

Chapter 4

Software Building Blocks: From Python to Version Control



Damith Herath, Adam Haskard, and Niranjan Shukla

4.1 Learning Objectives

Software is an essential part of robotics. In this chapter, we will be looking at some of the key concepts in programming and several tools we use in robotics. At the end of the chapter, you will be able to:

- Develop a familiarity with common programming languages used in robotics
- Learn about the fundamental programming constructs and apply them using the Python programming language
- Understand the importance of version control and how to use basic commands in Git
- Select appropriate tools and techniques needed to develop and deploy code efficiently

4.2 Introduction

Whether working with an industrial-grade robot or building your hobby robot, it is difficult to avoid coding. Coding or programming is how you instruct a robot to perform a task. In robotics, you will encounter many different programming languages, including programming languages such as C++, Python, and scientific

D. Herath (✉)

Collaborative Robotics Lab, University of Canberra, Canberra, ACT, Australia

e-mail: Damith.Herath@Canberra.edu.au

A. Haskard

Bluerydge, Canberra, ACT, Australia

e-mail: Adam.Haskard@bluerydge.com

N. Shukla

Accenture, Canberra, ACT, Australia

languages like MATLAB®. While many of the examples in this book will utilise Python, there will be instances where we will use code examples in C/C++ or MATLAB®. While we do not assume any prior programming knowledge, previous coding experience will undoubtedly help you advance quicker.

The following section will briefly outline some of the essential programming constructs. By any means, this is neither exhaustive nor comprehensive. It is simply to introduce you to some fundamental programming concepts that will be useful to get started if you do not already have any programming experience. We will begin with a few essential programming tools such as flowcharts and pseudocode and then expand into fundamental building blocks in programming. If you already have some experience in programming, you may skip this section.

In the subsequent sections, we will discuss two important software tools that would be extremely useful in programming robots, version control and containerisation. While these are all great starting points, there is no better way to build your confidence and skills than to practice and dive into coding. So, we will introduce many case studies and provide code snippets throughout the book for you to follow and try and a comprehensive set of projects at the end of the book. Once you have some confidence, you must explore new problems to code to develop your skills.

4.2.1 Thinking About Coding

As you may have already noticed, we use programming and coding interchangeably, and they both mean instructing your robot to do something logically. Before you start programming, it is essential to understand the problem you are going to address and develop an action plan for how to construct the code. Flowcharts and pseudocode are two useful tools that will help you with this planning phase. Once you have the programme's general outline, you will need to select the appropriate programming language for the task. For tasks where execution speed is important or low-level hardware is involved, this is usually a language like C or C++. However, when the intention is rapid prototyping, a language like Python comes in handy. Robotics researchers also tend to use languages like MATLAB® that are oriented towards mathematical programming. MATLAB® is a proprietary language developed by MathWorks¹ and provides a set of toolboxes with commonly used algorithms, data visualisation tools, allowing for testing complex algorithms with minimal coding. In addition to such code-based languages, several visual programming languages such as Max/MSP/Jitter, Simulink, LabVIEW, LEGO NXT-G are regularly used by roboticists, artists and enthusiasts for programming robots and robotic systems. Whatever language you use, the basic programming constructs are the same.

Irrespective of the programming language used, it is common to think of a programme as a set of inputs to be processed to deliver the desired output (Fig. 4.1).

¹ <https://www.mathworks.com/>.

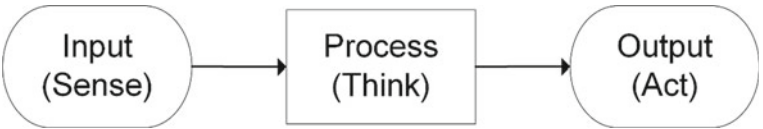


Fig. 4.1 A simple program flows from input to output after processing in the middle

In robotics, a similar framework is used called the sense–think–act loop, which we will explore further in Chap. 7.

4.2.1.1 Flowcharts

Flowcharts are a great way to think about and visualise the flow of your program and the logic. They are geometric shapes connected by arrows (see Figs. 1 and 2. The geometric shapes represent various activities that can be performed, and the arrows indicate the order of operation (*flowline*). Generally, the flowcharts flow from top to bottom and left to right. Flowcharts are a handy tool to have when first starting in programming. They give you a visual representation of the programme without needing to worry about the language-specific syntax. However, they are cumbersome to use in large programmes.

In the following sections, we will explore the meaning of these symbols further.




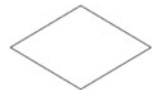

	Start/End (<i>Terminal</i>)	Indicates the beginning and the end of a programme.
	Input/Output	Indicates input and output operations. For example, this could be sensory input or keyboard input and a display output or a command to a motor.
	Process	This is where the upcoming information is processed.
	Decision	Indicates when a logical decision must be made. It contains one input and two outputs.
	Flowline	Order of operation.

