# Tuples

INTERMEDIATE JULIA

**Anthony Markham**
Quantitative Developer

# Tuples - introduction

- An immutable, ordered collection of values.
- Can contain values of any data type.
- Integer indexing, just as with vectors.

# Tuples - why?

## Advantages

- Your code is safer if you use tuples for data that you want to protect.
- Faster than vectors, using less memory.

## Disadvantages

- Unable to sort tuples.
- Unable to append, delete, or change values.

# Tuples - syntax

- Define a tuple using parentheses, with a comma between each value.

```
my_tuple = (10, 20, 30, 40)
```

- A tuple can contain multiple data types.

```
my_mixed_tuple = (10, "Hello", 30, true)
```

# Tuples - indexing

- Tuples are indexed using integers (the same as vectors).

```
my_tuple = (10, 20, 30, 40)
println(my_tuple[1])
```

```
10
```

```
println(my_tuple[2:3])
```

```
(20, 30)
```

# Tuples - immutability

- We can not add or remove values to a tuple after it has been created.
- We can not modify values contained within a tuple.

```
my_tuple = (10, 20, 30, 40, 50)
append!(my_tuple, 60)
```

```
MethodError: no method matching append!(::NTuple{5, Int64}, ::Int64)
```

```
my_tuple = (10, 20, 30, 40, 50)
my_tuple[1] = 15
```

```
MethodError: no method matching setindex!(::NTuple{5, Int64}, ::Int64, ::Int64)
```

# Tuples - immutability

- A tuple will track the number of elements it contains, as this can not change.
- This is reflected in the type of the variable.

```
my_vector = [10, 20, 30, 40, 50]
my_tuple = (10, 20, 30, 40, 50)

println(typeof(my_vector))
println(typeof(my_tuple))
```

```
Vector{Int64}
NTuple{5, Int64}
```

# Tuples - NamedTuple

- A NamedTuple is a tuple with elements that can also be accessed by a name assigned to the value.
- Each value in a NamedTuple is represented by a unique name.
- NamedTuples have the same immutability properties as tuples.

```
my_named_tuple = (name="Anthony", country="Australia", city="Sydney")
println(my_named_tuple[1])
println(my_named_tuple.name)
```

```
Anthony
Anthony
```

# Let's practice!

INTERMEDIATE JULIA

# Dictionaries

INTERMEDIATE JULIA

**Anthony Markham**
Quantitative Developer

# Dictionaries - introduction

- A collection of 'key-value' pairs.
- Each value can be accessed using the corresponding key.
- Mixed data types are allowed.

# Dictionaries - why are they useful?

- Access values using a key, far more intuitive

```
stock = ["AAPL", 131.86, 100000]
println(stock[1])
```

```
AAPL
```

```
# Dictionary definition
stock = Dict("ticker" => "AAPL", "price" => 131.86)
println(stock["ticker"])
```

```
AAPL
```

# Dictionaries - untyped

- Dict keyword to define a dictionary
- Key => value
- Separate key/value pairs with a comma

```
stock = Dict("ticker" => "AAPL", "price" => 131.86)
```

```
Dict{String, Any} with 2 entries:
  "ticker" => "AAPL"
  "price"  => 131.86
```

- Note the data types are mixed and not pre-defined.

# Dictionaries - typed

- Typed dictionaries force a key or value to be a certain data type.

```
stock_typed = Dict{String, Integer}("ticker" => "AAPL", "price" => 131.86)
```

```
MethodError: Cannot `convert` an object of type String to an object of type Integer
```

```
stock_typed = Dict{String, Any}("ticker" => "AAPL", "age" => 131.86)
```

```
Dict{String, Any} with 2 entries:
  "ticker" => "AAPL"
  "age"    => 131.86
```

# Dictionaries - iteration

- Iteration is similar to other data structures that we are familiar with.

```
stock = Dict("ticker" => "AAPL", "price" => 131.86)
for i in stock
    println(i)
end
```

```
Pair{String, Any}("ticker", "AAPL")
Pair{String, Any}("price", 131.86)
```

# Dictionaries - iteration over keys and values

- `keys()` and `values()` return the keys and values of a dictionary.

```
for i in keys(stock)
    println(i)
end
```

```
ticker
price
```

# Dictionaries - iteration via tuple unpacking

- Tuple unpacking the keys and values is another way to iterate over a dictionary.

```
for (ticker, price) in stock
    println(ticker, " ", price)
end
```

```
ticker AAPL
price 131.86
```

# Dictionaries - get()

- Use `get()` to get the value for a given key.

```
get(dictionary_name, dictionary_key, default_value)
```

- The final argument is the default value, the return value if the key is not found.

```
stock = Dict("ticker" => "AAPL", "price" => 131.86)
println(get(stock, "ticker", "Key not found."))
```

```
AAPL
```

- If we do not specify the default value argument and the key is not found, we get an error.

