

Федеральное государственное образовательное бюджетное учреждение
высшего профессионального образования
«ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ»
(Финансовый университет)

Департамент математики

Дисциплина «Программирование в среде R»

П.Б. Лукьянов

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ
ЛАБОРАТОРНОЙ РАБОТЫ № 4

Функции, написанные Пользователем

Для студентов, обучающихся по направлению подготовки
«Прикладная математика и информатика»
(программа подготовки бакалавра)

Москва 2021

Цель лабораторной работы – изучение особенностей использования функций языка R, правил создания собственных функций и способов их использования при написании программ.

Понятие функции

Функции в программировании используются очень широко: практически в любой программе не обойтись без обращения к различным функциям. Функции всегда выполняют что-то полезное, решают маленькие конкретные задачи: считывают файлы, устанавливают соединения, получают данные от Пользователя, выводят результаты расчетов на экран или в файл, рисуют графики, считают сложные математические выражения и т.д.

Программисту часто совершенно не нужно знать, что именно делается внутри вызываемых функций, достаточно помнить имя нужной функции, представлять, что она делает и знать способы передачи данных в функцию для получения результата.

Можно представить функцию в виде Черного Ящика. Черным Ящиком в науке называют некоторый процесс или явление, о котором нам ничего не известно, но мы точно знаем, что получим на ВЫХОДЕ из ящика, если на ВХОД ящика передадим некоторый набор значений (параметров) (см. рис. 1).

Функция очень похожа на Черный Ящик (рис. 2). Функция может выполнить некоторые действия, которыми мы управляем через входные параметры функции, и на этом работа функции завершится. Но часто результат работы функции нужно использовать дальше, в других частях программы. В этом случае функцию проектируют так, чтобы можно было забрать из нее полученный результат (рис. 2).