Fig. 4.2 Common flowchart elements

Fig. 4.3 A simple pseudocode example with a repetitive read, process, output loop

```
Program input_process_output
```

```
repeat  
    read input data  
    process input data  
    output the processed data  
until user exit
```

4.2.1.2 Pseudocode

Pseudocode is another tool that you can use to plan your code. You could think of them as simply replacing the geometric shapes discussed in the previous section in flowcharts with instruction based on simple English language statements. As the name suggests, pseudocode is programming code without aligning with a specific programming language. Therefore, pseudocode is a great way to write your programming steps in a code-like manner without referring to any particular language. For example, the input, process, output idea could be presented in simple pseudocode form, as shown in Fig. 4.3. In this example, we have extended the previous program by encompassing the read, process, output block within a repetitive loop structure, discussed later in the chapter. In this variation of the program, the input, process, output sequence repeats continually until the user exits the program. The equivalent flowchart is shown in Fig. 4.4.

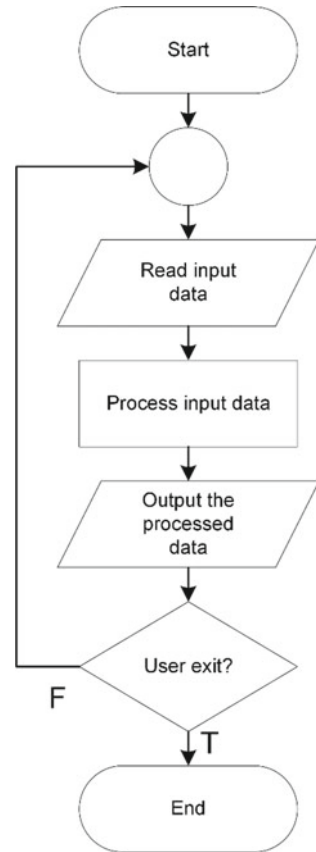
4.3 Python and Basics of Programming

First released in the 1990s, Python² is a high-level programming language. Python is an interpreted language meaning it is processed while being executed compared to a compiled language which needs to be processed before it is executed. Python has become a popular language for programming robots. This may be due to its easily readable language, the visually uncluttered and dynamically typed nature, and the availability of many ready-to-use libraries that provide common functionalities such as mathematical functions. Python is useful when you want to rapidly prototype as it requires minimal lines of code to realise complex tasks. It also alleviates another major headache for beginner programmers by being a garbage collecting language. Garbage collection is the automatic process by which memory is managed and used by the program.

Python uses indentation (whitespace or a tab inserted at the beginning of a line of code) to identify blocks of code. Unlike languages like C/C++ and Java that uses curly brackets { } to delimit code blocks, it is vital to maintain proper indentation

² <https://www.python.org/>.

Fig. 4.4 Flowchart diagram of a simple read, process, output loop



in Python for your code to work correctly. This requirement also improves code readability and aesthetics.

Let us now explore some of the common programming constructs with the help of Python as the example language.

4.3.1 Variables, Strings and Assignment Statements

Python is a dynamically typed language, which means that the variables are not statically typed (e.g. string, float, integer). Therefore, developers do not need to declare variables before using them or declare their type. In Python, all variables are an object.

A typical component of many other programming languages is that variables are declared from the outset with a specific data type, and any value assigned to it during its lifetime must always have that type. One of the accessibility components of Python

is that its variables are not subject to this restriction. In Python, a variable may be assigned a value of one type and later reassigned a new value of a different type. Every value in Python has a datatype. Other data types in Python include Numbers, Strings, Dictionary and many more. Variables are quickly declared by any name or even alphabets like a, ab, abc, so on and so forth.

Strings are a useful and widely used data type in Python. We create them by enclosing characters in quotes. Python treats single quotes and double quotes the same. Creating strings is as simple as assigning a value to a variable. For example,

```
var1 = 'Hello World!'
var2 = "Banana Robot"
```

We see two variables notated by the 'var1' and 'var2' labels in the example above. A simple way is to think of a variable as a name attached to a particular object. To create a variable, you just assign it a value and then start using it. The assignment is achieved with a single equal sign (=).

4.3.2 Relational and Logical Operators

To manage the flow of any program and in every programming language, including Python, conditions are required. Relational and logical operators define those conditions.

As an example, and for context, when you are asked if 3 is greater than 2, the response is yes. In programming, the same logic applies.

When the compiler is provided with some condition based on an *expression*, it computes the expression and executes the condition based on the output of the expression. In the case of relational and logical expressions, the answer will always be either *True* or *False*.

Operators are conventional symbols that bring *operands* together to form an expression. Thus, operators and operands are the deciding factors of the output.

Relational operators are used to define the relationship between two operands. Examples are less than, greater than or equal to operators. Python understands these types of operators and accordingly returns the output, which can be either *True* or *False*.

```
1 < 10
True
```

1 is Less Than 10, so the Output Returned is True.

