A **CRCPress** FREEBOOK

Programming for Data Science, Media Art, and Robotics

CRC Press
Taylor & Francis Group

TABLE OF CONTENTS

Introduction

1 • Introduction to R/Python Programming

2 • Preparing the Data

3 • HTML and CSS

4 • Designing Graphically with the Language of Code

5 • Introduction to ROS: ROS tools and nodes

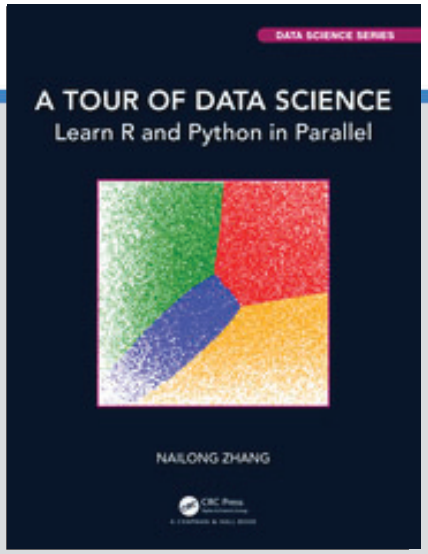
Introduction

Featuring chapters from 5 of our new and best-selling books, this free book introduces popular programming languages, and their use across a number of applications areas, including data science, media art, and robotics. The first three chapters explore programming in R, Python, and JavaScript for Data Science. The next chapter looks at how the popular programming language, Processing, can be used for media art. The last part of the book features a chapter from our best-selling textbook on programming for Robotics, using ROS.

CHAPTER

1

INTRODUCTION TO R/PYTHON PROGRAMMING



A Tour of Data Science

Learn R and Python in Parallel

By Nailong Zhang

Learn more

Introduction to R/Python Programming

In this chapter, I will give an introduction to general R and Python programming in a parallel fashion.

1.1 CALCULATOR

R and Python are general-purpose programming languages that can be used for writing softwares in a variety of domains. But for now, let us start with using them as basic calculators. The first thing is to have them installed. R[1] and Python[2] can be downloaded from their official website. In this book, I will be using R 3.5 and Python 3.7.

To use R/Python as basic calculators, let's get familiar with the interactive mode. After the installation, we can type R or Python (it is case insensitive so we can also type r/python) to invoke the interactive mode. Since Python 2 is installed by default on many machines, in order to avoid invoking Python 2 we type python3.7 instead.

R

```
1
2 ~ $R
3
4 R version 3.5.1 (2018-07-02) -- "Feather Sp\footnoteray"
5 Copyright (C) 2018 The R Foundation for Statistical Computing
6 Platform: x86_64-apple-darwin15.6.0 (64-bit)
7
8 R is free software and comes with ABSOLUTELY NO WARRANTY.
9 You are welcome to redistribute it under certain conditions.
10 Type 'license()' or 'licence()' for distribution details.
11
```

[1] https://www.r-project.org
[2] https://www.python.org

```
  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Python

```
~ $python3.7
Python 3.7.1 (default, Nov  6 2018, 18:45:35)
[Clang 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more
    information.
>>>
```

The messages displayed by invoking the interactive mode depend on both the version of R/Python installed and the machine. Thus, you may see different messages on your local machine. As the messages said, to quit R we can type `q()`. There are 3 options prompted by asking the user if the workspace should be saved or not. Since we just want to use R as a basic calculator, we quit without saving workspace.

To quit Python, we can simply type `exit()`.

R

```
> q()
Save workspace image? [y/n/c]: n
~ $
```

Once we are inside the interactive mode, we can use R/Python as a calculator.

R

```
> 1+1
[1] 2
> 2*3+5
[1] 11
> log(2)
```

```
[1] 0.6931472
> exp(0)
[1] 1
```

Python

```
>>> 1+1
2
>>> 2*3+5
11
>>> log(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'log' is not defined
>>> exp(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'exp' is not defined
```

From the code snippet above, R is working as a calculator perfectly. However, errors are raised when we call `log(2)` and `exp(2)` in Python. The error messages are self-explanatory - `log` function and `exp` function don't exist in the current Python environment. In fact, `log` function and `exp` function are defined in the `math` module in Python. A module[3] is a file consisting of Python code. When we invoke the interactive mode of Python, a few built-in modules are loaded into the current environment by default. But the `math` module is not included in these built-in modules. That explains why we got the `NameError` when we try to use the functions defined in the `math` module. To resolve the issue, we should first load the functions to use by using the `import` statement as follows.

Python

```
>>> from math import log, exp
>>> log(2)
0.6931471805599453
>>> exp(0)
1.0
```

1.2 VARIABLE AND TYPE

In the previous section we have seen how to use R/Python as calculators. Now, let's see how to write real programs. First, let's define some variables.

[3] https://docs.python.org/3/tutorial/modules.html

R

```
> a=2
> b=5.0
> x='hello world'
> a
[1] 2
> b
[1] 5
> x
[1] "hello world"
> e=a*2+b
> e
[1] 9
```

```
>>> a=2
>>> b=5.0
>>> x='hello world'
>>> a
2
>>> b
5.0
>>> x
'hello world'
>>> e=a*2+b
>>> e
9.0
```

Here, we defined 4 different variables `a, b, x, e`. To get the type of each variable, we can utilize the function `typeof()` in R and `type()` in Python, respectively.