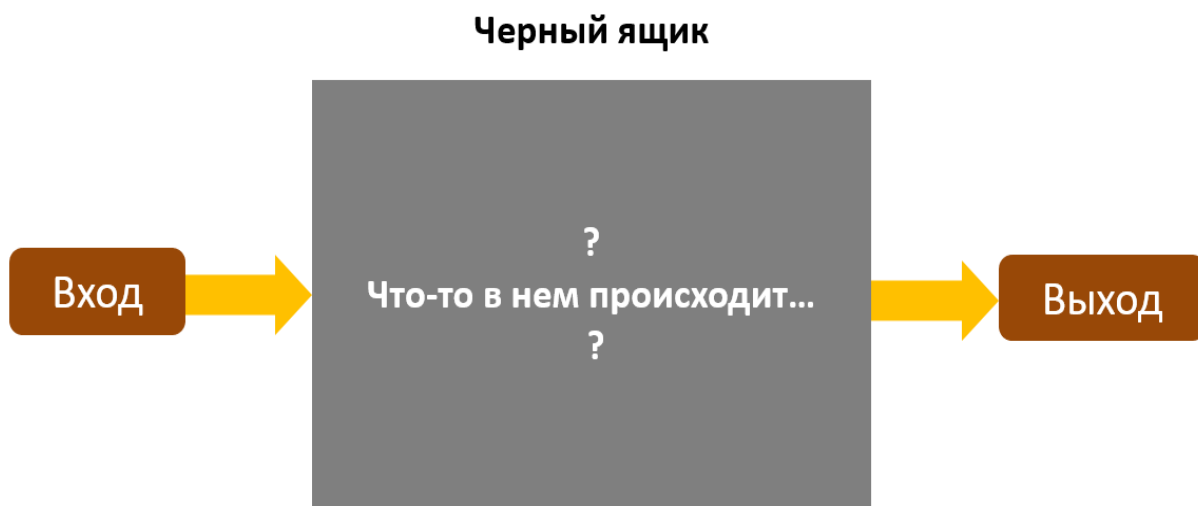


Рис. 1. Понятие черного ящика

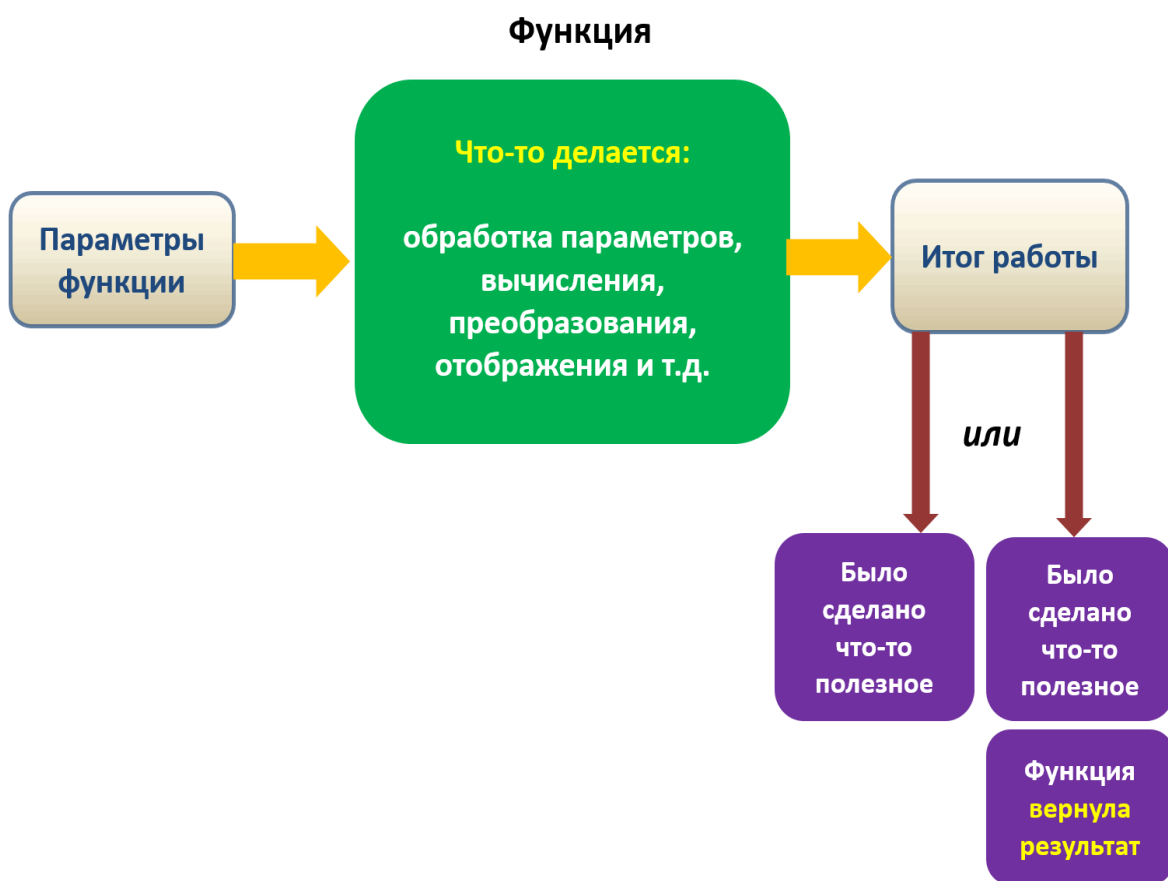


Рис. 2. Представление функции в виде черного ящика

Точнее говоря, сама функция после выполнения некоторых действий и получения результата этот результат возвращает в ту точку программы, откуда был выполнен вызов этой функции.

Использование функций

Программистами написано множество различных функций, чтобы с их помощью решать любую вычислительную задачу. Программист при написании программы постоянно вызывает те или иные функции для реализации своих целей.

В R большинство важных и часто используемых функций устанавливается вместе со средой R, ими сразу можно пользоваться и вызывать в своих программах. Кроме этого, огромное количество функций хранится в дополнительных пакетах, которые, в случае необходимости, загружаются из интернета и устанавливаются на компьютер Разработчика. Эти специальные функции существенно расширяют возможности языка.