A simple list of the most common operators:

1. **Less than** → used with <
2. **Greater than** → used with >
3. **Equal to** → used with ==
4. **Not equal to** → used with !=

5. Less than or equal to → used with <=
6. Greater than or equal to → used with >=

Logical operators are used in expressions where the operands are either True or False. The operands in a logical expression can be expressions that return True or False upon evaluation.

There are three basic types of logical operators:

1. **AND**: For AND operation, the result is True if and only if both operands are True. The keyword used for this operator is and.
2. **OR**: For OR operation, the result is True if either of the operands is True. The keyword used for this operator is or.
3. **NOT**: The result is True if the operand is False. The keyword used for this operator is not.

4.3.3 Decision Structures

Decision structures allow a program to evaluate a variable and respond in a scripted manner. At its core, the decision-making process is a response to conditions occurring during the execution of the program, with consequential actions taken according to the conditions. Basic decision structures evaluate a series of expressions that produce TRUE or FALSE as the output. The Python programming language provides you with the following types of decision-making sequences.

1. **if** statements: An if statement consists of a Boolean expression followed by one or more statements.
2. **if...else** statements: An if statement can be followed by an optional else statement, which executes when the Boolean expression is FALSE.
3. **nested if** statements: You can use one if or else if statement inside another if or else if statement(s).

Below is an example of a one-line if clause,

```
# this is a comment (beginning with the # symbol).
# Comments are important documentation element in programming
var = 1300#a variable assignment
if (var == 1300): print "Value of expression is 1300" #decision
structure in a single line
print "Bye!"#display the word Bye!
```

When the above code runs, the following is the output,

```
Value of expression is 1300
Bye!
```

In general, statements are executed sequentially. The first statement in a function is executed first, followed by the second, and so on. It is good to think of code as just a set of instructions, not too different from a favourite cooking recipe. There may be

a situation when you need to execute a block of code several times. A loop statement allows us to execute a statement or group of statements multiple times.

4.3.4 *Loops*

There are typically three ways for executing loops in Python. They all provide similar functionality; however, they differ in their syntax and condition checking time.

1. **While loop:** Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2. **For loop:** Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3. **Nested loops:** You can use one or more loops inside any another while, for or do..while loop.

```
# while loop.
count = 0.
while (count < 3):
    count = count + 1#note the indentation to indicate this section
of the code is inside the loop.
    print("Hello Robot")
```

When the code above is run, we would expect to see the following output.

```
Hello Robot
Hello Robot
Hello Robot
```

4.3.5 *Functions*

A function is a block of code designed to be reusable which is used to perform a single action. Functions give developers modularity for the application and a high degree of reusable code blocks. A well-built function library lowers development time significantly. For example, Python provides functions like `print()`, but users can develop their own functions. These functions are called user-defined functions.

e.g.

```
def robot_function():
    print("Robot function executed")
# You can then call this function in a different part of your program;
robot_function()
```

When you execute the code, the following will be displayed.

```
Robot function executed
```


You can pass external information to the function as arguments. Arguments are listed inside the parentheses that come after the function name.

e.g.

```
def robot_function(robot_name):
    print("Robot function executed for robot named " + robot_name)
```

We have modified the previous function to include an argument called `robot_name`. When we call the new function, we can now include the name of the robot as an argument:

```
robot_function('R2-D2')
which will result in the following output.
Robot function executed for robot named R2-D2
```

4.3.6 Callback Function

A *callback* function is a special function that can be passed as an argument to another function. The latter function is designed to call the former callback function in its definition. However, the callback function is executed only when it is required. You will find many uses for such functions in robotics. Particularly, when using ROS, you will see the use of callback functions to read and write various information to and from robotic hardware which may happen asynchronously. A simple example illustrates the main elements of a callback function implementation.

```
def callbackFunction(robot_status):
    print("Robot's current status is " + robot_status)
def displayRobotStatus(robot_name, callback):
    # This function takes robot_name and a callback function as
    # arguments
    # The code to read the robot status (stored in the variable
    # robot_status) goes here
    # the read status is then passed to the callback function
    callback(robot_status)
```

You can now call the `displayRobotStatus` function in your main program.

```
if __name__ == '__main__':
    displayRobotStatus("R2-D2", callbackFunc)
```

4.4 Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm based on the concept of ‘objects’, which may contain data in the form of fields, often known

as attributes, and code in the form of procedures, often known as methods. Here is a simple way to think about this idea;

1. A person is an object which has certain properties such as height, gender and age.
2. The person object also has specific methods such as move, talk and run.

Object—The base unit of object-oriented programming that combines data and function as a unit.

Class—Defining a class is defining a blueprint for an object. Describes what the class name means, what an object of the class will consist of and what operations can be performed on such an object. A class sets the blank canvas parameters for an object.