```
> typeof(x)
[1] "character"
> typeof(e)
[1] "double"
```

```
>>> type(x)
<class 'str'>
>>> type(e)
<class 'float'>
```

The type of `x` in R is called character, and in Python is called str.

1.3 FUNCTIONS

We have seen two functions `log` and `exp` when we use R/Python as calculators. A function is a block of code which performs a specific task. A major purpose of wrapping a block of code into a function is to reuse the code.

It is simple to define functions in R/Python.

```
> fun1=function(x){return(x
    *x)}
> fun1
function(x){return(x*x)}
> fun1(2)
[1] 4
```

```
>>> def fun1(x):
...     return x*x # note the
    indentation
...
>>> fun1(2)
4
```

Here, we defined a function `fun1` in R/Python. This function takes `x` as input and returns the square of `x`. When we call a function, we simply type the function name

followed by the input argument inside a pair of parentheses. It is worth noting that input or output are not required to define a function. For example, we can define a function `fun2` to print `Hello World!` without input and output.

One major difference between R and Python codes is that Python codes are structured with indentation. Each logical line of R/Python code belongs to a certain group. In R, we use `{}` to determine the grouping of statements. However, in Python we use leading whitespace (spaces and tabs) at the beginning of a logical line to compute the indentation level of the line, which is used to determine the statements' grouping. Let's see what happens if we remove the leading whitespace in the Python function above.

Python

```
>>> def fun1(x):
... return x*x # note the indentation
  File "<stdin>", line 2
    return x*x # note the indentation
         ^
IndentationError: expected an indented block
```

We got an `IndentationError` because of missing indentation.

```
> fun2=function(){print('
    Hello World!')}
> fun2()
[1] "Hello World!"
```

```
>>> def fun2(): print('Hello
    World!')
...
>>> fun2()
Hello World!
```

Let's go back to `fun1` and have a closer look at the `return`. In Python, if we want to return something we have to use the keyword `return` explicitly. `return` in R is a function but it is not a function in Python and that is why no parenthesis follows `return` in Python. In R, `return` is not required even though we need to return something from the function. Instead, we can just put the variables to return in the last line of the function defined in R. That being said, we can define `fun1` as follows.

R

```
> fun1=function(x){x*x}
```

Sometimes we want to give a default value to an argument for a function, and both R and Python allow functions to have default values.

R

Python

```
> log_fun =
    function(x, base=2){
+ return(log(x, base))
+ }
> log_fun(5, base=2)
[1] 2.321928
> log_fun(5, 2)
[1] 2.321928
> log_fun(base=2, 5)
[1] 2.321928
>
```

```
>>> def log_fun(x, base=2):
...
    return math.log(x, base)
...
>>> log_fun(5,2)
2.321928094887362
>>> log_fun(5, base=2)
2.321928094887362
>>> log_fun(base=2, 5)
  File "<stdin>", line 1
SyntaxError: positional
    argument follows keyword
    argument
```

In Python we have to put the arguments with default values at the end, which is not required in R. However, from a readability perspective, it is always better to put them at the end. You may have noticed the error message above about positional argument. In Python there are two types of arguments, i.e., positional arguments and keyword arguments. Simply speaking, a keyword argument must be preceded by an identifier, e.g., base in the example above. And positional arguments refer to non-keyword arguments.

1.4 CONTROL FLOWS

To implement a complex logic in R/Python, we may need control flows.

1.4.1 If/else

Let's define a function to return the absolute value of input.

```
> fun3=function(x){
+   if (x>=0){
+     return(x)}
+   else{
+     return(-x)}
+ }
> fun3(2.5)
[1] 2.5
> fun3(-2.5)
[1] 2.5
```

```
>>> def fun3(x):
...   if x>=0:
...     return x
...   else:
...     return -x
...
>>> fun3(2.5)
2.5
>>> fun3(-2.5)
2.5
```

The code snippet above shows how to use `if/else` in R/Python. The subtle

difference between R and Python is that the condition after `if` must be embraced by parenthesis in R but it is optional in Python.

We can also put `if` after `else`. But in Python, we use `elif` as a shortcut.

```
> fun4=function(x){
+   if (x==0){
+     print('zero')}
+   else if (x>0){
+     print('positive')}
+   else{
+     print('negative')}
+ }
> fun4(0)
[1] "zero"
> fun4(1)
[1] "positive"
> fun4(-1)
[1] "negative"
```

```
>>> def fun4(x):
...   if x==0:
...     print('zero')
...   elif x>0:
...     print('positive')
...   else:
...     print('negative')
...
>>> fun4(0)
zero
>>> fun4(1)
positive
>>> fun4(-1)
negative
```

1.4.2 For loop

Similar to the usage of `if` in R, we also have to use parenthesis after the keyword `for` in R. But in Python there should be no parenthesis after `for`.

```
> for (i in 1:3){print(i)}
[1] 1
[1] 2
[1] 3
```

```
>>> for i in
    range(1,4):print(i)
...
1
2
3
```

There is something more interesting than the `for loop` itself in the snippets above. In the R code, the expression `1:3` creates a vector with elements 1, 2 and 3. In the Python code, we use the `range()` function for the first time. Let's have a look at them.

