ÖGOR Summer-Workshop for PhD-candidates and Post-Docs

An introduction to Julia and JuMP for Operations Research

Prof. Dr. Xavier Gandibleux

Nantes Université – France
Département Informatique – Faculté des Sciences et Techniques

Topic 8

Data structure (part 2)

Tuple, dictionary, set

Constructing and accessing to a tuple

A tuple is an immutable array with 1 dimension

Definition:

Let a collection of n items: $item_1$, $item_2$, ..., $item_n$

```
(item_1, item_2, ..., item_n)
```

Examples:

```
julia> ("Austria", "France", "Belgium")
```

```
julia> primeNumbers = (2, 3, 5, "seven", 11)
```

Accessing to a tuple:

Same of a vector.

Constructing and accessing to a tuple

A tuple is an immutable array with 1 dimension

Definition:

Let a collection of n items: $item_1$, $item_2$, ..., $item_n$

```
(item_1, item_2, ..., item_n)
```

Examples:

```
julia> ("Austria", "France", "Belgium")
```

```
julia> primeNumbers = (2, 3, 5, "seven", 11)
```

Accessing to a tuple:

Same of a vector.

Constructing and accessing to a tuple

A tuple is an immutable array with 1 dimension

Definition:

Let a collection of n items: $item_1$, $item_2$, ..., $item_n$

$$(item_1, item_2, \ldots, item_n)$$

Examples:

```
julia> ("Austria", "France", "Belgium")
```

```
julia> primeNumbers = (2, 3, 5, "seven", 11)
```

Accessing to a tuple:

Same of a vector.

Constructing and accessing to a tuple

A tuple is an immutable array with 1 dimension

Definition:

Let a collection of n items: $item_1$, $item_2$, ..., $item_n$

$$(item_1, item_2, \ldots, item_n)$$

Examples:

```
julia> ("Austria", "France", "Belgium")
```

```
julia> primeNumbers = (2, 3, 5, "seven", 11)
```

Accessing to a tuple:

Same of a vector.

Constructing and accessing to a tuple

A tuple is immutable

```
julia> vct = [2,7]                                    # array
```

```
julia> push!(vct,3)
```

Right!

```
julia> tpl = (2,7)                                    # tuple
```

```
julia> push!(tpl,3)
```

Wrong!

Constructing and accessing to a dictionary (1/3)

A dictionary is an 1D array where the sequence of elements is not ordered

Definition:

Let a collection of *n* couple *key-value*

```
Dict(key_1 => value_1, key_2 => value_2, ...)
```

Example:

```
julia> Dict(43 => "Austria", 32 => "Belgium", 33 => "France")
```

Constructing and accessing to a dictionary (1/3)

A dictionary is an 1D array where the sequence of elements is not ordered

Definition:

Let a collection of *n* couple *key-value*

```
Dict(key_1 => value_1, key_2 => value_2, ...)
```

Example:

```
julia> Dict(43 => "Austria", 32 => "Belgium", 33 => "France")
```

Constructing and accessing to a dictionary (1/3)

A dictionary is an 1D array where the sequence of elements is not ordered

Definition:

Let a collection of *n* couple *key-value*

```
Dict(key_1 => value_1, key_2 => value_2, ...)
```

Example:

```
julia> Dict(43 => "Austria", 32 => "Belgium", 33 => "France")
```

Constructing and accessing to a dictionary (2/3)

Empty dictionary:

```
Dict()
```

```
julia> code = Dict()
```

Add an entry:

```
nameDictionary[key] = value
```

```
julia> code[49] = "Germany"
```

Delete an entry:

```
pop!(nameDictionary,key)
```

```
julia> pop!(code,33)
```

Constructing and accessing to a dictionary (3/3)

Return an iterator over all keys in a dictionary:

```
keys(dict)
```

```
julia> keys(code)
```

Return an iterator over all values in a dictionary:

```
values(dict)
```

```
julia> values(code)
```

Constructing and accessing to a set (1/2)

A set is a collection of unordered, unique values

Empty (zero-elements) set:

```
Set()
```

```
julia> a = Set()
```

Initialise a set with values:

```
nameSet([value_1, value_2, ..., value_n])
```

```
julia> a = Set([1,2,2,3,4])
```