OOP has four basic concepts,

1. **Abstraction**—It provides only essential information and hides their background details. For example, when ordering pizza from an application, the back-end processes for this transaction are not visible to the user.
2. **Encapsulation**—Encapsulation is the process of binding variables and functions into a single unit. It is also a way of restricting access to certain properties or components. The best example for encapsulation is the generation of a new class.
3. **Inheritance**—Creating a new class from an existing class is called inheritance. Using inheritance, we can create a child class from a parent class such that it inherits the properties and methods of the parent class and can have its own additional properties and methods. For example, if we have a class robot with properties like model and type, we can create two classes such as Mobile_robot and Drone_robot from those two properties, and additional properties specific to them such that Mobile_robot has a number of wheels while a Drone_robot has a number of rotors. This also applies to methods.
4. **Polymorphism**—The definition of polymorphism means to have many forms. Polymorphism occurs when there is a hierarchy of classes, and they are related by inheritance.

4.5 Error Handling

A Python program terminates as soon as it encounters an error. In Python, an error can be a syntax (typo) error or an exception. Syntax errors occur when the python parser detects an incorrect statement. Observe the following example:

```
>>> print( 0 / 0 )
1^
SyntaxError: invalid syntax
```

The arrow character points to where the parser has run into a **syntax error**. In this example, there was one bracket too many. When it is removed, the code will run without any error:

```
>>> print( 0 / 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

This time, Python has ‘thrown’ an **exception error**. This type of error occurs whenever correct Python code results in an error. The last line of the message indicated what type of exception error was thrown. In this instance, it was a `ZeroDivisionError`. Python has built-in exceptions. Additionally, the possibility exists to create user-defined exceptions.

4.6 Secure Coding

Writing secure code is essential for protecting data and maintaining the correct behaviour of the software. Writing secure code is a relatively new discipline, as typically developers have been commissioned to write functions and outputs, not necessarily in a secure manner. However, given the prevalence of exploits, it is important developers build in sound security practices from the outset.

Python development security practices to consider:

1. **Use an up-to-date version of Python:** Out of date versions have since been rectified with vulnerability updates. Not incorporating the updates into the python environment ensures vulnerabilities are available to exploit.
2. **Build the codebase in a sandbox environment:** Using a sandbox environment prevents malicious Python dependencies pushed into production. If malicious packages are present in Python environments, using a virtual environment will prevent having the same packages in the production codebase as it is isolated.
3. **Import packages correctly:** When working with external or internal Python modules, ensure they are imported using the right paths. There are two types of import paths in Python, and they are absolute and relative. Furthermore, there are two types of relative imports, implicit and explicit. Implicit imports do not specify the resource path relative to the current module, while Explicit imports specify the exact path of the module you want to import. Implicit import has been disapproved and removed from Python 3 onwards because if the module specified is found in the system path, it will be imported, and that could be very dangerous, as it is possible for a malicious module with an identical name to be in an open-source library and find its way to the system path. If the malicious module is found before the real module, it will be imported and used to exploit applications in their dependency tree. Ensure either absolute import or explicit relative imports as it guarantees the authentic and intended module.
4. **Use Python HTTP requests carefully:** When you send HTTP requests, it is always advisable to do it carefully by knowing how the library you are using handles security to prevent security issues. When you use a common HTTP request library like Requests, you should not specify the versions down in

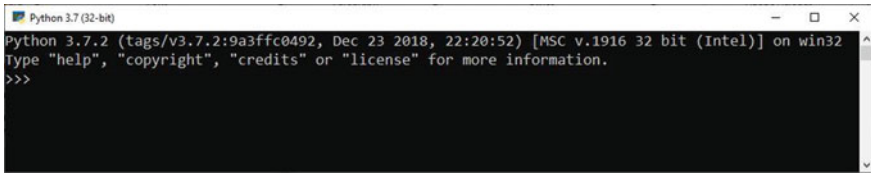


Fig. 4.5 Python command line

your requirements.txt because in time that will install outdated versions of the module. To prevent this, ensure you use the most up-to-date version of the library and confirm if the library is handling the SSL verification of the source.

5. Identify exploited and malicious packages.

Packages save you time as you don't need to build artefacts from scratch each time. Packages can be easily installed through the *Pip* package installer. Python Packages are published to PyPI³ in most cases, which essentially is code repository for Python Packages which is not subject to security review or check. This means that PyPI can easily publish malicious code.

Verify each Python package you are importing to prevent having exploited packages in your code. Additionally, use security tools in your environment to scan your Python dependencies to screen out exploited packages.

4.7 Case Study—Writing Your First Program in Python

To start experimenting with Python, you can install the current version of the Python program from the Python website.⁴ Follow the instruction on this website to download the recommended current version of your operating system. Once installed, you can call the **Python (command line)** shell for an interactive programming environment (see Fig. 4.5).

In any programming language, the Hello World program is a shared bond between all coders. You can go ahead and make your own 'hello world' program. Look at the classic example below. Note that the # symbol is a comment line, which means Python does not read this as code to execute. Instead, it is intended for human audiences, so coders can easily see what each line of code is supposed to do. Commenting well and regularly is key to good collaboration and development hygiene.

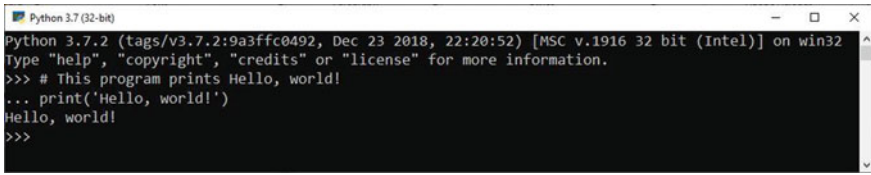
```
# This program prints Hello, world!
print('Hello, world!')
```

Output.