```
> typeof(1:3)
[1] "integer"
```

```
>>> type(range(1,4))
<class 'range'>
```

`range()` function returns a `range` type object, which represents an immutable

sequence of numbers. `range()` function can take three arguments, i.e., `range(start, stop, step)`. However, `start` and `step` are both optional. It's critical to keep in mind that the `stop` argument that defines the upper limit of the sequence is exclusive. And that is why in order to loop through 1 to 3 we have to pass 4 as the `stop` argument to `range()` function. The `step` argument specifies how much to increase from one number to the next. The default values of `start` and `step` are 0 and 1, respectively.

1.4.3 While loop

R

```
> i=1
> while (i<=3){
+   print(i)
+   i=i+1
+ }
[1] 1
[1] 2
[1] 3
```

```
>>> i=1
>>> while i<=3:
...   print(i)
...   i+=1
...
1
2
3
```

You may have noticed that in Python we can do `i+=1` to add 1 to `i`, which is not feasible in R by default. Both for loop and while loop can be nested.

1.4.4 Break/continue

Break/continue helps if we want to break the for/while loop earlier, or to skip a specific iteration. In R, the keyword for continue is called `next`, in contrast to `continue` in Python. The difference between `break` and `continue` is that calling `break` would exit the innermost loop (when there are nested loops, only the innermost loop is affected); while calling `continue` would just skip the current iteration and continue the loop if not finished.

R

```
> for (i in 1:3){
+   print(i)
+   if (i==1) break
+ }
[1] 1
> for (i in 1:3){
+   if (i==2){next}
+   print(i)
+ }
[1] 1
[1] 3
```

Python

```
>>> for i in range(1,4):
...   print(i)
...   if i==1: break
...
1
>>> for i in range(1,4):
...   if i==2: continue
...   print(i)
...
1
3
```

1.5 SOME BUILT-IN DATA STRUCTURES

In the previous sections, we haven't seen much difference between R and Python. However, regarding the built-in data structures, there are some significant differences we will see in this section.

1.5.1 Vector in R and list in Python

In R, we can use function `c()` to create a vector; A vector is a sequence of elements with the same type. In Python, we can use `[]` to create a list, which is also a sequence of elements. But the elements in a list don't need to have the same type. To get the number of elements in a vector in R, we use the function `length()`; and to get the number of elements in a list in Python, we use the function `len()`.

```
> x=c(1,2,5,6)
> y=c('hello','world','!')
> x
[1] 1 2 5 6
> y
[1] "hello" "world" "!"
> length(x)
[1] 4
> z=c(1,'hello')
> z
[1] "1"     "hello"
```

```
>>> x=[1,2,5,6]
>>> y=['hello','world','!']
>>> x
[1, 2, 5, 6]
>>> y
['hello', 'world', '!']
>>> len(x)
4
>>> z=[1,'hello']
>>> z
[1, 'hello']
```

In the code snippet above, the first element in the variable `z` in R is coerced from 1 (numeric) to "1" (character) since the elements must have the same type.

To access a specific element from a vector or list, we could use `[]`. In R, sequence types are indexed beginning with the one subscript. In contrast, sequence types in Python are indexed beginning with the zero subscript.

```
> x=c(1,2,5,6)
> x[1]
[1] 1
```

Python

```
>>> x=[1,2,5,6]
>>> x[1]
2
>>> x[0]
1
```

What if the index to access is out of boundary?

R

```
> x=c(1,2,5,6)
> x[-1]
[1] 2 5 6
> x[0]
numeric(0)
> x[length(x)+1]
[1] NA
> length(numeric(0))
[1] 0
> length(NA)
[1] 1
```

Python

```
>>> x=[1,2,5,6]
>>> x[-1]
6
>>> x[len(x)+1]
Traceback (most recent call
    last):
  File "<stdin>", line 1,
     in <module>
IndexError:
    list index out of range
```

In Python, negative index number means indexing from the end of the list. Thus, `x[-1]` points to the last element and `x[-2]` points to the second-last element of the list. But R doesn't support indexing with negative numbers in the same way as Python. Specifically, in R `x[-index]` returns a new vector with `x[index]` excluded.

When we try to access with an index out of boundary, Python would throw an `IndexError`. The behavior of R when indexing out of boundary is more interesting. First, when we try to access `x[0]` in R we get a `numeric(0)` whose length is also 0. Since its length is 0, `numeric(0)` can be interpreted as an empty numeric vector. When we try to access `x[length(x)+1]` we get an `NA`. In R, there are also `NaN` and `NULL`.

`NaN` means "Not A Number" and it can be verified by checking its type - "double". `0/0` would result in an `NaN` in R. `NA` in R generally represents missing values. And `NULL` represents a NULL (empty) object. To check if a value is `NA`, `NaN` or `NULL`, we can use `is.na()`, `is.nan()` or `is.null`, respectively.

```
> typeof(NA)
[1] "logical"
> typeof(NaN)
[1] "double"
> typeof(NULL)
[1] "NULL"
> is.na(NA)
[1] TRUE
> is.null(NULL)
[1] TRUE
> is.nan(NaN)
```

Python

```
>>> type(None)
<class 'NoneType'>
>>> None is None
True
>>> 1 == None
False
```

In Python, there is no built-in `NA` or `NaN`. The counterpart of `NULL` in Python is

`None`. In Python, we can use the `is` keyword or == to check if a value is equal to `None`.

From the code snippet above, we also notice that in R the boolean type value is written as "TRUE/FALSE", compared with "True/False" in Python. Although in R "TRUE/FALSE" can also be abbreviated as "T/F", I don't recommend using the abbreviation.

There is one interesting fact that we can't add a `NULL` to a vector in R, but it is feasible to add a `None` to a list in Python.

