

Практическая работа

Julia предрасположена к процедурному программированию с элементами функционального, поэтому основным методом выражения поведения в программе являются функции.

Существует несколько способов инициализировать функцию.

Основной, это использовать семантику именованной функции

```

1 md"""
2
3 # Практическая работа
4
5 Julia предрасположена к процедурному программированию с элементами функционального,
6 поэтому основным методом выражения поведения в программе являются функции.
7
8 Существует несколько способов инициализировать функцию.
9
10 Основной, это использовать семантику именованной функции
11 """

```

`say42` (generic function with 1 method)

```

1 function say42(question)
2     return "Ответ на вопрос $question: 42!"
3 end

```

Теперь вызов функции доступен по ее имени

```

1 md"""
2 Теперь вызов функции доступен по ее имени
3 """

```

"Ответ на вопрос Вселенной и всего такого: 42!"

```
1 say42("Вселенной и всего такого")
```

Существует второй способ инициализации функции - строчный, схожий с математической нотацией

```

1 md"""
2 Существует второй способ инициализации функции - строчный, схожий с математической
3 нотацией
4 """

```

`say42again` (generic function with 1 method)

```
1 say42again(question) = println("Повторяю. $(say42(question))")
```

```
1 say42again("Вселенной и всего такого")
```

Повторяю. Ответ на вопрос Вселенной и всего такого: 42!



И еще один вариант для инициализации анонимной функции.

Анонимная функция в программировании — это функция, определенная без присвоения ей идентификационного имени. Такие функции часто используются для создания компактных блоков кода, которые можно передать как аргументы в другие функции, используемые для кратковременных операций, таких как обработка элементов коллекций, событий или выполнение обратных вызовов (callbacks).

```
1 md"""
2 И еще один вариант для инициализации анонимной функции.
3 > Анонимная функция в программировании – это функция, определенная без присвоения
   ей идентификационного имени. Такие функции часто используются для создания
   компактных блоков кода, которые можно передать как аргументы в другие функции,
   используемые для кратковременных операций, таких как обработка элементов коллекций,
   событий или выполнение обратных вызовов (callbacks).
4 """
```

```
f1 = #7 (generic function with 1 method)
```

```
1 f1 = x -> x ^2
```

```
4
```

```
1 f1(2)
```

Типизация функций

Julia не стремится заставить программиста указывать типы данных, поэтому вынуждена использовать утиную типизацию для динамических нетипизированных данных.

Утиная типизация (Duck typing) — это метод в программировании, при котором типы не выбираются строго на основании их объявления. Вместо этого объект считается принадлежащим к определенному типу в зависимости от наличия у него определенных методов и свойств, независимо от формального наследования.

Принцип утиной типизации можно описать фразой: «Если это выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, утка».

Это означает, что если Julia сможет сопоставить признаки данных и объявленные типы, то будет вызвана реализация поведения, связанная с сопоставленным типом, вне зависимости от логики работы алгоритма.

Например, функция отвечающая на вопрос `say42` нетипизированна, но исходя из логики алгоритма, ожидается строковый аргумент. Но функция сработает, если мы передадим и число

```
1 md"""
2
3 ### Типизация функций
4 Julia не стремится заставить программиста указывать типы данных, поэтому вынуждена
   использовать утиную типизацию для динамических нетипизированных данных.
5
6 > Утиная типизация (Duck typing) – это метод в программировании, при котором типы
   не выбираются строго на основании их объявления. Вместо этого объект считается
   принадлежащим к определенному типу в зависимости от наличия у него определенных
   методов и свойств, независимо от формального наследования. Принцип утиной типизации
   можно описать фразой: «Если это выглядит как утка, плавает как утка и крякает как
   утка, то это, вероятно, утка».
7
8 Это означает, что если Julia сможет сопоставить признаки данных и объявленные типы,
   то будет вызвана реализация поведения, связанная с сопоставленным типом, вне
   зависимости от логики работы алгоритма.
9
10 Например, функция отвечающая на вопрос `say42` нетипизированна, но исходя из логики
    алгоритма, ожидается строковый аргумент. Но функция сработает, если мы передадим и
    число
11 """
```

"Ответ на вопрос 42: 42!"

```
1 say42(42)
```

Это происходит, потому что и у чисел и у строк есть поведение, которое позволяет их печатать.

