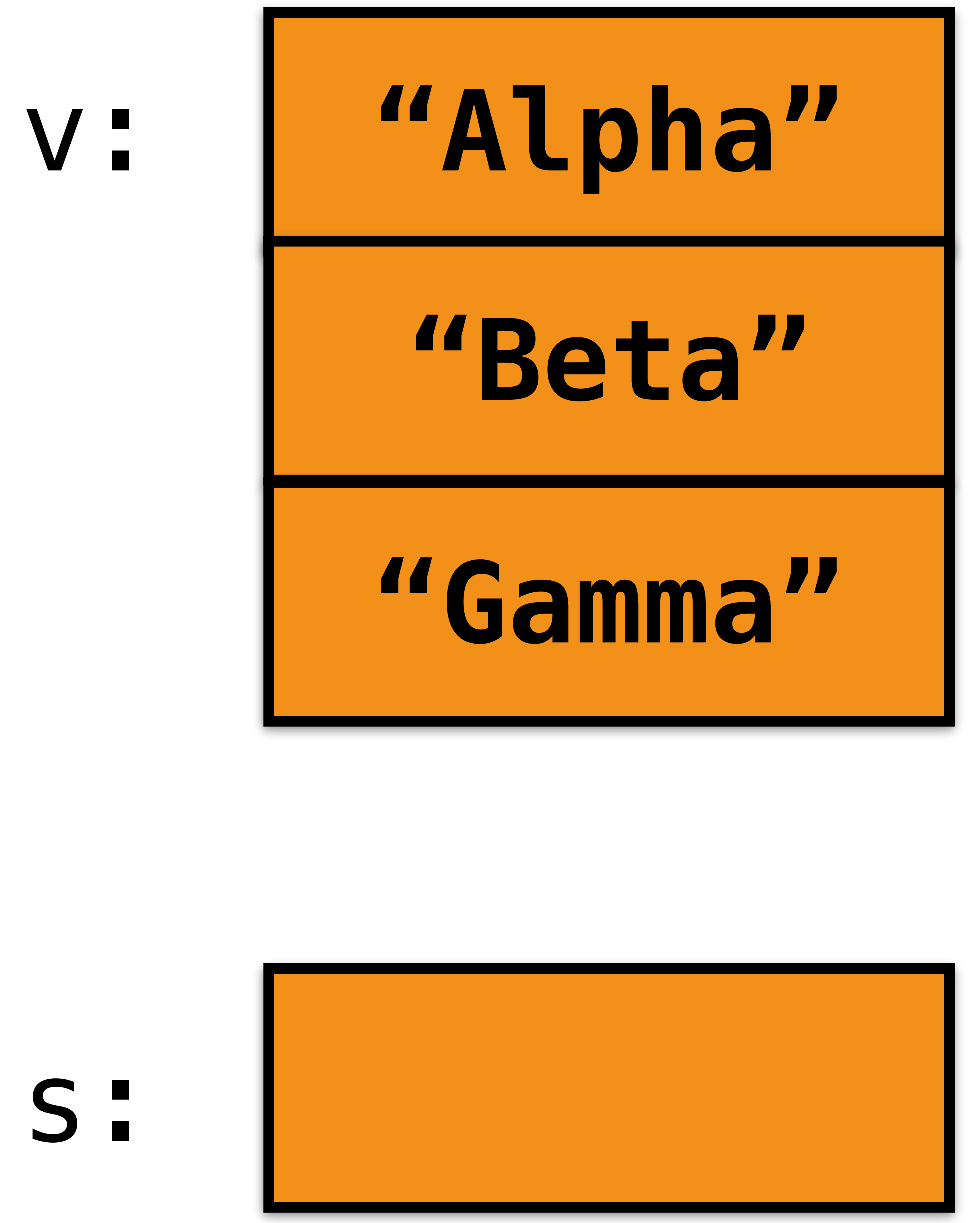


For Loops

```
fn main() {
  let mut v = vec![format!("Alpha"),
                   format!("Beta"),
                   format!("Gamma")];

  for s in &v {
    println!("{:?}", s);
  }

  for s in &mut v {
    s.push_str(".");
  }
}
```

