ST2195 Programming for Data Science
Block 10 Introduction to Software Development

VLE Resources

There are extra resources for this Block on the VLE – including quizzes and videos please make sure you review them as part of your learning. You can find them here: https://emfss.elearning.london.ac.uk/course/view.php?id=382#section-10

Software Development Phases

It may sound surprising that the majority of time and effort when developing software for solving a problem in business and research environments is not spent in writing code! In fact, principled software development involves multiple phases, most of which require little or no coding! Software development involves at least elements of each of the following phases:

- Requirements analysis and planning
- Software design
- Implementation
- Testing
- Integration and deployment
- Maintenance

The majority of the code is written during the implementation phase.

How the above phases are combined and in what order, and how much time or effort is invested in each phase is guided by the software development life cycle (SDLC) methodology that each software development team or organization chooses to adopt.

Software Development Life Cycle Methodologies

There is a range of SLDC methodologies (or models), each with its own advantages and disadvantages depending on the project at hand. The most prominent SLDCs at the time of writing are:

- *Waterfall SDLC model*: The phases are organized in a rigid way in the order given above with strong investment of time and effort in each phase and with no option to revisit past phases. Most software companies nowadays require more flexibility than what the waterfall model provides, but it still remains a strong methodology for mission-critical projects, where mistakes in the solution can result in extreme costs (e.g. piloting software, software used in nuclear power stations, etc.)

- *Iterative SDLC model*: The key idea of this model is to develop a system through repeated iterations or the waterfall model, where at each iteration (also known as a "mini waterfall") a sub-problem is solved. Despite the fact that the phases are again organized in a rigid way, developers can take advantage of what was learned from the iterations of earlier parts.
- *Agile SDLC model*: The agile model is in a sense the opposite of the waterfall model. Instead of organizing the phases in a rigid way, the agile model makes all phases an ongoing process that requires the involvement of developers, management and customers. Work is typically organized in two- to four-week segments (known as "sprints"), in which the responsible teams tackle the major needs of their customers and perform testing, integration and deployment on the go. The agile model is popular in start-ups where speed and flexibility (e.g. because of changes in requirements) are essential.
- *V-shaped SDLC model*: This is an evolution of the waterfall model, where each development phase before implementation is associated with a testing phase (e.g. requirement analysis and planning—acceptance testing, software design—software testing). The next phase starts only after there is a testing activity corresponding to it.
- *DevOps SDLC models*: One of the most recent SDLC models, where *dev*elopment and *op*erational teams are merged together into a single team. Software developers and engineers work across the entire SDLC from development to testing, to deployment and operations, including interaction with customers.
- *Spiral SDLC model*: The spiral model is a blend of iterative and waterfall approaches, and allows teams to adopt multiple SDLC models based on the risk patterns of the project.

The SDLC tutorial at tutorialspoint.com provides an accessible overview of SDLC methodologies, and of their advantages and disadvantages. Useful Links and Resources below provide links with more details on each SDLC.

Useful Links and Resources

- tutorialspoint pages for the waterfall model: Details on the waterfall model
- tutorialspoint pages for the iterative model
- tutorialspoint pages for the agile model
- tutorialspoint pages for the V-shaped model: Details on a version of the V-shaped SDLC model
- Wikipedia's DevOps page
- tutorialspoint pages for the spiral model: Details on the spiral SDLC model

Developing R packages

Packages

Python and R packages play a key role in data science projects, not only because they are routinely being used as dependencies or tools during at least some of the software development phases but also because they are often the key outputs and deliverables of a project or part of those.

In fact, one of the reasons that Python and R are so pervasive in data science is because they benefit from diverse and active ecosystems of packages that, as we have seen in this course, provide ready methods for read and write operations, data wrangling and visualization, and for developing machine learning pipelines. As we will also see next week, there are even dedicated packages for testing software.

A package bundles together code, data, documentation, and potentially tests. This makes it easy to share with others, either by submitting to one of the official repositories (e.g. CRAN for R and PyPi for Python) or by hosting it in a repository hosting service like GitHub.