Рассмотрим другой пример

```
1 md"""
2 Это происходит, потому что и у чисел и у строк есть поведение, которое позволяет их
3
4 Рассмотрим другой пример
5 """
```

42×42 Matrix{Float64}:

8.94707	10.7721	8.71428	9.31622	...	11.7136	11.6066	11.2154	8.47664
10.0572	11.3948	10.5139	11.4691		11.3634	12.0613	12.4269	10.5249
9.223	11.9046	10.4248	11.3505		11.1658	10.8835	11.7779	9.85543
9.72076	12.0815	10.3995	11.4529		11.2232	11.428	12.4727	9.54891
8.8736	10.9223	11.1113	10.4928		11.7508	11.6485	11.121	9.56681
9.32753	12.1847	11.4231	11.0089	...	11.7062	12.1473	13.4429	9.87218
9.81109	12.3676	9.40983	11.2167		12.5559	11.5068	12.4175	10.2321
:					:			:
10.5019	12.8394	10.3333	11.7941		12.6742	12.087	12.8474	10.1567
8.89985	10.5496	10.6935	10.9954		11.7088	12.3062	11.5413	10.4302
6.36767	8.84889	8.14185	8.81725		8.88297	8.56734	9.23706	7.99184
8.89508	12.7494	10.5595	11.7597		12.8618	12.0549	11.8259	11.0109
10.701	11.5452	11.5423	12.9931	...	13.6547	12.8912	13.6464	11.1342
7.44263	8.63777	8.11352	9.69327		9.10761	9.15025	9.8028	8.93594

```
1 f1(rand(42, 42))
```

Поскольку для матриц так же реализовано поведение возведения в степень, то функция работает корректно

```
1 md"""
2 Поскольку для матриц так же реализовано поведение возведения в степень, то функция
3
4 работает корректно
5 """
```

Мутирующие и не мутирующие функции

По соглашению с разработчиками языка Julia, функции оканчивающиеся на ! являются мутирующими, а те, которые не имеют в имени восклицательного знака – не мутирующими. Это не значит, что это соблюдается в автоматическом режиме, поэтому стоит относится внимательнее к определению функции (читайте документацию).

Для примера рассмотрим разницу между мутирующей функцией и не мутирующей функцией.

```
1 md"""
2 ### Мутирующие и не мутирующие функции
3
4 По соглашению с разработчиками языка Julia, функции оканчивающиеся на `!` являются
5 мутирующими, а те, которые не имеют в имени восклицательного знака -- не
6 мутирующими. Это не значит, что это соблюдается в автоматическом режиме, поэтому
7 стоит относится внимательнее к определению функции (читайте документацию).
8 """
```

```
data = [3, 5, 1]
```

```
1 data = [3, 5, 1]
```

```
[1, 3, 5]
```

```
1 sort(data)
```

sort возвращает отсортированную последовательность. При этом оригинальные данные остаются без изменений

```
1 md"""
```

```
2 'sort' возвращает отсортированную последовательность. При этом оригинальные данные остаются без изменений
```

```
3 """
```

```
[3, 5, 1]
```

```
1 data
```

Но если вызвать sort!, то data изменится

```
1 md"""
```

```
2 Но если вызвать 'sort!', то 'data' изменится
```

```
3 """
```

```
[1, 3, 5]
```

```
1 sort!(data)
```

```
[1, 3, 5]
```

```
1 data
```

Некоторые функции высшего порядка

Функции высшего порядка — это функции, которые могут принимать другие функции в качестве аргументов и/или возвращать их как результат своей работы. Этот подход используется для создания более абстрактных уровней программирования, позволяет упростить композицию функций, повысить модульность кода и ускорить разработку за счет повторного использования готовых решений.

`map` — одна из классических функций высшего порядка, реализацию которой можно встретить в других языках. Эта функция принимает два аргумента: другую функцию и последовательность. `map` применяет функцию к членам последовательности.

```

1 md"""
2 ### Некоторые функции высшего порядка
3
4 > Функции высшего порядка — это функции, которые могут принимать другие функции в качестве аргументов и/или возвращать их как результат своей работы. Этот подход используется для создания более абстрактных уровней программирования, позволяет упростить композицию функций, повысить модульность кода и ускорить разработку за счет повторного использования готовых решений.
5
6
7 'map' — одна из классических функций высшего порядка, реализацию которой можно встретить в других языках. Эта функция принимает два аргумента: другую функцию и последовательность. 'map' применяет функцию к членам последовательности.
8 """
9 """

```

[1, 4, 9, 16]

```
1 map(f1, [1, 2, 3, 4])
```

Для сокращения шаблонного кода, предлагается использовать анонимные функции в качестве аргументов, если это не вредит читаемости

```

1 md"""
2 Для сокращения шаблонного кода, предлагается использовать анонимные функции в качестве аргументов, если это не вредит читаемости
3 """