```
> x=c(1,NA,NaN,NULL)
> x
[1]   1  NA NaN
> length(x)
[1] 3
```

```
>>> x=[1,None]
>>> x
[1, None]
>>> len(x)
2
```

Sometimes we want to create a vector/list with replicated elements, for example, a vector/list with all elements equal to 0.

```
> x=rep(0, 10)
> x
 [1] 0 0 0 0 0 0 0 0 0 0
> y=rep(c(0,1), 5)
> y
 [1] 0 1 0 1 0 1 0 1 0 1
```

```
>>> x=[0]*10
>>> x
[0, 0, 0, 0, 0, 0, 0, 0, 0,
    0]
>>> y=[0, 1]*5
>>> y
[0, 1, 0, 1, 0, 1, 0, 1, 0,
    1]
```

When we use the $*$ operator to make replicates of a list, there is one caveat - if the element inside the list is mutable then the replicated elements point to the same memory address. As a consequence, if one element is mutated other elements are also affected.

Python

```
>>> x=[0] # x is a list which is mutable
>>> y=[x]*5 # each element in y points to x
>>> y
[[0], [0], [0], [0], [0]]
>>> y[2]=2 # we point y[2] to 2 but x is not mutated
>>> y
[[0], [0], 2, [0], [0]]
>>> y[1][0]=-1 # we mutate x by changing y[1][0] from 0 to -1
```

```
>>> y
[[-1], [-1], 2, [-1], [-1]]
>>> x
[-1]
```

How to get a list with replicated elements but pointing to different memory addresses?

Python

```
>>> x=[0]
>>> y=[x[:] for _ in range(5)] # [:] makes a copy of the list x;
    another solution is [list(x) for _ in range(5)]
>>> y
[[0], [0], [0], [0], [0]]
>>> y[0][0]=2
>>> y
[[2], [0], [0], [0], [0]]
```

Besides accessing a specific element from a vector/list, we may also need to do slicing, i.e., to select a subset of the vector/list. There are two basic approaches of slicing:

- Integer-based

R

```
> x=c(1,2,3,4,5,6)
> x[2:4]
[1] 2 3 4
> x[c(1,2,5)] # a vector of indices
[1] 1 2 5
> x[seq(1,5,2)] # seq creates a vector to be used as
    indices
[1] 1 3 5
```

Python

```
>>> x=[1,2,3,4,5,6]
>>> x[1:4] # x[start:end] start is inclusive but end is
    exclusive
[2, 3, 4]
>>> x[0:5:2] # x[start:end:step]
[1, 3, 5]
```

The code snippet above uses hash character # for comments in both R and Python. Everything after # on the same line would be treated as comment (not executable). In the R code, we also used the function `seq()` to create a vector. When I see a function that I haven't seen before, I might either google it or use the built-in helper mechanism. Specifically, in R use `?` and in Python use `help()`.

```
> ?seq
```

```
>>> help(print)
```

- Condition-based

  Condition-based slicing means to select a subset of the elements which satisfy certain conditions. In R, it is quite straightforward by using a boolean vector whose length is the same as the vector to be sliced.

R

```
> x=c(1,2,5,5,6,6)
> x[x %% 2==1] # %% is the modulo operator in R; we select
    the odd elements
[1] 1 5 5
> x %% 2==1 # results in a boolean vector with the same
    length as x
[1]  TRUE FALSE  TRUE  TRUE FALSE FALSE
```

The condition-based slicing in Python is quite different from that in R. The prerequisite is list comprehension which provides a concise way to create new lists in Python. For example, let's create a list of squares of another list.

Python

```
>>> x=[1,2,5,5,6,6]
>>> [e**2 for e in x] # ** is the exponent operator, i.e.,
    x**y means x to the power of y
[1, 4, 25, 25, 36, 36]
```

We can also use `if` statement with list comprehension to filter a list to achieve list slicing.

Python

```
>>> x=[1,2,5,5,6,6]
>>> [e for e in x if e%2==1] # % is the modulo operator in
    Python
[1, 5, 5]
```

It is also common to use `if/else` with list comprehension to achieve more complex operations. For example, given a list x, let's create a new list y so that the non-negative elements in x are squared and the negative elements are replaced by 0s.

Python

```
>>> x=[1,-1,0,2,5,-3]
>>> [e**2 if e>=0 else 0 for e in x]
[1, 0, 0, 4, 25, 0]
```

The example above shows the power of list comprehension. To use `if` with list comprehension, the `if` statement should be placed in the end after the `for` loop statement; but to use `if/else` with list comprehension, the `if/else` statement should be placed before the `for` loop statement.

We can also modify the value of an element in a vector/list variable.

```
> x=c(1,2,3)
> x[1]=-1
> x
[1] -1  2  3
```

```
>>> x=[1,2,3]
>>> x[0]=-1
>>> x
[-1, 2, 3]
```

Two or multiple vectors/lists can be concatenated easily.