Even if you do not want to share your code, organizing it in a package always pays off. Python and R packages follow strict rules on how code is organized, documented and tested, and as a result, packaged code is easier to return to than plain text files, even after several years.

In the following sections you will learn the basic structure of R and Python packages.

Developing R Packages

Package Skeleton

Suppose that you wrote a function `circle()` that takes as input the radius of a circle and returns a list of class `"circle"` with the radius, and a function `area.circle()` that takes as input objects of class `"circle"` and calculates the area of the circle

```
circle <- function(radius) {
    shape <- list(radius = radius)
    class(shape) <- "circle"
    shape
}
```

```
#' @export
area.circle <- function(object) {
    pi * object$radius^2
}
```

You may notice the commented-out line `#' @export` before `area.circle()`. Ignore it for now; this is special terminology, whose importance will become apparent later in this page.

You immediately realize that there is potential for an R package that provides functions and methods to compute the area of circles and other 2D shapes, called **areacalc**. How can you go about it?

The first step in writing an R package is to set up up the basic *skeleton* for the package directory. The conventions that should be used for a valid R package are detailed in Section 1.1 of the "Writing R extensions" resource. While you can always go ahead and create the required directories and files manually, many of those tasks are nowadays abstracted away through high-level, specialized utilities such as the ones provided by the **usethis** R package. Assuming that you have already installed **usethis**, the following code chunk sets up the skeleton for **areacalc** under the directory `"~/Repositories/"` (feel free to replace this with any *existing* directory in your system):

```
library("usethis")
create_package("~/Repositories/areacalc/")
✔ Creating '~/Repositories/areacalc/'
✔ Setting active project to '~/Repositories/areacalc'
✔ Creating 'R/'
✔ Writing 'DESCRIPTION'
Package: areacalc
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R (parsed):
    * First Last <first.last@example.com> [aut, cre] (YOUR-ORCID-ID)
Description: What the package does (one paragraph).
License: `use_mit_license()`, `use_gpl3_license()` or friends to
    pick a license
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.1
✔ Writing 'NAMESPACE'
✔ Changing working directory to '~/Repositories/areacalc/'
```

The output above details the steps that `create_package()` took while setting up the package skeleton for the **areacalc** R package. Now, the directory `~/Repositories/areacalc` looks like (using the `dir_tree()` function of **fs** R package)

```
library("fs")
dir_tree("~/Repositories/areacalc")
~/Repositories/areacalc
├── DESCRIPTION
├── NAMESPACE
└── R
```

This is the most basic structure for an R package. There is `DESCRIPTION` file, whose contents where automatically generated by `create_package()` earlier (see the output from `create_packages()` above), a `NAMESPACE` file which defines the package name space, and a directory called `R`, which will host the R code we will put in the package.

Adding New Functionality

A convenient way to add new functionality in the R package is by using **usethis**’s `use_r()` function. For example, the following code chunk will create a empty script called `cicle.R` under `~/Repositories/areacalc/R/` and open it for editing

```
use_r("circle")
✓ Setting active project to '~/Repositories/areacalc'
● Modify 'R/circle.R'
● Call `use_test()` to create a matching test file
```

Go ahead and copy and paste in that script the code chunk where the functions `circle()` and `area.circle()` are defined above and hit save. Feel free to ignore the last piece of advice in the output about calling `use_test()` for now; we will come back to this in the following week.

Congratulations! You have just created your first bare-bones R package! You can use the `load_all()` function of the **devtools** to load your new functions

```
library("devtools")
Loading required package: usethis
load_all("~/Repositories/areacalc")
Loading areacalc
ci <- circle(1)
area.circle(ci)
[1] 3.141593
```

Now, as an exercise, include a new script called `square.R` in **areacalc** with the following two functions to compute the area of a square

```
square <- function(side) {
    shape <- list(side = side)
    class(shape) <- "square"
    shape
}

#' @export
area.square <- function(object) {
    object$side^2
}
```

areacalc’s directory structure should look like

```
dir_tree("~/Repositories/areacalc")
~/Repositories/areacalc
├── DESCRIPTION
├── LICENSE
├── LICENSE.md
├── NAMESPACE
├── R
│   ├── area.R
│   ├── circle.R
│   ├── circle.R~
│   └── square.R
└── man
    ├── area.circle.Rd
    └── circle.Rd
```

and you should be able to do

```
load_all("~/Repositories/areacalc")
Loading areacalc
sq <- square(3)
area.square(sq)
[1] 9
```