```
Hello, world!
```

³ <https://pypi.org/>.

⁴ <https://www.python.org/downloads/>.

A screenshot of a Python 3.7 (32-bit) command line window. The window title is "Python 3.7 (32-bit)". The prompt is "Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916 32 bit (Intel)] on win32". The user enters "Type 'help', 'copyright', 'credits' or 'license' for more information." followed by ">>> # This program prints Hello, world!". Then the user enters "... print('Hello, world!')". The prompt changes to "Hello, world!" and the user enters ">>>".

```
Python 3.7 (32-bit)
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> # This program prints Hello, world!
... print('Hello, world!')
Hello, world!
>>>
```

Fig. 4.6 Hello, World program interactively executed in a Python command line window

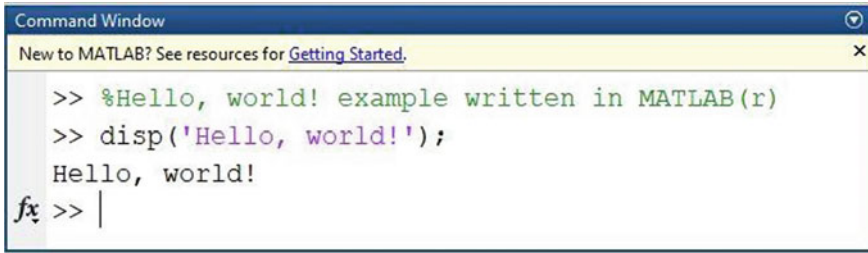
4.7.1 A Note on Migrating from MATLAB® to Python

As you dwell into robotics programming and writing algorithms, you will notice that many examples are written in MATLAB®, particularly in academia due to previously mentioned reasons. However, there are compelling reasons to use Python instead of a proprietary language like MATLAB. One of the main reasons is the cost of acquiring MATLAB and related toolboxes. Python allows you to easily distribute your code without worrying about your end-users needing to purchase MATLAB® licences to run your code. In addition, Python being a general-purpose programming language offers you a better development environment for projects targeting a wide use and deployment audience.

If you are thinking of migrating any code from MATLAB® to Python, the good news is that the two languages are 'very similar'. This allows for relatively easy transitioning from MATLAB to Python. One of the key reasons for MATLAB's popularity has been its wide array of well-crafted toolboxes by experts in the field. For example, there are several popular toolboxes related to robotics including the *Robotics Toolbox* developed by Peter Corke.⁵ These toolboxes provide specific mathematical functions reducing the time it takes to develop new code when building or testing new ideas for your robot. Python also offers a similar mechanism to expand its capabilities through Python packages. For example, one of the powerful elements of MATLAB is its native ability to work with matrices and arrays (side note: matrices and arrays will play a major role in robotics programming!). Python, being a general-purpose language does not have this capability built-in. But a package available in Python called NumPy⁶ provides a way to address this through multidimensional arrays allowing you to write fast, efficient, and concise matrix operators comparable to MATLAB. As your knowledge in robotics and programming matures, it would be a worthwhile investment to spend some time to explore the similarities and differences between the two languages and to understand when to utilise one or the other. Figure 4.7 shows our humble Hello world program being executed in a MATLAB® command line window. Can you spot the differences between the syntaxes from our Python example in Fig. 4.6?

⁵ <https://petercorke.com/toolboxes/robotics-toolbox/>.

⁶ <https://numpy.org/>.

A screenshot of a MATLAB Command Window. The window has a blue title bar with the text 'Command Window'. Below the title bar is a yellow banner with the text 'New to MATLAB? See resources for [Getting Started.](#)'. The main area of the window is white and contains the following text: '>> %Hello, world! example written in MATLAB(r)' in green, '>> disp('Hello, world!');' in purple, 'Hello, world!' in black, and 'fx >> |' in black. The 'fx' icon is a small blue square with a white 'fx' inside.