```

[1, 0, 1, 0, 1, 0]

```
1 map(x -> x % 2, [1, 2, 3, 4, 5, 6])
```

Еще одна классическая функция высшего порядка — `reduce`. Эта функция также принимает два аргумента: другую функцию и последовательность. `reduce` осуществляет операцию свертки, применяя функцию к членам последовательности и аккумулируя результат.

```

1 md"""
2 Еще одна классическая функция высшего порядка -- 'reduce'. Эта функция также принимает два аргумента: другую функцию и последовательность. 'reduce' осуществляет операцию свертки, применяя функцию к членам последовательности и аккумулируя результат.
3 """

```

10

```
1 reduce(max, [1, 3, 10, 7, 3, 6])
```

`broadcast` – функция высшего порядка, схожая с `map`. Эта функция используется тогда, когда оригинальная последовательность может не поддерживать поэлементное вычисление.

Например, вектор не поддерживает возведение компонент в степень, поэтому, если мы вызовем `f1` на векторе, то получим ошибку

```
1 md"""
2 `broadcast` -- функция высшего порядка, схожая с `map`. Эта функция используется
   тогда, когда оригинальная последовательность может не поддерживать поэлементное
   вычисление. Например, вектор не поддерживает возведение компонент в степень,
   поэтому, если мы вызовем `f1` на векторе, то получим ошибку
3 """
```

Error message

```
MethodError: no method matching ^(::Vector{Int64}, ::Int64)
```

The function `^` exists, but no method is defined for this combination of argument types.

Closest candidates are:

```
^(::Float16, ::Integer)
@ Base math.jl:1234
^(::BigInt, ::Integer)
@ Base gmp.jl:649
^(::Regex, ::Integer)
@ Base regex.jl:895
...
-----
```

Stack trace

Here is what happened, the most recent locations are first:

```
1. literal_pow
   from intfuncs.jl:389

2. anonymous function(x::Vector{...}) ...show types...
   from Other cell: line 1
      1 f1 = x -> x ^2
      cell preview

3. Show more...
```

```
1 f1([1, 3, 4])
```

Для того чтобы это избежать, используется `broadcast`, которая принудительно применяет функцию поэлементно к членам последовательности

- 1 `md"""`
- 2 Для того чтобы это избежать, используется '`broadcast`', которая принудительно применяет функцию поэлементно к членам последовательности
- 3 `"""`

`[1, 9, 16]`

- 1 `broadcast(f1, [1, 3, 4])`

Для удобства записи вызов `broadcast` можно заменить синтаксическим сахаром в виде точечной записи.

Синтаксический сахар в программировании — это особенности синтаксиса языка, предназначенные для упрощения записи кода, делая его более лаконичным и легким для восприятия, но не добавляющие новые функции к языку. Примеры синтаксического сахара включают различные краткие формы записи условных операций, циклов и других конструкций. Эти особенности улучшают читаемость и упрощают написание кода, но транслятор или компилятор сводит их к более простым базовым конструкциям перед выполнением.

Например, можно вызвать `f1` для вектора не записывая `broadcast` в явном виде

- 1 `md"""`
- 2 Для удобства записи вызов '`broadcast`' можно заменить синтаксическим сахаром в виде точечной записи.
- 3
- 4 > Синтаксический сахар в программировании — это особенности синтаксиса языка, предназначенные для упрощения записи кода, делая его более лаконичным и легким для восприятия, но не добавляющие новые функции к языку. Примеры синтаксического сахара включают различные краткие формы записи условных операций, циклов и других конструкций. Эти особенности улучшают читаемость и упрощают написание кода, но транслятор или компилятор сводит их к более простым базовым конструкциям перед выполнением.
- 5
- 6 Например, можно вызвать '`f1`' для вектора не записывая '`broadcast`' в явном виде
- 7 `"""`

`[1, 9, 16]`

- 1 `f1.([1, 3, 4])`

Задания

Задание 1

Реализуйте функцию `add_one`, которая принимает один аргумент, прибавляет к нему единицу и возвращает его.

```
1 md"""
2 # Задания
3
4 ## Задание 1
5 Реализуйте функцию `add_one`, которая принимает один аргумент, прибавляет к нему
6 единицу и возвращает его.
6 """
```

Задание 2

Примените `add_one` на матрицу из чисел в диапазоне от 1 до 30 при помощи точечной записи.

```
1 md"""
2 ## Задание 2
3 Примените `add_one` на матрицу из чисел в диапазоне от 1 до 30 при помощи точечной
4 записи.
4 """
```

Задание 3

Напишите функцию `compose(f, g)`, которая возвращает новую функцию, описывающую композицию двух функций `f` и `g`. Т.е. `(compose(f, g))(x)` должно быть эквивалентно `f(g(x))`.

```
1 md"""
2 ## Задание 3
3 Напишите функцию `compose(f, g)`, которая возвращает новую функцию, описывающую
4 композицию двух функций `f` и `g`. Т.е. `(compose(f, g))(x)` должно быть
5 эквивалентно `f(g(x))`.
5 """
```