```
> x=c(1,2)
> y=c(3,4)
> z=c(5,6,7,8)
> c(x,y,z)
[1] 1 2 3 4 5 6 7 8
```

```
>>> x=[1,2]
>>> y=[3,4]
>>> z=[5,6,7,8]
>>> x+y+z
[1, 2, 3, 4, 5, 6, 7, 8]
```

As the list structure in Python is mutable, there are many things we can do with list.

Python

```
>>> x=[1,2,3]
>>> x.append(4) # append a single value to the list x
>>> x
[1, 2, 3, 4]
>>> y=[5,6]
>>> x.extend(y) # extend list y to x
>>> x
[1, 2, 3, 4, 5, 6]
>>> last=x.pop() # pop the last element from x
>>> last
6
>>> x
[1, 2, 3, 4, 5]
```

I like the list structure in Python much more than the vector structure in R. list in Python has a lot more useful features which can be found from the python official documentation[4].

1.5.2 Array

Array is one of the most important data structures in scientific programming. In R, there is also an object type "matrix", but according to my own experience, we can almost ignore its existence and use array instead. We can definitely use list as array in Python, but lots of linear algebra operations are not supported for the list type. Fortunately, there is a Python package `numpy` off the shelf.

R

```
> x=1:12
> array1=array(x, c(4,3)) # convert vector x to a 4 rows * 3
    cols array
> array1
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> y=1:6
> array2=array(y, c(3,2)) # convert vector y to a 3 rows * 2
    cols array
> array2
     [,1] [,2]
```

[4] https://docs.python.org/3/tutorial/datastructures.html

```
[1,]    1    4
[2,]    2    5
[3,]    3    6
> array3 = array1 %*% array2 # %*% is the matrix multiplication
    operator
> array3
     [,1] [,2]
[1,]   38   83
[2,]   44   98
[3,]   50  113
[4,]   56  128
> dim(array3) # get the dimension of array3
[1] 4 2
```

Python

```
>>> import numpy as np # we import the numpy module and alias it
    as np
>>> array1=np.reshape(list(range(1,13)),(4,3)) # convert a list
    to a 2d np.array
>>> array1
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
>>> type(array1)
<class 'numpy.ndarray'>
>>> array2=np.reshape(list(range(1,7)),(3,2))
>>> array2
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> array3=np.dot(array1, array2) # matrix multiplication using
    np.dot()
>>> array3
array([[ 22,  28],
       [ 49,  64],
       [ 76, 100],
       [103, 136]])
>>> array3.shape # get the shape(dimension) of array3
(4, 2)
```

You may have noticed that the results of the R code snippet and Python code snippet are different. The reason is that in R the conversion from a vector to an array

is by-column; but in `numpy` the reshape from a list to a 2D `numpy.array` is by-row. There are two ways to reshape a list to a 2D `numpy.array` by column.

Python

```
>>> array1=np.reshape(list(range(1,13)),(4,3),order='F') # use
    order='F'
>>> array1
array([[ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11],
       [ 4,  8, 12]])
>>> array2=np.reshape(list(range(1,7)),(2,3)).T # use .T to
    transpose an array
>>> array2
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> np.dot(array1, array2) # now we get the same result as using
     R
array([[ 38,  83],
       [ 44,  98],
       [ 50, 113],
       [ 56, 128]])
```

To learn more about `numpy`, the official website[5] has great documentation/tutorials.

The term broadcasting describes how arrays with different shapes are handled during arithmetic operations. A simple example of broadcasting is given below.

R

```
> x = c(1, 2, 3)
> x+1
[1] 2 3 4
```

Python

```
>>> import numpy as np
>>> x = np.array([1, 2, 3])
>>> x + 1
array([2, 3, 4])
```

However, the broadcasting rules in R and Python are not exactly the same.

R

```
> x = array(c(1:6), c(3,2))
> y = c(1, 2, 3)
> z = c(1, 2)
```

Python

```
>>> import numpy as np
>>> x = np.array([[1, 2],
    [3, 4], [5, 6]])
```

[5] http://www.numpy.org

```
# point-wise multiplication
> x * y
     [,1] [,2]
[1,]    1    4
[2,]    4   10
[3,]    9   18
> x*z
     [,1] [,2]
[1,]    1    8
[2,]    4    5
[3,]    3   12
```

```
>>> y = np.array([1, 2, 3])
>>> z = np.array([1, 2])
>>> # point-wise
    multiplication
>>> x * y
Traceback (most recent call
    last):
  File "<stdin>", line 1,
     in <module>
ValueError: operands could
    not be broadcast together
     with shapes (3,2) (3,)
>>> x * z
array([[ 1,  4],
       [ 3,  8],
       [ 5, 12]])
```

From the R code, we see the broadcasting in R is like recycling along with the column. In Python, when the two arrays have different dimensions, the one with fewer dimensions is padded with ones on its leading side. According to this rule, when we do $x * y$, the dimension of x is $(3, 2)$ but the dimension of y is 3. Thus, the dimension of y is padded to $(1, 3)$, which explains what happens when $x * y$.

1.5.3 List in R and dictionary in Python

Yes, in R there is also an object type called list. The major difference between a vector and a list in R is that a list could contain different types of elements. list in R supports integer-based accessing using `[[]]` (compared to `[]` for vector).

R

```
> x=list(1,'hello world!')
> x
[[1]]
[1] 1

[[2]]
[1] "hello world!"