Вместе с тем, при написании собственных программ у Программиста часто возникает необходимость написать **свои функции**, которые упростят использование программы, сделают код более наглядным и структурированным. Разработанные и отлаженные функции затем можно будет использовать и в других программах (рис. 3). Но каким бы способом функции ни появились в среде R, все они имеют общие свойства и правила использования, о которых необходимо знать.

Разберем формальную сторону вызова функций и задания параметров функции. В языке R функция отличается от переменной наличием круглых скобок: `w28` – переменная, `w28()` – функция. Круглые скобки – обязательный атрибут функции, они нужны для управления функцией, для передачи ей управляющих параметров. Другое название управляющих параметров – аргументы функции.



Рис. 3. Функции Пользователя расширяют возможности языка

Можно сказать, что круглые скобки – это двери, через которые внутрь функции попадают нужные значения: строки, числа, векторы, выражения и т.д. Функция считывает эти параметры и используя их, делает некоторые полезные действия.

Сколько аргументов может быть у функции? Сколько параметров можно ей передать? Здесь нет жестких правил, количество возможных параметров зависит от самой функции.

Многие функции можно вызывать без параметров:

- функция выхода из R `quit()` или `q()`
- вызов справки `help()`
- считывание ввода с клавиатуры `readline()`
- вывод системной даты и времени `date()`

Но эти же функции вызываются и с параметрами:

- `quit('yes')`, `q('no')`, `quit('default')`
- вызов справки по конкретной функции `help(q)`, `help(help)`
- считывание ввода с клавиатуры с пояснением `readline('Ваше имя? ')`

У некоторых функций параметров не предусмотрено (например `date()`), некоторые функции всегда должны вызываться с параметрами (`is.numeric(...)`, `plot(...)`, `round(...)`, `sqrt(...)`, `axis(...)` и др.).

Создание собственных функций

Функции, написанные Пользователем (программистом), дают ему широкие возможности и дополнительную гибкость: кусок кода, который делает что-то полезное, может быть использован повторно и многократно в разных вариациях, с разными наборами данных, в различных программах.

В случае необходимости что-то исправить в работе функции все исправления делаются только в одном месте – в коде самой функции, а результат этих изменений автоматически проявится везде, где эта функция будет вызвана.

Когда нужно создавать свою функцию? Существует несколько правил, когда это необходимо делать (см. рис. 4):

Когда нужна своя функция?

- Если часть кода в программе выполняется **более одного раза**
- Если код решает некоторую **конкретную** задачу
- Если в дальнейшем предполагается использовать код **повторно**
- Если количество строк кода **превышает размер экрана**. Если код не помещается на одном экране, его надо делить на части и из частей делать функции

Рис. 4. Причины создания своей функции

Последнее правило связано с удобством программирования при написании кода: весь код, с которым в данный момент работает программист, должен быть виден на экране полностью.

Большое количество строк кода и необходимость постоянно прокручивать файл вверх-вниз замедляют и усложняют работу. В этом случае часть кода или сворачивают (среда разработки делает этот код невидимым, чтобы он не занимал места на экране), или выносят в отдельную функцию.

Как правило, все описания функций хранятся в специальных файлах, так называемых библиотеках. Затем библиотека добавляется к программе, и мы можем пользоваться функциями, по мере надобности вызывая их из библиотеки для решения различных задач.

В зависимости от того, что делает функция, мы либо присваиваем какой-либо переменной результат ее работы, либо просто вызываем

функцию для выполнения определенных действий: производить вычисления, рисовать график, считывать данные из файла, сохранять расчетные значения в файл и т.д.

Как написать свою функцию? См. рис. 5.

Как написать свою функцию?