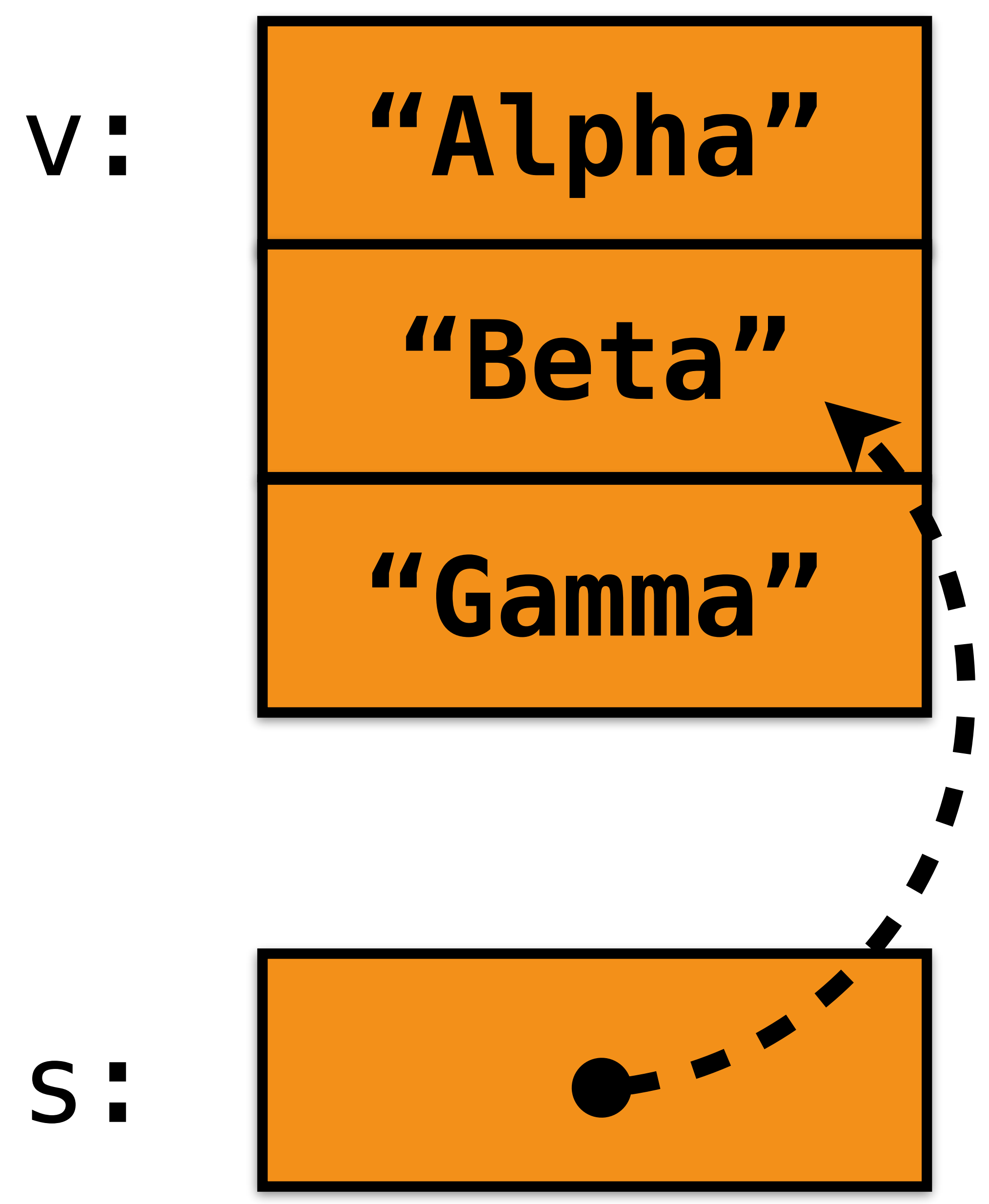


For Loops

```
fn main() {
  let mut v = vec![format!("Alpha"),
                   format!("Beta"),
                   format!("Gamma")];

  for s in &v {
    println!("{:?}", s);
  }

  for s in &mut v {
    s.push_str(".");
  }
}
```