> x[[1]]
[1] 1
> x[[2]]
[1] "hello world!"
> length(x)
[1] 2
```

list in R could be named and support accessing by name via either `[[]]` or `$` operator. But vector in R can also be named and support accessing by name.

R

```
> x=c('a'=1,'b'=2)
> names(x)
[1] "a" "b"
> x['b']
b
2
> l=list('a'=1,'b'=2)
> l[['b']]
[1] 2
> l$b
[1] 2
> names(l)
[1] "a" "b"
```

However, elements in list in Python can't be named as R. If we need the feature of accessing by name in Python, we can use the dictionary structure. If you used Java before, you may consider dictionary in Python as the counterpart of HashMap in Java. Essentially, a dictionary in Python is a collection of key:value pairs.

Python

```
>>> x={'a':1,'b':2} # {key:value} pairs
>>> x
{'a': 1, 'b': 2}
>>> x['a']
1
>>> x['b']
2
>>> len(x) # number of key:value pairs
2
>>> x.pop('a') # remove the key 'a' and we get its value 1
1
>>> x
{'b': 2}
```

Unlike dictionary in Python, list in R doesn't support the `pop()` operation. Thus, in order to modify a list in R, a new one must be created explicitly or implicitly.

1.5.4 data.frame, data.table and pandas

data.frame is a built-in type in R for data manipulation. In Python, there is no such built-in data structure since Python is a more general-purpose programming language. The solution for data.frame in Python is the `pandas`[6] module.

Before we dive into data.frame, you may be curious as to why we need it. In other words, why can't we just use vector, list, array/matrix and dictionary for all data manipulation tasks? I would say yes - data.frame is not a must-have feature for most of ETL (extraction, transformation and Load) operations. But data.frame provides a very intuitive way for us to understand the structured dataset. A data.frame is usually flat with 2 dimensions, i.e., row and column. The row dimension is across multiple observations and the column dimension is across multiple attributes/features. If you are familiar with relational database, a data.frame can be viewed as a table.

Let's see an example of using data.frame to represent employees' information in a company.

R

```
> employee_df = data.frame(name=c("A", "B", "C"),department=c("Engineering","Operations","Sales"))
> employee_df
  name  department
1    A Engineering
2    B  Operations
3    C       Sales
```

Python

```
>>> import pandas as pd
>>> employee_df=pd.DataFrame({'name':['A','B','C'],'department':["Engineering","Operations","Sales"]})
>>> employee_df
  name   department
0    A  Engineering
1    B   Operations
2    C        Sales
```

There are quite a few ways to create data.frame. The most commonly used one is to create data.frame object from array/matrix. We may also need to convert a numeric data.frame to an array/matrix.

R

```
> x=array(rnorm(12), c(3,4))
> x
           [,1]       [,2]      [,3]       [,4]
[1,] -0.8101246 -0.8594136 -2.260810  0.5727590
[2,] -0.9175476  0.1345982  1.067628 -0.7643533
[3,]  0.7865971 -1.9046711 -0.154928 -0.6807527
> random_df=as.data.frame(x)
> random_df
          V1         V2        V3         V4
1 -0.8101246 -0.8594136 -2.260810  0.5727590
```

[6] https://pandas.pydata.org/

```
2 −0.9175476  0.1345982  1.067628 −0.7643533
3  0.7865971 −1.9046711 −0.154928 −0.6807527
> data.matrix(random_df)
              V1         V2        V3         V4
[1,] −0.8101246 −0.8594136 −2.260810  0.5727590
[2,] −0.9175476  0.1345982  1.067628 −0.7643533
[3,]  0.7865971 −1.9046711 −0.154928 −0.6807527
```

Python

```
>>> import numpy as np
>>> import pandas as pd
>>> x=np.random.normal(size=(3,4))
>>> x
array([[−0.54164878, −0.14285267, −0.39835535, −0.81522719],
       [ 0.01540508,  0.63556266,  0.16800583,  0.17594448],
       [−1.21598262,  0.52860817, −0.61757696,  0.18445057]])
>>> random_df = pd.DataFrame(x)
>>> random_df
          0         1         2         3
0 −0.541649 −0.142853 −0.398355 −0.815227
1  0.015405  0.635563  0.168006  0.175944
2 −1.215983  0.528608 −0.617577  0.184451
>>> np.asarray(random_df)
array([[−0.54164878, −0.14285267, −0.39835535, −0.81522719],
       [ 0.01540508,  0.63556266,  0.16800583,  0.17594448],
       [−1.21598262,  0.52860817, −0.61757696,  0.18445057]])
```

In general, operations on an array/matrix are much faster than those on a data frame. In R, we may use the built-in function `data.matrix` to convert a data.frame to an array/matrix. In Python, we could use the function `asarray` in `numpy` module.

Although data.frame is a built-in type, it is not quite efficient for many operations. I would suggest using data.table[7] whenever possible. `dplyr`[8] is also a very popular package in R for data manipulation. Many good resources are available online to learn data.table and pandas.

1.6 REVISIT OF VARIABLES

We have talked about variables and functions so far. When a function has a name, its name is also a valid variable. After all, what is a variable?

In mathematics, a variable is a symbol that represents an element, and we do

[7]https://cran.r-project.org/web/packages/data.table/index.html
[8]https://dplyr.tidyverse.org/

not care whether we conceptualize a variable in our mind, or write it down on paper. However, in programming a variable is not only a symbol. We have to understand that a variable is a name given to a memory location in computer systems. When we run `x = 2` in R or Python, somewhere in memory has the value 2, and the variable (name) points to this memory address. If we further run `y = x`, the variable `y` points to the same memory location pointed to by (x). What if we run `x=3`? It doesn't modify the memory which stores the value 2. Instead, somewhere in the memory now has the value 3 and this memory location has a name `x`. And the variable `y` is not affected at all, as well as the memory location it points to.

1.6.1 Mutability

Almost everything in R or Python is an object, including these data structures we introduced in previous sections. Mutability is a property of objects, not variables, because a variable is just a name.

A list in Python is mutable, meaning that we could change the elements stored in the list object without copying the list object from one memory location to another. We can use the `id` function in Python to check the memory location for a variable. In the code below, we modified the first element of the list object with name `x`. And since Python list is mutable, the memory address of the list doesn't change.

Python