Name Spaces and Methods

Packages have their own name spaces and, as such, are great places for object-oriented programming. Name spaces are a fairly advanced topic for which you can find more details in the section about name spaces in the “Advanced R” book; loosely, a namespace provides a context for looking up the value of an object associated with a name. As such they are vital for *encapsulating* the package and making it self-contained, ensuring that that other packages do not interfere with your code.

Back to **areacalc**, using the S3 OOP system we can easily implement a *generic method* called `area()` that will calculate the area of a shape, depending on whether the object we pass to the generic method is of class `"circle"` (e.g. as constructed by the `circle()` function) or `"square"` (e.g. as constructed by the `square()` function).

The names we used for the functions `area.circle()` and `area.square()` in **areacalc** were far from accidental. In fact, the particular naming convention `generic_method.class()` we used is S3-specific and makes `area.circle()` and `area.square()` *methods* ready to be used by a *generic method* `area()`!

In order to implement the generic method, we first create a new script under `~/Repositories/areacalc/R/` (I called it `area.R`) with the following very simple generic function specification

```
area <- function (object, ...) {
   UseMethod("area")
}
```

Then, we can call the `document()` function from the **devtools** R package to update the package’s name space. `document()` uses the methods from the **roxygen2** package to populate the NAMESPACE file.

```
document("~/Repositories/areacalc")
Updating areacalc documentation
Loading areacalc
Writing NAMESPACE
Writing NAMESPACE
```

The `#' @export` statements before the definitions of `area.circle()` and `area.square` told **roxygen2** to generate the table of exported methods. Your `NAMESPACE` file now looks like

```
# Generated by roxygen2: do not edit by hand

S3method(area,circle)
S3method(area,square)
```

in essence telling R that **areacalc** provides two S3 methods for the generic `area()` for objects of class `"circle"` and `"square"`.

Now we can do

```
ci <- circle(1)
sq <- square(1)
area(ci)
area(sq)
```

Notice here that the generic method `area()` automatically decides which specific method to dispatch (`area.circle()` or `area.square()`) depending on the class of the object we pass to it (`"circle"` or `"square"`, respectively). If you want to add support for a new shape, say a rectangle, you just need to write the `rectangle()`, and `area.rectangle()` functions, add `#' @export` before the definition of the `area.rectangle()`, and rerun `document()`.

DESCRIPTION File

The DESCRIPTION file is where we specify the package title, version, authors, licence, and much other information, including other packages that the package depends on. Go ahead and edit **areacalc**'s description file to add an informative title and your details. In my copy, I have edited the title and person specification, set the version to `0.0.1`, and used the `use_mit_license("John Doe")` function to populate the License attribute with the details of the MIT licence. After all this, my DESCRIPTION file looks like

```
Package: areacalc
Title: Methods to Define and Compute the Area of Various Shapes
Version: 0.0.1x
Authors@R:
    person(given = "John",
           family = "Doe",
           role = c("aut", "cre"),
           email = "john.doe@nowhere.com")
Description: Provides methods to define various shapes in terms of their
defining characteristics and compute their area.
License: MIT + file LICENSE
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.1
```

areacalc's directory structure is now

```
dir_tree("~/Repositories/areacalc")
~/Repositories/areacalc
├── DESCRIPTION
├── LICENSE
├── LICENSE.md
├── NAMESPACE
├── R
│   ├── area.R
│   ├── circle.R
│   ├── circle.R~
│   └── square.R
└── man
    ├── area.circle.Rd
    └── circle.Rd
```

Building Packages

If we want to share the package or submit it in a repository we can do

```
build("~/Repositories/areacalc")
✔  checking for file '~/Repositories/areacalc/DESCRIPTION'
─  preparing 'areacalc':
✔  checking DESCRIPTION meta-information
─  checking for LF line-endings in source and make files and shell scripts
─  checking for empty or unneeded directories
   Removed empty directory 'areacalc/man'
─  building 'areacalc_0.0.1.tar.gz'

[1] "~/Repositories/areacalc_0.0.1.tar.gz"
```

As the output suggests, the package directory structure is now in the compressed file `areacalc_0.0.1.tar.gz`, which can be shared as is for installation and use by other users.