```
Command Window
New to MATLAB? See resources for Getting Started.
>> %Hello, world! example written in MATLAB(r)
>> disp('Hello, world!');
Hello, world!
fx >> |
```

Fig. 4.7 Hello, World program interactively executed in a MATLAB command line window

4.8 Version Control Basics

Version control is the practice of managing changes to the codebase over time and potentially between multiple developers working on the same project. It is alternatively called *source control*. Version control provides a snapshot of development and includes tracking of code *commits*. It also provides features to merge the code contributions arising from multiple sources, including managing merge conflicts.

A version control system (or source control management system) allows the developer to provide a suite of features to track code changes and switch to previous versions of the codebase. Further, it provides a collaborative platform for teamwork while enabling you to work independently until you are ready to commit your work. A version control system aims to help you streamline your work while providing a centralised home for your code. Version control is critical to ensure that the tested and approved code packages are deployed to the production environment.

4.8.1 Git

Git is a powerful open-source distributed version control system.⁷ Unlike other version control systems, which think of version control as a list of file-based changes, Git thinks of its data more like a series of snapshots of a miniature filesystem. A snapshot is a representation of what all the files look like at a given moment. Git stores reference to snapshots as part of its version management.

Teams of developers use Git in varying forms because of Git's distributed and accessible model. There is no policy on how a team uses Git. However, projects will generally develop their own processes and policies. The only imperative is that the team understands and commits to the workflow process that maximises their ability to commit code frequently and minimise merge conflicts.

A Git versioned project consists of three areas: the *working tree*, the *staging area* and the *Git directory*.

⁷ <https://git-scm.com/>.

As you progress with your work, you typically stage your commits to the staging area, followed by committing them to the Git directory (or repository). At any time, you may *checkout* your changes from the Git directory.

4.8.1.1 Install Git

To check if Git has already been bundled with your OS, run the following command (at the command prompt):

```
git --version
```

To install Git, head over to the download site⁸ and select the appropriate version for your operating system and follow the instructions.

4.8.1.2 Setting up a Git Repository

To initialise a Git repository in a project folder on the file system, execute the following command from the root directory of your folder:

```
git init
```

Alternatively, to clone a remote Git repository into your file system, execute the following command:

```
git clone <remote_repository_url>
```

Git repositories provide SSH URLs of the format `git@host:user_name/repository_name.git`.

Git provides several commands for this syncing with a remote repository:

Git remote: This command enables you to manage connections with a remote repository, i.e. create, view, update, delete connections to remote repositories. Further, it provides you with an alias to reference these connections instead of using their entire URL.

The below command would list the connections to all remote repositories with their URL.

```
git remote -v
```

The below command creates a new connection to a remote repository.

```
git remote add <repo_name> <repo_url>
```

The below command removes a connection to a remote repository.

```
git remote rm <repo_name>
```

⁸ <https://git-scm.com/download/>.

The below command renames a remote connection from `repo_name_1` to `repo_name_2`

```
git remote rename <repo_name_1> <repo_name_2>
```

Upon cloning a remote repository, the connection to the remote repository is called *origin*.

To pull changes from a remote repository, use either *Git fetch* or *git pull*.

To fetch a specific branch from the remote repository, execute the below command:

```
git fetch <repo_url> <branch_name>
```

where `repo_url` is the name of the remote repository, and `branch_name` is the name of the branch.

Alternatively, to fetch all branches, use the below command:

```
git fetch --all
```

To pull the changes from the remote repository, execute the following command:

```
git pull <repo_url>
```

The above command will fetch the remote repository's copy of your current branch and will merge the changes into your current branch.

If you would like to view this process in detail, use the verbose flag, as shown below

```
git pull --verbose
```

As `git pull` uses merge as a default strategy, if you would like to use rebase instead, execute the below command:

```
git pull --rebase <repo_url>
```

To push changes to a remote repository, use `git push`, as described below:

```
git push <repo_name> <branch_name>
```

Where `repo_name` is the name of the remote repository, and `branch_name` is the name of the local branch.

4.8.1.3 Git SSH

An SSH key is an access credential for the secure shell network protocol. SSH uses a pair of keys to initiate a secure handshake between remote parties—a public key and a private key.

SSH keys are generated using a public key cryptography algorithm.

1. To generate an SSH key on Mac, execute the following command:

```
ssh-keygen -t rsa -b 4096 -C "your_email@domain"
```


2. Upon being prompted to enter the file path, enter a file path to which you would like the key to be stored.
3. Enter a secure passphrase.
4. Add the generated SSH key to the ssh-agent

```
ssh-add -K <file_path_from_step_2>
```

4.8.1.4 Git Archive

To export a Git project to an archive, execute the following command:

```
git archive --output=<output_archive_name> --format=tar HEAD
```

The above command generates an archive from the current HEAD of the repository. The HEAD refers to the current commit.

4.8.1.5 Saving Changes

As you make changes to your local codebase, for instance, feature development or bug fixes, you will want to stage them. To do so, please execute the following command for each file you would like to add to the staging area:

```
git add <file_name>  
git commit -m <commit_message>
```

The first command puts your changes to the staging area while the second command creates a snapshot of these changes, which can then be pushed to the remote repository.

If you would like to add all files in one go, consider using the variation of Git add with the—`all` option.

Once you add the file(s) to the staging area, they are tracked.

4.8.1.6 Syncing

Upon committing changes to the local repository, it is time to update the Git remote repository with the commits from the local repository. Please refer to the syncing commands listed at the start of this section.

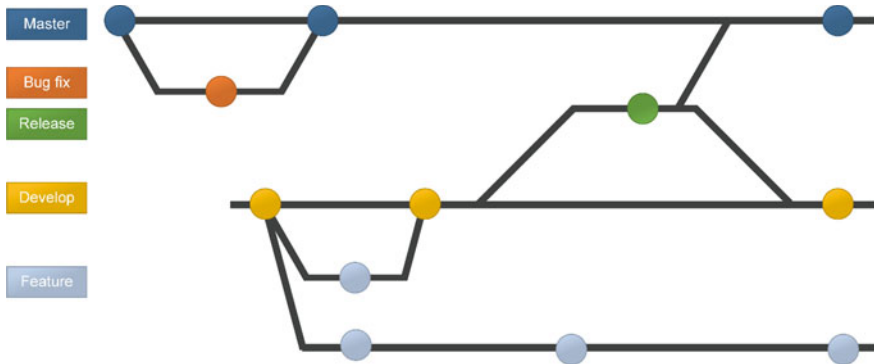


Fig. 4.8 Examples of pull requests

4.8.1.7 Making a Pull Request

A pull request is used to notify the development of changes, such as a new feature or a bug fix so that the development team (or assigned reviewers) can review the code changes (or commits) and either approve/decline them entirely or ask for further changes.

As part of this process:

1. A team member creates a new local branch (or creates their local branch from an existing remote branch) and commits their changes in this branch.
2. Upon finalising the changes, the team member pushes these changes to their own remote branch in the remote repository.
3. The team member creates a pull request via the version control system. As part of this process, they select the source and destination branches and assign some reviewers.
4. The assigned reviewer(s) discuss the code changes in a team, using the collaboration platform that is integrated into the version control system, and ultimately either accepts or declines the changes in full or part.
5. The above step #4 may go through more cycles or reviews.
6. Upon completing the review process, when all changes have been accepted (or approved), the team member merges the remote branch into the code repository, closing the pull request (Fig. 4.8).

4.8.1.8 Common Git Commands

The table lists some commonly used Git commands that are useful to remember. Figure 4.9 depicts the relative execution direction of some of these commands.

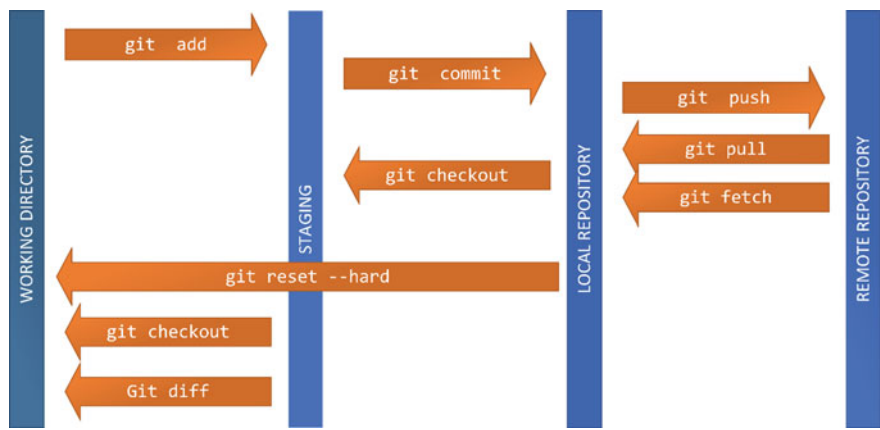


Fig. 4.9 Common git commands and relative execution directions

Configure your username and email address with Git	<code>git config --global user.name "<user_name>"</code>
Initialise a Git repository	<code>git init</code>
Clone a Git repository	<code>git clone <repo_url></code>
Connect to a remote Git repository	<code>git remote add origin <remote_server></code>
Add files to a Git repository	<code>git add <file_name></code>
Check the status of the files	<code>git status</code>
Commit changes to the local repository	<code>git commit -m "<message>"</code>
Push changes to the remote repository	<code>git push origin master</code>
Switch across branches	<code>git checkout -b <branch_name></code> <code>git checkout <branch_name></code> <code>git branch</code> <code>git branch -d <branch_name></code>
Update from the remote repository	<code>git pull</code> <code>git merge <branch_name></code> <code>git diff</code>
Overwrite local changes	<code>git checkout -- <file_name></code> <code>git reset --hard origin/master</code>

4.9 Containerising Applications

A minor difference in the version of a library can alter the functionality of your application, resulting in an unintended outcome. Fortunately, containerising an application allows it to execute in the same way regardless of the workspace or computer that it

is deployed on. You can think of containerisation as an efficient alternative to virtual machines.

*Docker*⁹ is a great tool to consider for containerisation. A key reason why the development community has adopted Docker is that if you containerise your application and transfer the image to a teammate's environment, the application will have the same performance on both devices. This is because the container includes all the dependencies needed by the application.

4.10 Chapter Summary

The chapter began with an introduction to common constructs found in programming and discussed using Python as an example language. The intention has been to provide a starting point for readers who are not familiar with the basics of programming or as a quick refresher for those picking up coding after some lapse in practice. We also discussed a few useful tools in aiding computational thinking, such as flowcharts and pseudocode. We then covered several important concepts, including OOP, error handling, secure coding and version control. Any robotics programmer worth their salt must be well versed in these aspects. Again, we have aimed to provide you with pointers to essential concepts to explore further and build on. Finally, we discussed containerisation as an efficient way to deploy your code on multiple platforms and operating systems. The projects section of the book will provide further opportunities to practice and explore these ideas further.

4.11 Revision Questions

1. What are some of the common programming languages used in robotics?
2. You are required to display the following pattern on a screen. Write the pseudocode of a suitable algorithm for this task.