Constructing and accessing to a set (1/2)

A set is a collection of unordered, unique values

Empty (zero-elements) set:

```
Set()
```

```
julia> a = Set()
```

Initialise a set with values:

```
nameSet([value_1, value_2, ..., value_n])
```

```
julia> a = Set([1,2,2,3,4])
```

Constructing and accessing to a set (2/2)

Some operations:

```
union(set1,set2)
```

```
julia> union([1, 2], [3, 4])
```

```
intersect(set1,set2)
```

```
julia> intersect([1, 2, 3], [3, 4, 5])
```

```
setdiff(set1,set2)
```

```
julia> setdiff([1,2,3], [3,4,5])
```

Summary

	mutable	ordered collections
array	yes	yes
tuple	no	yes
dictionary	yes	no
set	no	no

Data structure

Strings

Constructing and accessing to a string (1/4)

A string is an immutable finite sequence of characters

```
julia> s1 = "A string."                # ASCII characters
```

```
julia> s2 = """Also a string."""
```

```
julia> """A "special" string."""
```

```
julia> "\u2200 x \u2203 y"          # unicode characters
```

Constructing and accessing to a string (1/4)

A string is an immutable finite sequence of characters

```
julia> s1 = "A string."                    # ASCII characters
```

```
julia> s2 = """Also a string."""
```

```
julia> """A "special" string."""
```

```
julia> "\u2200 x \u2203 y"                 # unicode characters
```

Constructing and accessing to a string (1/4)

A string is an immutable finite sequence of characters

```
julia> s1 = "A string."                    # ASCII characters
```

```
julia> s2 = """Also a string."""
```

```
julia> """A "special" string."""
```

```
julia> "\u2200 x \u2203 y"          # unicode characters
```

Constructing and accessing to a string (2/4)

String interpolation

Use the $ sign

- to insert existing variables into a string and
- to evaluate expressions within a string.

```
julia> city = "Linz"
```

```
julia> "Hello $city"
```

```
julia> zc = 4020
```

```
julia> "zc of $city:  $zc, $(zc+10), $(zc+20)"
```

Constructing and accessing to a string (2/4)

String interpolation

Use the $ sign

- to insert existing variables into a string and
- to evaluate expressions within a string.

```
julia> city = "Linz"
```

```
julia> "Hello $city"
```

```
julia> zc = 4020
```

```
julia> "zc of $city:  $zc, $(zc+10), $(zc+20)"
```

Constructing and accessing to a string (3/4)

String concatenation

Two manners:

- use the `string()` function:

```
julia> string("Hello ", city)
```

`string()` converts non-string inputs to strings

- use the `*` operator:

```
julia> "Hello " * city
```

Constructing and accessing to a string (4/4)

```
julia> s1 = "A string."
```

```
julia> s1[3]
```

```
julia> s1[3:5]
```

```
julia> s1[begin:3]
```

```
julia> s1[3:end]
```

```
julia> s1[3] = 'b'                                    # error!
```

```
julia> s1 = s1[begin:3] * "w" * s1[5:end]
```

Constructing and accessing to a string (4/4)

```
julia> s1 = "A string."
```

```
julia> s1[3]
```

```
julia> s1[3:5]
```

```
julia> s1[begin:3]
```

```
julia> s1[3:end]
```

```
julia> s1[3] = 'b'                                    # error!
```

```
julia> s1 = s1[begin:3] * "w" * s1[5:end]
```

Constructing and accessing to a string (4/4)

```
julia> s1 = "A string."
```

```
julia> s1[3]
```

```
julia> s1[3:5]
```

```
julia> s1[begin:3]
```

```
julia> s1[3:end]
```

```
julia> s1[3] = 'b'                                    # error!
```

```
julia> s1 = s1[begin:3] * "w" * s1[5:end]
```

Constructing and accessing to a string (4/4)

```
julia> s1 = "A string."
```

```
julia> s1[3]
```

```
julia> s1[3:5]
```

```
julia> s1[begin:3]
```

```
julia> s1[3:end]
```

```
julia> s1[3] = 'b'                                    # error!
```

```
julia> s1 = s1[begin:3] * "w" * s1[5:end]
```

Review and exercises

(notebook)