The `use_github()` function of the **usethis** R package allows you to upload the package in GitHub. This can also be done "less manually" by using the GUIs that RStudio provides.

Installing

If you want to install **areacalc** version `0.01` and have it ready for use in future R sessions you simply do `install("~/Repositories/areacalc")` if you want to install directly from the package directory structure, or `install.packages("~/Repositories/areacalc_0.0.1.tar.gz")` if you want to install from the compressed file you've built in earlier.

Publishing

The easiest way to release an R package is using the `release()` from the **devtools** R package, which will guide you through the steps required for release and make recommendations about what you are expected to have done prior to release; see section on releasing R packages in the "R packages" book. Of course, **areacalc** is not ready yet for release. More work is needed to documenting the package, and it is also useful to incorporate some tests to make sure that the methods we developed work as expected.

Useful Links and Resources

- *R packages* book by Hadley Wickham: An authoritative account about how to build and publish R packages.
- Writing R extensions resource: The definitive guide for creating R packages and writing R documentation.

4. R package primer by Karl Broman: A minimal tutorial about writing R packages.

Modules and Packages in Python

Modularization

Modular programming is a design technique to separate the functionality of a program into some smaller independent parts for some specific tasks. As each module is for some specific tasks, it is much easier to navigate and locate the code. Different developers can also work on different parts of the program independently. We have already seen one tool to promote code modularization: functions. With the use of functions, we can break down a complex problem into some smaller sub-problems. The use of functions also facilitates code reusing. In this page, we introduce two other tools for code modularization: *modules* and *packages*.

Modules

In Python, a *module* is a file with the suffix `.py` that contains Python code. The name of the module is the same of the name of file. For example, the module `circle` is created by writing relevant code in the `circle.py` file.

The content of the `circle.py` file is as follows:

```
'''
This module contains functions for circle calculation
'''

pi = 3.14159
def area(radius):
    '''
    return area of a circle given the radius. Radius is assumed to be a
non-negative number
    '''
    return pi *(radius**2)


def circumference(radius):
    '''
    return circumference of a circle given the radius. Radius is assumed to
be a non-negative number
    '''
    return 2 * pi * radius
```

Using modules is very similar to using packages, as we have done so far. We first import the module by `import module`. For example, to import the module `circle` we write:

```
import circle
```

By using the function `help()`, we can see how to use the module:

```
>> help(circle)
Help on module circle:

NAME
    circle - This module contains functions for circle calculation

FUNCTIONS
    area(radius)
        return area of a circle given the radius. Radius is assumed to be a
non-negative number

    circumference(radius)
        return circumference of a circle given the radius. Radius is
assumed to be a non-negative number

DATA
    pi = 3.14159
```

The docstring of this module tells us that the module aims to do circle-related calculation. The `help()` function also tells us that the module has two functions for calculating the area and circumference of a circle given the radius, and an object `pi`. While the `help()` function tells us how to *use* the module and its functions, it does not tell us how the module and its functions are *implemented*. This is *abstraction* - we hide the unnecessary details from the user.

To get the attributes (which includes functions and other objects) in the module, we use `module.attribute`. For functions, we call `module.func()`. For example, if we want to use the function `area()` from the module `circle`, we need to call the function as follows:

```
circle.area(3)
28.26
```

Similarly, if you want to get the retrieve the objects from the module, you need to write `module.obj`. For example, if we want to retrieve `pi` from the module `circle`, we need to write:

```
circle.pi
3.14
```

We will get an error if we call the functions or use other objects directly, as functions and other objects *within* the module scope.