For Loops

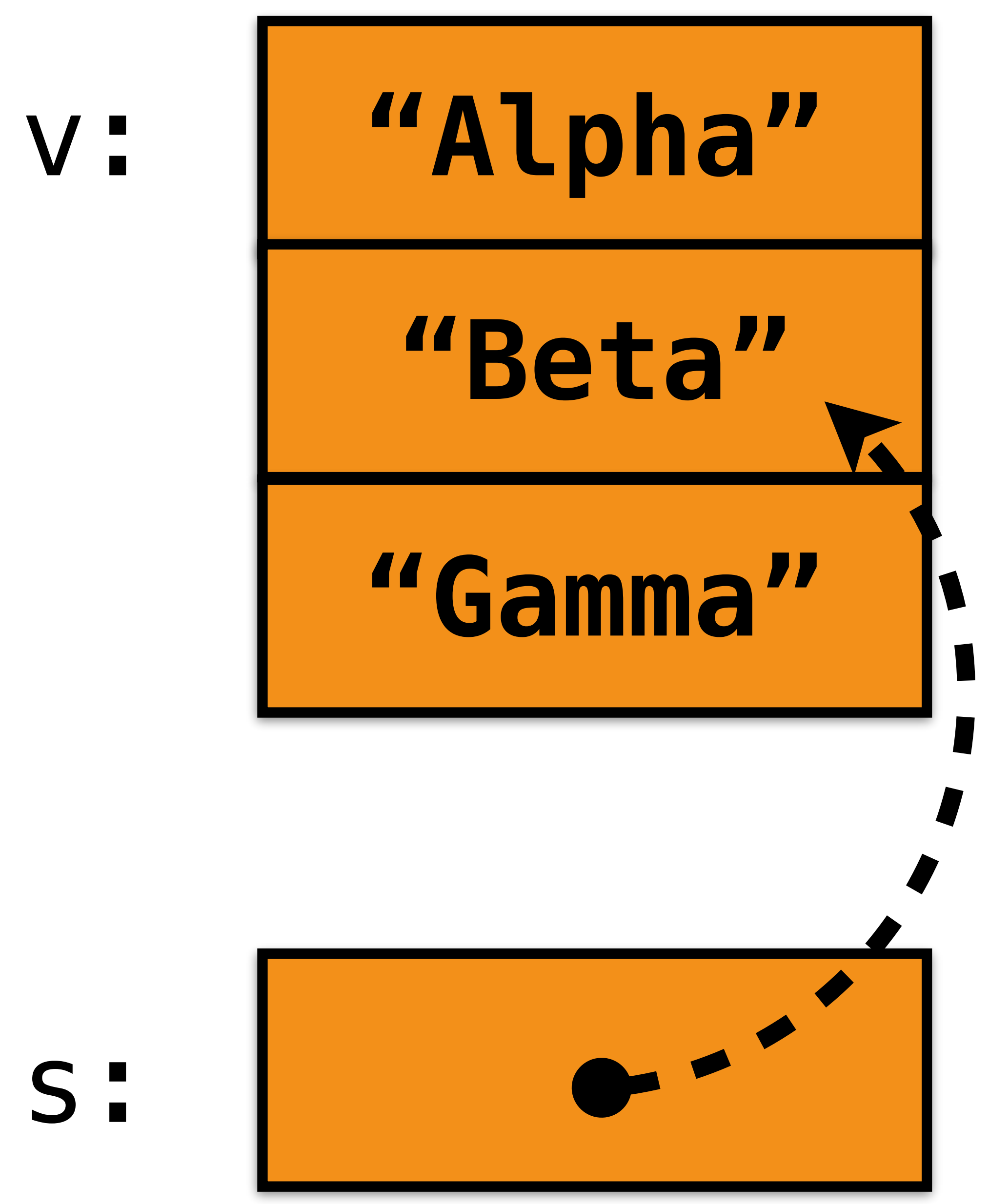
```
fn main() {
  let mut v = vec![format!(“Alpha”),
                   format!(“Beta”),
                   format!(“Gamma”)];

  for s in &v {
    println!(“{:?}”, s);
  }

  for s in &mut v {
    s.push_str(“.”);
  }
}
```