- Как правило, функция объявляется в отдельном скрипте
- Программист пишет функцию (создает определение функции):
 - придумывает **имя**, перечисляет **параметры** (аргументы) функции
 - задает **правила обработки** параметров функции
 - определяет **результат** выполнения функции
- Сначала идет **объявление** функции, затем ее **использование** (вызов функции)

```
# объявление функции
someFunction <- function (argument) {
    statement
}

# использование функции в тексте программы
w <- someFunction(argument)
```

- Программист может делать вызов функции в разных программах

Рис. 5. Правила создания своей функции

Таким образом, порядок шагов для написания функции следующий:

1. Нужно создать новый скрипт, дать ему имя, например userFuncs.R

2. В этом скрипте написать конструкцию, задающую определение функции, представленную на рис. 5.
3. Придумать имя функции, определить список необходимых параметров. Список параметров указывается после имени функции в круглых скобках. В частном случае параметров у функции может и не быть, но круглые скобки после имени функции обязательны. Именно круглые скобки говорят среде R, что мы вызываем функцию, а не обращаемся к переменной.
4. Написать код тела функции, в котором обрабатываются переданные параметры и выполняются различные действия.
5. Создать или открыть скрипт, в котором будет использоваться созданная функция.
6. В первых строках этого скрипта написать оператор `source("userFuncs.R")` для подключения библиотеки с описанием функции. В случае, если скрипт `userFuncs.R` располагается не в текущей папке с создаваемой программой, при вызове `source('.....')` нужно указывать абсолютный или относительный путь к `userFuncs.R`, например
`source("c:\\RLessons\\userFuncs.R")` или
`source("c:/RLessons/userFuncs.R")`
7. В соответствии с логикой работы разрабатываемой программы вызвать функцию, передать в нее параметры, получить результат ее выполнения.

Вначале, для освоения навыков написания функции, определение функции и код с ее вызовом можно размещать в одном файле. Разберем на примере, как создать функцию, возводящую x в степень y и выводящую результат на экран.

Создадим функцию **pow()** (рис. 6)

```
pow <- function(x, y){  
  result <- x^y  
  print(paste(x, "в степени", y, 'равно', result))  
}
```

Рис. 6 Пример функции Пользователя

Описание того, какие возможны варианты вызова функции `pow()`, представлены на рис. 7.

При вызове функции **фактические аргументы (2, 8)** сопоставляются с **формальными аргументами (x, y)** в порядке следования аргументов

`x = 2; y = 8`

Можно вызвать функцию, используя **именованные аргументы**:

`pow(8, 2)`

`pow(x = 8, y = 2)`

`pow(y = 2, x = 8)` # изменили порядок следования аргументов

`pow(x = 8, 2)`

`pow(2, x = 8)` # изменили порядок следования аргументов

При вызове функции таким образом порядок фактических аргументов не имеет значения.

Сначала сопоставляются все именованные аргументы, а затем оставшиеся неименованные аргументы сопоставляются в порядке следования

Рис. 7 Различные варианты вызова `pow()`

Значения параметров по умолчанию

Вызов функции без задания значений параметрам называется вызовом функции «по умолчанию». При вызове функции без параметров функция считает, что параметры все равно переданы, но они установлены в какое-то конкретное значение «по умолчанию», и исходя из этого значения, функция работает определенным образом.

В общем случае у функции может быть множество параметров, и у каждого параметра могут быть свои значения по умолчанию. Программист при вызове функции, как правило, задает значения лишь нескольким параметрам, оставляя значения других параметров по умолчанию.

Как задать значение по умолчанию для параметра функции? В описании функции после указания формального имени параметра задается его начальное значение, т.е. выполняется инициализация параметра. В этом случае значение этого параметра при вызове функции можно не задавать (рис. 8).

```
pow <- function(x, y=2){  
  result <- x^y  
  print(paste(x, "в степени", y, 'равно', result))  
}
```

Рис. 8 Задание значения по умолчанию для одного из параметров

Теперь вызов функции может быть таким:

```
pow(3)
```

Результат равняется 9.

Если же вызвать функцию как обычно, с двумя параметрами, то для второго значения будет использоваться фактическое, переданное значение:

```
pow(3, 3)
```

Результат равняется 27. Проверьте это.