```
>>> x=list(range(1,1001)) # list() convert a range object to a
    list
>>> hex(id(x)) # print the memory address of x
'0x10592d908'
>>> x[0]=1.0 # from integer to float
>>> hex(id(x))
'0x10592d908'
```

Is there any immutable data structure in Python? Yes, for example tuple is immutable, which contains a sequence of elements. The element accessing and subset slicing of tuple follows the same rules as list in Python.

Python

```
>>> x=(1,2,3,) # use () to create a tuple in Python, it is
    better to always put a comma in the end
>>> type(x)
<class 'tuple'>
>>> len(x)
3
>>> x[0]
1
>>> x[0]=-1
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

If we have two Python variables pointed to the same memory, when we modify the memory via one variable the other is also affected as we expect (see the example below).

Python

```
>>> x=[1,2,3]
>>> id(x)
4535423616
>>> x[0]=0
>>> x=[1,2,3]
>>> y=x
>>> id(x)
4535459104
>>> id(y)
4535459104
>>> x[0]=0
>>> id(x)
4535459104
>>> id(y)
4535459104
>>> x
[0, 2, 3]
>>> y
[0, 2, 3]
```

In contrast, the mutability of vector in R is more complex and sometimes confusing. First, let's see the behavior when there is a single name given to the vector object stored in memory.

R

```
> a=c(1,2,3)
> .Internal(inspect(a))
@7fe94408f3c8 14 REALSXP g0c3 [NAM(1)] (len=3, tl=0) 1,2,3
> a[1]=0
> .Internal(inspect(a))
@7fe94408f3c8 14 REALSXP g0c3 [NAM(1)] (len=3, tl=0) 0,2,3
```

It is clear in this case the vector object is mutable since the memory address

doesn't change after the modification. What if there is an additional name given to the memory?

R

```
> a=c(1,2,3)
> b=a
> .Internal(inspect(a))
@7fe94408f238 14 REALSXP g0c3 [NAM(2)] (len=3, tl=0) 1,2,3
> .Internal(inspect(b))
@7fe94408f238 14 REALSXP g0c3 [NAM(2)] (len=3, tl=0) 1,2,3
> a[1]=0
> .Internal(inspect(a))
@7fe94408f0a8 14 REALSXP g0c3 [NAM(1)] (len=3, tl=0) 0,2,3
> .Internal(inspect(b))
@7fe94408f238 14 REALSXP g0c3 [NAM(2)] (len=3, tl=0) 1,2,3
> a
[1] 0 2 3
> b
[1] 1 2 3
```

Before the modification, both variable `a` and `b` point to the same vector object in the memory. But surprisingly, after the modification the memory address of variable `a` also changed, which is called "copy on modify" in R. And because of this unique behavior, the modification of `a` doesn't affect the object stored in the old memory and thus the vector object is immutable in this case. The mutability of R `list` is similar to that of R `vector`.

1.6.2 Variable as function argument

Most functions/methods in R and Python take some variables as argument. What happens when we pass the variables into a function?

In Python, the variable, i.e., the name of the object, is passed into a function. If the variable points to an immutable object, any modification to the variable, i.e., the name, doesn't persist. However, when the variable points to a mutable object, the modification of the object stored in memory persists. Let's see the examples below.

```
>>> def g(x):
...     print(id(x))
...     x-=1
...     print(id(x))
...     print(x)
>>> a=1
>>> id(a)
```

```
>>> def f(x):
...     id(x)
...     x[0]-=1
...     id(x)
>>> a=[1,2,3]
>>> id(a)
4535423616
```

```
4531658512
>>> g(a)
4531658512
4531658480
0
>>> a
1
```

```
>>> f(a)
4535423616
4535423616
>>> a
[0, 2, 3]
```

We see that the object is passed into function by its name. If the object is immutable, a new copy is created in memory when any modification is made to the original object. When the object is immutable, no new copy is made and thus the change persists out of the function.

In R, the passed object is always copied on a modification inside the function, and thus no modification can be made to the original object in memory.

R

```
> f=function(x){
+   print(.Internal(inspect(x)))
+   x[1]=x[1]-1
+   print(.Internal(inspect(x)))
+   print(x)
+ }
>
> a=c(1,2,3)
> .Internal(inspect(a))
@7fe945538688 14 REALSXP g0c3 [NAM(1)] (len=3, tl=0) 1,2,3
> f(a)
@7fe945538688 14 REALSXP g0c3 [NAM(3)] (len=3, tl=0) 1,2,3
[1] 1 2 3
@7fe945538598 14 REALSXP g0c3 [NAM(1)] (len=3, tl=0) 0,2,3
[1] 0 2 3
[1] 0 2 3
> a
[1] 1 2 3
```

People may argue that R functions are not as flexible as Python functions. However, it makes more sense to do functional programming in R since we usually can't modify objects passed into a function.

1.6.3 Scope of variables

What is the scope of a variable and why does it matter? Let's first have a look at the code snippets below.