If you need to use the functions a lot, calling the functions in the module by `module.func()` may not convenient. Instead you can import the function or other objects from the module by `from module import function`. This enables you to use the them directly:

```
from circle import area, pi
print(area(3))
28.26
print(pi)
3.14
```

Packages

Packages are a collection of modules. We have been using packages like **NumPy** and **pandas** in this course. In each package, there is at least one `__init__.py` file. The `__init__.py` files tell Python that the directories should be treated as packages. The `__init__.py` file can just be an empty file, but often it contains some some initialization code for the package.

Now let's create our own package `my_math` for calculation. We put the file `__init__.py` to tell Python this is a package. We also put the files `vector.py` and `circle.py` for the vector and circle-related calculations. The structure of the files are as follows:

```
my_math/          Top-level package
    __init__.py    Initialize the my_math package
    vector.py
    circle.py
```

We can import the package `my_math` in the same way we have imported other third-party packages:

```
import my_math.circle
my_math.circle.area(3)
28.26
```

Or we could only import the module `circle`:

```
from my_math import circle
circle.area(3)
28.26
```

Sub-Packages

A package often has a large number of files, and a hierarchical structure is often imposed by grouping the modules files into different folders and include a `__init__.py` file in the folders. Take a look of the source code of **NumPy** from GitHub and you can see that modules are organized via folders. For example, the modules related to random sampling are under the sub-folder `numpy/random` and the modules related to polynomial calculation are under the sub-folder `numpy/polynomial`. For each sub-folder, there is another `__init__.py`. Each sub-folder therefore is a sub-package. For **NumPy**, `numpy.random` is a sub-package.

Let us create another package `my_math_2`, which with the following structure:

```
my_math_2/              Top-level package
├───__init__.py          Initialize the my_math_2 package
├───linear_algebra/      Sub-package for linear algebra
│   └─── __init__.py
│   └─── vector.py
│   └─── matrix.py
└───geometry/            Sub-package for geometry
    └─── __init__.py
```

```
└── circle.py
└── triangle.py
```

To use a sub-package from the package we can import it by:

```
from my_math_2.geometry import circle
circle.area(3)
28.26
```

The *Python Package Index (PyPI)* is a repository of software for the Python programming language. Packages are shared

Python Package Repositories and Package Installer

Similar to CRAN in R, Python has its own software repositories. Python packages are typically installed from one of two package repositories:

- The Python Package Index (PyPI)
- Conda

PyPI is the official third-party software repository for Python, and it is the default source for packages for `pip`, which is a popular package installer for Python. The name Conda is for both the general-purpose package management system and the package repository. Unlike PyPI, Conda manages software of *any* language.

Publishing a Package

To publish the package, see the official Python document and PyPI

Useful Links and Resources

- Official Python tutorial on modules and packages
- Official Python tutorial on packaging Python projects

Documenting Code

Documentation

You have (hopefully) typed `help` or `?` at least twice in this course while studying the workings of R and Python, and, hence, can appreciate how important it is for software to be documented well. Documentation allows both developers and users to at least get an idea what a function or package does and how, or what particular objects are, without necessarily having to read the code. In fact, it would be almost impossible to write Python or R code without having access to documentation of Python or R!

Documenting R Code

R objects are documented in files with extension `.Rd`, written in "R documentation" (Rd) format Rd is a simple markup language much of which closely resembles (La)TeX and can be processed into a variety of formats, including LaTeX, HTML and plain text. Typically, documentation files are distinct from the script files of an R package, put under the `man/` directory in the directory structure of an R package. A clear description of how `.Rd` files should be authored and the conventions for their elements is at the "Writing R extensions" resource.