Использование ключевого слова **return**

Ключевое слово **return** используется в описании функции, чтобы явным образом задать результат, который возвратит функция при ее вызове. При достижении **return** работа функции завершается и происходит возврат в основную программу, в точку вызова функции. Если в основной программе результат вызова функции присваивался переменной, то в переменной будет значение, указанное в круглых скобках после **return** в теле функции:

```
return(result)    # функция возвратит значение переменной result
```

Если **return** не использовать, то результатом работы функции будет результат выполнения последнего оператора в теле функции (см. пример на рис. 9).

```
nSignDefiner <- function(x) {  
  if (x > 0) {  
    result <- "Положительное число"  
  }  
  else if (x < 0) {  
    result <- "Отрицательное число"  
  }  
  else {  
    result <- "А это ноль"  
  }  
  result  
}
```

Рис. 9 Функция без **return()** выполняется до последнего оператора

В описании функции конструкция **return(возвращаемое выражение)** может встречаться несколько раз, но выполнение функции прервется при достижении первого оператора **return** (см. рис. 10).

```
nSignDefiner2 <- function(x) {
  if (x > 0) {
    return("Положительное число")
  }
  else if (x < 0) {
    return("Отрицательное число")
  }
  else {
    return("А это ноль")
  }
}
```

Рис. 10 Функция будет выполняться до первого return()

Использование **return** сокращает время работы функции, и логика ее работы становится более понятной.

Передача значений через параметры при вызове функции

Рассмотрим пример. Самостоятельно напишите функцию `sum3Numb()`, возвращающую сумму трех чисел. У функции должно быть три числовых параметра: `x1`, `x2`, `x3`. Каждому параметру задайте свое значение по умолчанию: `x1 = 10`, `x2 = 20`, `x3 = 30`. Это означает, что теперь функцию `sum3Numb()` можно вызывать без параметров, и она должна вернуть значение 60. Проверьте это.

Пусть программист вызывает `sum3Numb()` и хочет переопределить второй параметр, передав ему значение 27. Если он запишет вызов функции в виде `sum3Numb(27)`, то функция не догадается, что речь идет о ее втором параметре.

Существует несколько способов решения этого вопроса. Первый способ – указывать место параметра в списке всех параметров функции:

```
sum3Numb( , 27)
```

Смысл (, 27) следующий: первый параметр не меняем, оставляем пустое место, а для второго параметра передаем 27. Третий параметр и последующие, если бы они были, не трогаем.

Для передачи 27 в третий параметр вызов функции будет таким:

```
sum3Numb( , , 27)
```

Такой способ не всегда удобен – чтобы им пользоваться, нужно точно знать порядковый номер требуемого параметра в списке аргументов функции. У сложной функции счет аргументов может идти на десятки, и в случае необходимости задать 17-й параметр легко ошибиться, переопределив вместо него 16-й или 18-й параметр.

Поэтому более надежный и простой способ заключается в **использовании формальных имен** аргументов функции и присваивании значений этим именам при вызове функции.

У любой функции каждый аргумент имеет формальное имя (в нашем случае это x1, x2, x3), и если оно известно, то можно больше не думать о порядковом номере параметра в списке аргументов. При вызове функции мы просто присваиваем имени аргумента нужное значение. Вот как это работает:

Напишите функцию row(x,y) для возведения числа x в степень y. Как должна работать эта функция:

```
w1<- row(2,5)   # результат равен 32
```

```
w2<- row(5,2)   # результат равен 25
```

В качестве первого параметра передается x, в качестве второго – y.

Теперь вызовем функцию row() с явным заданием параметров, используя имена параметров:

```
w3<- row(x=2, y=5)   # результат равен 32
```

```
w4<- row(y=5, x=2)   # результат равен 32
```

Если в первом случае, без использования имен аргументов, порядок следования параметров был важен, то во втором случае порядок не важен, мы явно определяем нужный параметр.