```
*
* *
* * *
* * * *
* * * * *
```

3. Convert the pseudocode developed in 2. above to a Python implementation.
4. What are the four basic concepts of OOP?
5. What is Git and why is it important?

⁹ <https://www.docker.com/>.

4.12 Further Reading

It is far too numerous to suggest a set of suitable reading for this chapter as there are many online resources as well as excellent books were written on each of the topics covered in this chapter. You may head over to the book's website for a list of up-to-date resources. The following have served as useful online resources in writing this chapter:

- Python programming (Python.org, 2019; Python Application, 2021; Python Exceptions, 2021; Python Security, 2020; Python Tutorial, 2021)
- Git (Atlassian, 2021)
- Containerisation (Docker, 2013)

References

- Atlassian. *Git Tutorials and Training* | Atlassian Git Tutorial. Atlassian. <https://www.atlassian.com/git/tutorials> (Accessed 2021, December 22).
- Docker, *What is a Container?* | Docker. Docker. (2013). <https://www.docker.com/resources/what-container>
- How to Containerize a Python Application. *Engineering Education (EngEd) Program* | Section. <https://www.section.io/engineering-education/how-to-containerize-a-python-application/> (Accessed 2021, December 22).
- Python Exceptions: *An Introduction—Real Python*. <https://realpython.com/python-exceptions> (Accessed 2021, December 22).
- Python Security Practices You Should Maintain. *SecureCoding*. (2020, May 18). <https://www.securecoding.com/blog/python-security-practices-you-should-maintain/>
- Python Tutorial. www.tutorialspoint.com. <https://www.tutorialspoint.com/python>. (Accessed 2021, December 22).
- Welcome to Python.org. (2019, May 29). <https://www.python.org/>

Damith Herath is an Associate Professor in Robotics and Art at the University of Canberra. Damith is a multi-award winning entrepreneur and a roboticist with extensive experience leading multidisciplinary research teams on complex robotic integration, industrial and research projects for over two decades. He founded Australia's first collaborative robotics startup in 2011 and was named one of the most innovative young tech companies in Australia in 2014. Teams he led in 2015 and 2016 consecutively became finalists and, in 2016, a top-ten category winner in the coveted Amazon Robotics Challenge—an industry-focused competition amongst the robotics research elite. In addition, Damith has chaired several international workshops on Robots and Art and is the lead editor of the book “Robots and Art: Exploring an Unlikely Symbiosis”—the first significant work to feature leading roboticists and artists together in the field of Robotic Art.

Adam Haskard is a cyber security and technology professional with over 16 years' experience within the Department of Defence. Adam has led GRC and Security Engineering activities in Defence Gateway Operations, JP2047, AIR6000, 1771 and L4125 as the DIE ITSM. Adam possesses a strong understanding of information systems, cross domain solutions, the certification

and accreditation process and the military and wider technology landscape. He has in-depth technical and GRC experience leading multi-disciplinary teams on sensitive and complex cyber security activities. Adam has gained significant work experience from his various roles in the ADF and Industry, which included Cyber Security Professional, Engineer and Network Security Administrator, that enabled him to develop his cyber security and ICT skills. He was a member of the ADF between 2006–2013 where he progressed through information system (CIS) and leadership-based trainings. Adam's expertise includes evaluating, designing, monitoring, administering and implementing cybersecurity systems, protections and capabilities.

Niranjan Shukla has 15 years of prior experience working as a Software Engineer, Team Lead and TechnoloArchitect with experience in Data-driven development, API Design, Frontend technologies, Data Visualization, Virtual Reality and Cloud. He practices Design thinking through digital-Art on-the-side.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