&mut String

&mut Vec<String>

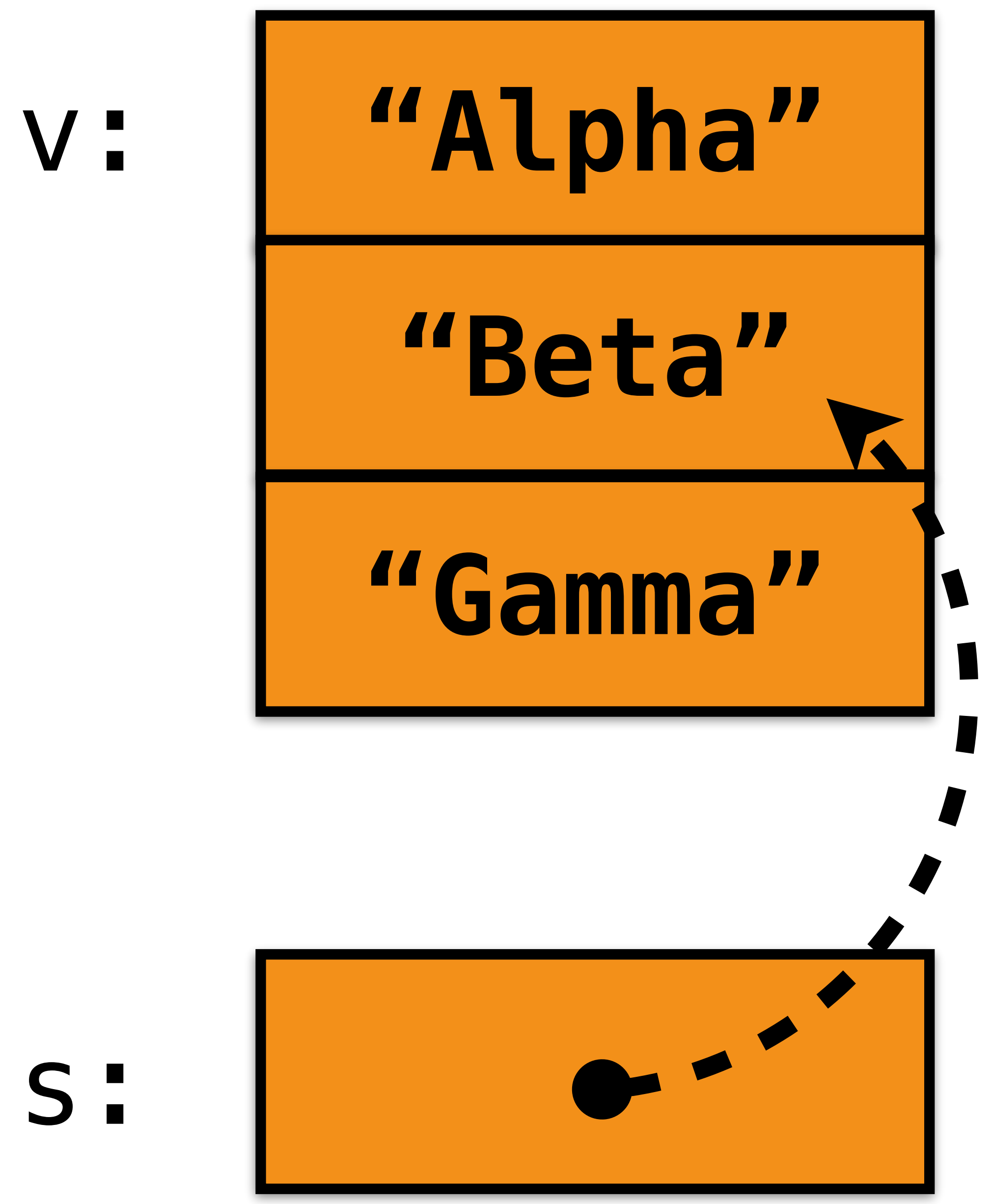


For Loops

```
fn main() {
  let mut v = vec![format!("Alpha"),
                   format!("Beta"),
                   format!("Gamma")];


  for s in &v {
    println!("{:?}", s);
  }

  for s in &mut v {
    s.push_str(".");
  }
}
```



Exercise: **structs**

http://rust-tutorials.com/RustConf17

Implement

```
fn total_price(..)
```

Cheat sheet:

```
for s in v { … }          let mut some_var = 0.0;

for s in &v { … }         some_var += x;

while … { … }             println!(“{:?}”, s);
```

http://doc.rust-lang.org/std