Если у функции есть значения по умолчанию, то мы используем такую форму вызова:

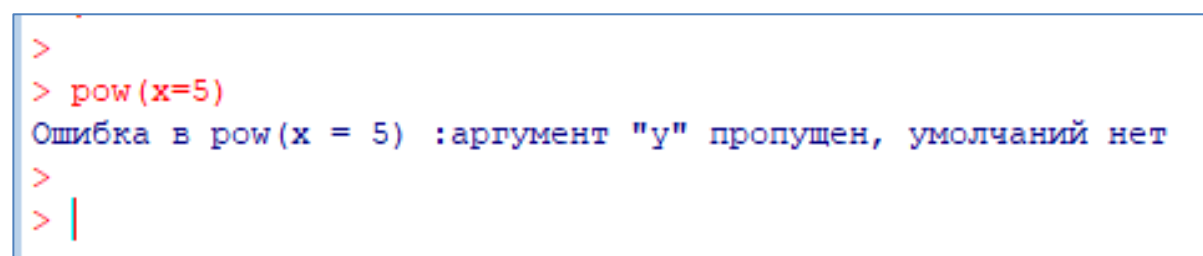
```
q1<- row(x=5) # значение y оставляем по умолчанию
```

```
q2<- row(y=5) # значение x оставляем по умолчанию
```

Если же у аргументов функции значений по умолчанию нет, мы получим ошибку (см. рис. 11).

Обратите внимание, что хотя в языке R поддерживается несколько вариантов операторов присваивания, но при вызове функции с заданием значения формальному аргументу нужно использовать традиционный знак равенства, в этом случае мы явно определяем нужный нам аргумент числовым значением.

Если же при задании параметра у использовать вместо равенства присвоение вида `y<-5`, это приведет к тому, что в основной программе будет создана новая переменная `y`, вызов функции будет выполнен как `row(5)`, и значение 5 будет присвоено первому параметру функции `row()`, т.е. фактически выполнится `row(x=5)`. Проверьте это.



```
>  
> row(x=5)  
Ошибка в row(x = 5) :аргумент "y" пропущен, умолчаний нет  
>  
> |
```

Рис. 11. Ошибка при вызове функции при отсутствии требуемого параметра

Возникает следующий вопрос – как узнать имена аргументов функции? Вспомним, какие функции могут использоваться в программе:

1. функции, поставляемые вместе со средой R и включенные в дистрибутив R
2. функции, которые содержатся в пакетах, устанавливаемых дополнительно
3. функции, которые пишет сам Программист

В первом случае нужно вызвать подсказку, набрав ?имяФункции или help(имяФункции); во втором случае нужно смотреть документацию, которая поставляется с пакетом; в третьем случае программист сам задает имена аргументов своим функциям.

Еще раз про возвращаемое значение функции

Все многообразие функций языка можно разделить на две большие категории:

- функции, которые возвращают значение (в результате каких-либо расчетов появляется результат)
- функции, которые делают что-то полезное (выводят сообщение, например), но никакой результат не формируют. В этом случае результатом функции будет NULL. Проверьте это.

В некоторых языках программирования функции первого типа так и называют – функции, а функции второго типа называют процедурами.

Если результат вызова функции в дальнейшем планируется использовать, то этот результат для его сохранения присваивают переменной:

```
res <- pow(2,5) # в переменной res теперь хранится значение 32
res.int <- round(res / pi, 0) # округлили до целого
res.dec4 <- round(32 / pi, 4) # округлили до 4 знаков после запятой
res.dec4 <- round(digits = 4, 32/ pi) # другая форма записи
```


Теперь рассмотрим использование функций, не возвращающих никакого значения. Напишите функцию `print6lines()`, которая последовательно выводит на экран шесть строчек каких-либо сообщений. Если вызвать эту функцию, то она выведет на экран эти строчки. Но что будет сохранено в переменную, если ей присвоить вызов этой функции?

```
mes <- print6lines()
```

Если затем в программе вызвать переменную `mes`, то скорее всего будет выведено последнее сообщение из шести (см. рис. 12). В переменную `mes` будет записан результат команды, которая в функции `print6lines()` была последней.

```
>
> mes <- print6lines()
[1] "Это первая строчка"
[1] "Это вторая строчка"
[1] "А я третья!"
[1] "Я уже четвертая"
[1] "Я пятая"
[1] "А я шестая по счету, последняя!"
> mes
[1] "А я шестая по счету, последняя!"
>
```

Рис. 12. Пример работы с функцией, не возвращающей значение

Но значение переменной `mes` может быть и другим (см. рис.13). Как понять результат, представленный на рис. 13? Почему значение переменной `mes` стало равно числу?