# Dictionaries - modification

- We can add new keys, modify existing values, and delete keys.

```
# Add a new key
stock["volume"] = 62128300
println(stock)
```

```
Dict{String, Any}("ticker" => "AAPL", "price" => 131.86, "volume" => 62128300)
```

```
# Modify an existing value
stock["price"] = 125.27
println(stock)
```

```
Dict{String, Any}("ticker" => "AAPL", "price" => 125.27, "volume" => 62128300)
```

# Let's practice!

INTERMEDIATE JULIA

# Multi-dimensional arrays

INTERMEDIATE JULIA

**Anthony Markham**
Quantitative Developer

datacamp

# 1D array recap

- A vector is simply an array with only one dimension.

```
# Create a 1-D array (vector)
my_1d_array = [1, 2, 3, 4, 5, 6]
```

```
6-element Vector{Int64}:
 1
 2
 3
 4
 5
 6
```

# 2D array structure

- A 2D array is also referred to as a matrix.
- A tabular representation of data, with multiple rows and columns.
- Remove the commas between elements, use the semicolon `;` to denote a new row.

```
# Define a matrix
my_matrix = [1 2 3; 4 5 6]
```

```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

# Indexing a 2D array

- Indexing is similar to vectors, but we need two indexes, not one.
- The first index is the row, and the second is the column.

```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

```
# Return '2' - first row, second column
println(my_matrix[1, 2])
```

```
2
```

# Slicing a 2D array

- The colon `:` operator allows us to select all values in a row or column.
- This is the same as the previous DataFrame slicing that we have seen!

```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

```
# Return the entire third column
println(my_matrix[:, 3])
```

```
[3, 6]
```

# getindex()

- `getindex()` is another way to access elements within an array.
- Pass the variable name and the two indices to return the value at these indices.

```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

```
println(getindex(stock, 1, 2))
```

```
2
```

# 2D array concatenation

- Array concatenation involves combining two arrays together by stacking them.

```
array_1 = [1 2 3; 4 5 6]
array_2 = [7 8 9; 10 11 12]
concat_array = [array_1; array_2]
```

```
4×3 Matrix{Int64}:
  1   2   3
  4   5   6
  7   8   9
 10  11  12
```

# Let's practice!

INTERMEDIATE JULIA

# Structs

INTERMEDIATE JULIA

**Anthony Markham**
Quantitative Developer

datacamp

# Structs - introduction

- Structs are an application of composite types.
- These are like other types, but with as many fields as you want
- We can use a composite type to then create an object, a representation of our type.

# Structs - syntax

- Use the `struct` keyword to define a structure.
- Each field within the `struct` is indented inside the `struct` declaration.

```
struct Person
    age
    height
    location
end
```

```
steve = Person(18, 180, "London")
println(steve.height)
```

```
180
```

# Structs - typeof object

- The type of `steve` is `Person`.

```
struct Person
    age
    height
    location
end
```

```
steve = Person(18, 180, "London")
println(typeof(steve))
```

```
Person
```

# Structs - demonstrate immutability

- A struct is immutable by default.

```
# It's Steve's birthday!
steve = Person(18, 180, "London")
steve.age = 19
```

```
setfield!: immutable struct of type Person cannot be changed
```

# Structs - mutable structs

- Use the `mutable` keyword to create a mutable struct.

```
mutable struct Person
    age
    height
    location
end
```

```
steve = Person(18, 180, "London")
steve.age = 19
println(steve)
```

```
Person(19, 180, "London")
```

# Structs - typed structs

- We can specify a data type for each field in our struct definition.

```
mutable struct Person
    age::Int64
    height::Int64
    location::String
end
```

```
steve = Person(18.5, 180, "London")
```

```
InexactError: Int64(18.5)
```

# Let's practice!

INTERMEDIATE JULIA