The **roxygen2** R package considerably simplifies the documentation of R objects, by enabling users to document them "in place," directly in the scripts that they are defined. To illustrate, recall the `circle.R` in the `R/` directory of **areacalc**'s directory structure. If we open it we see

```
circle <- function(radius) {
    shape <- list(radius = radius)
    class(shape) <- "circle"
    shape
}

#' @export
area.circle <- function(object) {
    pi * object$radius^2
}
```

Both `circle()` and `area.cicle()` are currently not documented; that is if you type `?circle` or `?area.circle` no guidance on how these functions are used comes up. In order to add some documentation we edit the file to look like

```
#' Define a circle through its radius
#'
```

```
#' This returns an object of class `"circle"` which specifies a circle
through its radius.
#'
#' @param radius a [`numeric`] of length one.
#' @return A list inheriting from `"circle"` with a single element
`"radius"`.
#' @seealso [area.circle()]
#' @examples
#' ci <- circle(2.2)
#' ci
#' @export
circle <- function(radius) {
    shape <- list(radius = radius)
    class(shape) <- "circle"
    shape
}

#' Compute the area of a circle generated using [circle()]
#'
#' This computes the area of a circle as defined by the [circle()].
#'
#' @param object an object of class `"circle"`, as generated for example
using the function [circle()].
#' @seealso [circle()]
#' @examples
#' ci <- circle(3)
#' area(ci)
#' @export
area.circle <- function(object) {
    pi * object$radius^2
}
```

As you notice **roxygen2** comments start with `#'`. In this file we have added documentation for two functions, with the documentation appearing *before* the definition of each function. The first line of the documentation is the title of the help file. The following line is a short description of what the function does. `@param` followed by the argument name is used to document each of the function arguments; for functions with multiple arguments we use multiple lines starting with `@param` followed by the argument name. `@seealso` provides links to other related objects, and after `@examples` we have a code chunk that illustrates what the function does. `@export` is a statement we have already seen last week, and it is used to tell `document()` that we want this function to be visible to the users of the package. Everything else is just plain Markdown markup. A detailed account of the conventions used and how Markdown is translated in `.Rd` markup can be found in **roxygen2**'s pages.

Now,

```
document("~/Repositories/areacalc")
```

will parse all scripts, and produce appropriate `.Rd` files under `man/`. If you install **areacalc** and type `?circle` you will now see the help page for the function `circle()`.

The basic steps for creating documentation for objects in an R package are:

- Add **roxygen2** comments to your script files
- Run `document()` to convert these **roxygen2** comments to `.Rd` files

- Preview documentation using `?` or `help()` on the function names
- Repeat this process until you are satisfied with the documentation

As an exercise, try documenting all methods and generics in the **areacalc** package. Make sure you consult **roxygen2**'s pages or search in the web if anything does not work for you.

Documenting Python Code

Documenting Python code and projects is typically done using `Docstrings`. We are not going to get into the details, but if you are interested a very clear tutorial is provided at realpython.com's pages. Sphinx is another popular tool (with very detailed documentation!) that makes it easy to create intelligent documentation for Python (and not only) projects.

Useful Links and Resources

- Writing R documentation files from the *Writing R extensions* resource.
- **roxygen2**'s pages: Guides and tutorials on how to use **roxygen** for documentations of R objects.
- realpython.com's pages: A tutorial on how do document Python code.
- Sphinx: A popular tool for generating intelligent and good-looking documentation for Python code and more.

Test-Driven Software Development

Software Testing

Most probably, you have already been testing things out while writing code, by inspecting the output of your programs and, if it is not as expected, going back to the code, inspecting the output again, and so on, until you are happy with the result.

As can be inferred from the various software development models we introduced last week, testing is not only a vital part of software development process, but the various SDLC models (see “Introduction to Software Development” section) also define when and how software testing is done. Testing your code ensures that it does what you want it to do.

Wikipedia’s page on software testing provides a good overview of the many software testing approaches (e.g. white-box testing, black-box testing, grey-box testing) and software testing levels (unit testing, integration testing, system testing, acceptance testing) that have been devised.

In this page we will focus on *unit testing* because of its usefulness and importance when developing software for data science projects.

Unit Testing

According to Wikipedia’s page on software testing

> Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors

Unit testing refers to tests of the components of the software (e.g. functions and classes) to verify that those and the software itself work as expected. It is typically done using inputs and testing against an expected output. Remarkably, the most experienced programmers start by writing out the test before even writing the code itself!