Видимо, после вывода строк с сообщениями в функции `print6lines()` выполнялись еще какие-либо действия, и результатом выполнения последней команды функции стало значение `-3.87905`.

Выше были рассмотрены случаи, когда функция или возвращает какое-либо значение, или не возвращает никакого значения. Но что делать,

если в результате расчетов получен ряд значений, и все эти значения нужно сохранить? В этом случае возвращаемым значением функции может быть вектор, матрица, таблица или список.

```
>  
> mes <- print6lines()  
[1] "Это первая строка"  
[1] "Это вторая строка"  
[1] "А я третья!"  
[1] "Я уже четвертая"  
[1] "Я пятая"  
[1] "А я шестая по счету, последняя!"  
> mes  
[1] -3.87905  
>
```

Рис. 13. Результат функции, не возвращающей значение, неочевиден

Использование параметров для управления поведением функции

Еще один вариант использования параметров функции – управление выводом и представлением результатов. Рассмотрим нашу функцию `row(x,y)`. Сейчас эта функция выводит строку с результатом. При выполнении расчетов пользоваться такой функцией неудобно, так как нам от вызванной функции нужно число – значение x в степени y , чтобы затем использовать это число в дальнейших расчетах.

Вместе с тем, например, для отладки, может понадобиться вывод информационного сообщения с указанием переданных параметров, промежуточных расчетов и т.д.

Можно ли эти два сценария работы реализовать в одной функции? Да, можно. Для этого к списку параметров добавим еще один аргумент `isDebug`, принимающий логические значения `TRUE` или `FALSE`.

Если передано `FALSE`, функция будет возвращать число, в противном случае функция выведет подробную информацию о своей

работе. Вариант кода приведен на рис. 14. Обратите внимание, что для параметра `isDebug` задано значение по умолчанию `FALSE`. Смысл этого в том, чтобы в расчетах при вызове функции не загромождать код основной программы передачей параметра с одним и тем же значением `FALSE`.

В больших программах режим отладки может понадобиться в разных частях, поэтому логично сделать две глобальные константы, задающие тот или иной режим выполнения функций и частей кода.

В нашу тестирующую программу в самом начале нужно добавить секцию констант и определить следующие константы:

```
DEBUG_ON <- TRUE
```

```
DEBUG_OFF <- FALSE
```

При создании констант программисты придерживаются следующего соглашения: константы пишутся заглавными буквами, слова в константах отделяются друг от друга нижним подчеркиванием.

Теперь наша программа написана по всем правилам (см. рис. 15).

```
#####  
# ОПИСАНИЯ ФУНКЦИЙ  
  
pow2 <- function(x, y, isDebug = FALSE) {  
  if (isDebug) {  
    return(print(paste0('pow2: x=', x, ', y=', y, ', расчет=', x ^ y)))  
  } else {  
    return(x ^ y)  
  }  
}  
  
#####  
# ОСНОВНАЯ ПРОГРАММА  
{  
  # запускаем функцию в режиме отладки:  
  z <- pow2(4, 6, TRUE)  
  
  # запускаем функцию для выполнения каких-либо расчетов:  
  q <- 29 * pow2(3, 7) / sqrt(12)  
  q  
}
```

Рис. 14. Работа функции в режимах отладки и вычислений

```
#####
# КОНСТАНТЫ
DEBUG_ON <- TRUE
DEBUG_OFF <- FALSE

#####
# ОПИСАНИЯ ФУНКЦИЙ

pow2 <- function(x, y, isDebug = DEBUG_OFF) {
  if (isDebug) {
    return(print(paste0('pow2: x=', x, ', y=', y, ', расчет=', x ^ y)))
  } else{
    return(x ^ y)
  }
}

#####
# ОСНОВНАЯ ПРОГРАММА
{
  # запускаем функцию в режиме отладки:
  z <- pow2(4, 6, DEBUG_ON)

  # запускаем функцию для выполнения каких-либо расчетов:
  q <- 29 * pow2(3, 7) / sqrt(12)
  q
}
```