Задание 4

Реализуйте функцию `filter_array`, которая принимает предикативную функцию и массив, а возвращает новый массив с элементами, для которых предикативная функция вернула `true`

```
# Пример работы
function is_odd(x)
    return x % 2 != 0
end

filtered = filter_array(is_odd, [1, 2, 3, 4, 5])
println(filtered) # Выведет [1, 3, 5]
```

```
1 md"""
2 ## Задание 4
3 Реализуйте функцию `filter_array`, которая принимает предикативную функцию и
4 массив, а возвращает новый массив с элементами, для которых предикативная функция
5 вернула `true`
6 """
7 # Пример работы
8 function is_odd(x)
9     return x % 2 != 0
10 end
11
12 filtered = filter_array(is_odd, [1, 2, 3, 4, 5])
13 println(filtered) # Выведет [1, 3, 5]
14 """
15 """
```

Задание 5

Реализуйте функцию `make_threshold_function(threshold)`, которая создает и возвращает функцию, которая проверяет, превышает ли число заданный порог. Если число больше или равно порогу, функция должна возвращать `true`, иначе — `false`

```
is_over_10 = make_threshold_function(10)
println(is_over_10(9)) # Выводит false
println(is_over_10(10)) # Выводит true
println(is_over_10(11)) # Выводит true
```

```
1 md"""
2 ## Задание 5
3 Реализуйте функцию `make_threshold_function(threshold)`, которая создает и
4 возвращает функцию, которая проверяет, превышает ли число заданный порог. Если
5 число больше или равно порогу, функция должна возвращать `true`, иначе – `false`
6 """
7 is_over_10 = make_threshold_function(10)
8 println(is_over_10(9)) # Выводит false
9 println(is_over_10(10)) # Выводит true
10 println(is_over_10(11)) # Выводит true
11 """
12 """
```

Задание 6

Создайте функцию `generate_math_function(op, y)`, которая принимает строку `op` (оператор: "+", "-", "*", "/") и число `y`, возвращая новую функцию, которая применяет данную операцию и число к аргументу.

```
subtract_five = generate_math_function("-", 5)
println(subtract_five(10)) # Выведет 5, так как 10 - 5 = 5

multiply_by_three = generate_math_function("*", 3)
println(multiply_by_three(6)) # Выведет 18, так как 6 * 3 = 18
```

```
1 md"""
2 ## Задание 6
3 Создайте функцию `generate_math_function(op, y)`, которая принимает строку `op` (оператор: "+", "-", "*", "/") и число `y`, возвращая новую функцию, которая применяет данную операцию и число к аргументу.
4 """
5 subtract_five = generate_math_function("-", 5)
6 println(subtract_five(10)) # Выведет 5, так как 10 - 5 = 5
7
8 multiply_by_three = generate_math_function("*", 3)
9 println(multiply_by_three(6)) # Выведет 18, так как 6 * 3 = 18
10 """
11 """
```

Задание 7

Напишите функцию `transform_strings`, которая принимает массив строк и функцию преобразования. Функция должна возвращать новый массив, где каждая строка будет изменена согласно переданной функции преобразования.

```
uppercase_transform = s -> uppercase(s)
result = transform_strings(["hello", "world"], uppercase_transform)
println(result) # Выведет ["HELLO", "WORLD"]
```

```
1 md"""
2 ## Задание 7
3 Напишите функцию `transform_strings`, которая принимает массив строк и функцию преобразования. Функция должна возвращать новый массив, где каждая строка будет изменена согласно переданной функции преобразования.
4 """
5 uppercase_transform = s -> uppercase(s)
6 result = transform_strings(["hello", "world"], uppercase_transform)
7 println(result) # Выведет ["HELLO", "WORLD"]
8 """
9 """
```

Задание 8

Создайте функцию `apply_to_arrays`, которая принимает два массива и функцию, применяющуюся к элементам обоих массивов (например, функцию суммирования). Верните результат в виде нового массива.

```
add_elements = (x, y) -> x + y
result = apply_to_arrays([1, 2, 3], [4, 5, 6], add_elements)
println(result) # Выведет [5, 7, 9]
```

```
1 md"""
2
3 ## Задание 8
4 Создайте функцию `apply_to_arrays`, которая принимает два массива и функцию,
5 применяющуюся к элементам обоих массивов (например, функцию суммирования). Верните
6 результат в виде нового массива.
7 """
8
9 """
10 """
```