For example, suppose you intend to add a `rectangle()` functions in **areacalc**. An experienced programmer would first write some unit tests for that function, perhaps the most basic of those being

```
isTRUE(rectangle(length = 2, width = 3)$length == 2)
Error in rectangle(length = 2, width = 3): could not find function
"rectangle"
isTRUE(rectangle(length = 2, width = 3)$width == 3)
Error in rectangle(length = 2, width = 3): could not find function
"rectangle"
isTRUE(class(rectangle(length = 2, width = 3)) == "rectangle")
```

```
Error in rectangle(length = 2, width = 3): could not find function
"rectangle"
```

Of course, all tests above *fail* with errors, simply because the code for `rectangle()` has not been written yet! This may seem like a waste of time, but it pays off very quickly.

Let's think for a moment what have we achieved by writing the tests: i) We have determined that for a well-specified rectangle ones needs to specify a length and a width, ii) We thought about the interface of the `rectangle()` function, i.e. it should take as input the arguments `length` and `width`, iii) We decided that the result should be a list with elements `length` and `width` and of class "rectangle," iv) We set ourselves the goal to write code for passing the tests. So, oftentimes by writing the tests you have almost written the code! In this case,

```
rectangle <- function(length, width) {
    out <- list(length = length, width = width)
    class(out) <- "rectangle"
    out
}
```

and then, all unit tests above pass

```
isTRUE(rectangle(length = 2, width = 3)$length == 2)
[1] TRUE
isTRUE(rectangle(length = 2, width = 3)$width == 3)
[1] TRUE
isTRUE(class(rectangle(length = 2, width = 3)) == "rectangle")
[1] TRUE
```

Imagine doing the above throughout development and for as many components of your code as possible, testing results, corner cases, exceptions, and so on. You are not only more confident that your code does exactly what you want it to do (validation), but you have also eliminated many potential defects in your code. You also have a list of ready tests to run whenever you change anything to see if anything breaks, hence improving maintainability and making code refactoring easier.

It is inevitable that your tests will eventually fail after changing your code! Once they fail, though, you already know what failed and, if your unit tests are granular enough, you also know why. Then, you can use the techniques (debugging and error handling) and the strategies (e.g. defensive programming) we discussed on the "Exceptions, Error Handling and Debugging in R" section to both fix the issue and improve your code.

Testing Frameworks in R

There is nothing wrong with doing unit testing by incrementally adding tests into a script, which you then run whenever you make changes or additions to the software.

However, it is best to use one of the specialized dedicated R packages, like **testthat** and **tinytest**, which allow you to unit test in a more effective and structured manner, and also integrate the unit tests within your R package. For example, with **tinytest** we can write the above unit tests, and a few more for **areacalc**, as

```
library("devtools")
Loading required package: usethis
load_all("~/Repositories/areacalc")
Loading areacalc
library("tinytest")
re <- rectangle(length = 2, width = 3)
expect_true(is.list(re))
----- PASSED      : <-->
 call| expect_true(is.list(re))
expect_identical(re$length, 2)
----- PASSED      : <-->
 call| expect_identical(re$length, 2)
expect_identical(re$width, 3)
----- PASSED      : <-->
 call| expect_identical(re$width, 3)
expect_identical(class(re), "rectangle")
----- PASSED      : <-->
 call| expect_identical(class(re), "rectangle")
expect_error(circle(diameter = 3))
----- PASSED      : <-->
 call| expect_error(circle(diameter = 3))
expect_equal(area(circle(1)), pi)
----- PASSED      : <-->
 call| expect_equal(area(circle(1)), pi)
```

Both **testthat** and **tinytest** have detailed vignettes and tutorials on how they can be used and be integrated within an R package; just follow the links on their CRAN pages. As a note of caution, it is not recommended to mix **tinytest** and **testthat** unit testing, or to have them both loaded at the same R session.

Testing Frameworks in Python

Python, like R, provides unit testing frameworks, like **unittest** and **pytest**. The unit testing tutorial by Anthony Shaw is an accessible tutorial about unit testing in Python.

Useful Links and Resources

- Wikipedia's page on software testing
- "Getting started with testing in Python" by Anthony Shaw