Рис. 15. Использование констант для удобства управления программой

Рассмотрим негласные правила создания имен функций. Имя функции должно начинаться со строчной буквы и быть смысловым, состоять из нескольких английских слов, отражающих работу функции; первое слово – глагол, описывающий выполняемое действие, затем идет одно или несколько существительных, отражающих то, над чем выполняется действие. Например, calcRent, checkInputVal, writeTabToServer и т.д.

В языке R часто в качестве разделителя в именах функций и переменных используется точка: `calc.rent()`, `check.input.val()`, `write.tab.to.server()`. В этом случае заглавные буквы не ставят.

Возможные проблемы

Если в результате запуска скрипта с программой выводится сообщение о том, что среда R не может найти файл `userFuncs.R`, значит, нужно изменить текущую рабочую директорию среды R на ту папку, в которой сейчас сохраняются файлы R.

Если работаем в консоли R, нужно выполнить команду перехода в папку со своими файлами

```
setwd("путь_к_своей_папке")
```

где функция `setwd()` означает "задать рабочую директорию", а "путь_к_своей_папке" может выглядеть, например, следующим образом: `"c:/Rlab/lab4"`.

Если работаем в RStudio, нужно изменить настройки: Tools – Global Options – General – Default working directory и задать свою папку.

Если в создаваемой функции при выводе сообщений использовались русские буквы, а при вызове этой функции вместо осмысленных сообщений получаем крючки и непонятные символы, значит, имеются проблемы с совместимостью кодировки по умолчанию среды R, или RStudio, и текущей кодировки используемой операционной системы.

Один из путей решения проблемы – принудительно задать нужную кодировку, для этого используют вызов функции `Sys.setlocale()`:

```
Sys.setlocale(category = "LC_ALL", locale = "Russian") # для Windows
```

```
Sys.setlocale("LC_CTYPE", "en_RU.UTF-8") # для других ОС
```

Полезной может оказаться функция `sessionInfo()`, которая показывает текущие значения используемых национальных настроек. Если решить проблемы с кодировкой не получается, рекомендуется использовать ОДИН

ОБЩИЙ ФАЙЛ (скрипт) для размещения в нем самих функций и кода программы, использующего эти функции.

Самостоятельные задания

Задание 1.

Аналогично примеру, разобранному выше, создать функцию, в которой переданный параметр x возводится в степень y , а затем делится на параметр z . Результат выводится на экран. В случае деления на ноль функция должна выводить сообщение об ошибке в параметре z .

Также функция должна проверять тип переданных параметров. Если это не целые или действительные числа, выдавать соответствующее предупреждение.

Задание 2.

Создать функцию, которая по переданному параметру N возвращает наименование дня недели на русском языке. Если N меньше 1, функция возвращает пробел. Если число действительное, преобразовывать число к целому.

Предусмотреть в функции режим отладки. В режиме отладки выводить полученные функцией параметры и день недели.

Задание 3.

Развитие задания 2. Функция должна уметь работать с языком вывода. Если передан строковый параметр «Eng», или «eng», или «English», или «english», или «англ», или «Англ», или «анг» выводить дни недели на английском языке.

Если передан строковый параметр «rus», или «ru», или «RU», или «рус», или «ру», выводить дни недели на русском языке.

Если параметр не задан или не распознан, текст выводить на русском языке.

Задание 4.

Развитие задания 3. Функция должна уметь выводить название дня недели в двух формах: полной или сокращенной в зависимости от переданного логического параметра. Если параметр не задан, выводить полное название дня недели.

Задание 5.

Развитие задания 4. В качестве параметра N передавать вектор с набором значений произвольной длины.