R

```
> x=1
> var_func_1 =
    function(){print(x)}
> var_func_1()
[1] 1
> var_func_2 = function(){x=
    x+1; print(x)}
> var_func_2()
[1] 2
> x
[1] 1
```

Python

```
>>> x=1
>>>
   def var_func_1():print(x)
>>> var_func_1()
1
>>> def var_func_2():x+=1
...
>>> var_func_2()
Traceback (most recent call
   last):
  File "<stdin>", line 1,
     in <module>
  File "<stdin>", line 1,
     in var_func_2
UnboundLocalError: local
   variable 'x
   ' referenced before
   assignment
```

The results of the code above seem strange before knowing the concept of variable scope. Inside a function, a variable may refer to a function argument/parameter or it could be formally declared inside the function which is called a local variable. But in the code above, `x` is neither a function argument nor a local variable. How does the `print()` function know what the identifier `x` points to?

The scope of a variable determines where the variable is available/accessible (can be referenced). Both R and Python apply lexical/static scoping for variables, which set the scope of a variable based on the structure of the program. In static scoping, when an 'unknown' variable is referenced, the function will try to find it from the most closely enclosing block. That explains how the `print()` function could find the variable `x`.

In the R code above, `x=x+1` the first `x` is a local variable created by the = operator; the second `x` is referenced inside the function so the static scoping rule applies. As a result, a local variable `x` which is equal to 2 is created, which is independent with the `x` outside of the function `var_func_2()`. However, in Python when a variable is assigned a value in a statement, the variable would be treated as a local variable and that explains the `UnboundLocalError`.

Is it possible to change a variable inside a function which is declared outside the function without passing it as an argument? Based on the static scoping rule only, it's impossible. But there are workarounds in both R/Python. In R, we need the help of environment; and in Python we can use the keyword `global`.

So what is an environment in R? An environment is a place where objects are stored. When we invoke the interactive R session, an environment named `.GlobalEnv` is created automatically. We can also use the function `environment()` to get the

present environment. The `ls()` function can take an environment as the argument to list all objects inside the environment.

R

```
$r
> typeof(.GlobalEnv)
[1] "environment"
> environment()
<environment: R_GlobalEnv>
> x=1
> ls(environment())
[1] "x"
> env_func_1=function(x){
+   y=x+1
+   print(environment())
+   ls(environment())
+ }
> env_func_1(2)
<environment: 0x7fc59d165a20>
[1] "x" "y"
> env_func_2=function(){print(environment())}
> env_func_2()
<environment: 0x7fc59d16f520>
```

The above code shows that each function has its own environment containing all function arguments and local variables declared inside the function. In order to change a variable declared outside of a function, we need the access of the environment enclosing the variable to change. There is a function `parent_env(e)` that returns the parent environment of the given environment `e` in R. Using this function, we are able to change the value of `x` declared in `.GlobalEnv` inside a function which is also declared in `.GlobalEnv`. The `global` keyword in Python works in a totally different way, which is simple but less flexible.

```
> x=1
> env_func_3=function(){
+   cur_env=environment()
+   par_env=
    parent.env(cur_env)
+   par_env$x=2
+ }
> env_func_3()
> x
[1] 2
```

Python

```
>>> def env_func_3():
...   global x
...   x = 2
...
>>> x=1
>>> env_func_3()
>>> x
2
```

I seldom use the `global` keyword in Python, if ever. But the environment in R could be very handy on some occasions. In R, environment could be used as a purely mutable version of the `list` data structure. Let's use the R function `tracemem` to trace the copy of an object. It is worth nothing that `tracemem` can't trace R functions.

```
# list is not purely mutable
> x=list(1)
> tracemem(x)
[1] "<0x7f829183f6f8>"
> x$a=2
> tracemem(x)
[1] "<0x7f828f4d05c8>"
```

```
# environment is purely
    mutable
> x=new.env()
> x
<environment: 0x7f8290aee7e8
    >
> x$a=2
> x
<environment: 0x7f8290aee7e8
    >
```

Actually, the object of an R6 class type is also an environment.

R

```
> # load the Complex class that we defined in chapter 1
> x = Complex$new(1,2)
> typeof(x)
[1] "environment"
```

In Python, we can assign values to multiple variables in one line.

```
>>> x,y = 1,2
>>> x
1
>>> y
2
```

```
>>> x,y=(1,2)
>>> print(x,y)
1 2
>>> (x,y)=(1,2)
>>> print(x,y)
1 2
>>> [x,y]=(1,2)
>>> print(x,y)
1 2
```

Even though in the left snippet above there aren't parentheses embracing `1, 2` after the `=` operator, a tuple is created first and then the tuple is unpacked and assigned to `x, y`. Such a mechanism doesn't exist in R, but we can define our own multiple assignment operator with the help of environment.

R chapter1/multi_assignment.R

```r
`%=%` = function(left, right) {
  # we require the RHS to be a list strictly
  stopifnot(is.list(right))
  # dest_env is the desitination environment enclosing the
     variables on LHS
  dest_env = parent.env(environment())
  left = substitute(left)

  recursive_assign = function(left, right, dest_env) {
    if (length(left) == 1) {
      assign(x = deparse(left),
             value = right,
             envir = dest_env)
      return()
    }
    if (length(left) != length(right) + 1) {
      stop("LHS and RHS must have the same shapes")
    }

    for (i in 2:length(left)) {
      recursive_assign(left[[i]], right[[i - 1]],  dest_env)
    }
  }

  recursive_assign(left, right, dest_env)
}
```

Before going more deeply into the script, first let's see the usage of the multiple assignment operator we defined.

R

```
> source('multi_assignment.R')
> c(x,y,z) %=% list(1,"Hello World!",c(2,3))
> x
[1] 1
> y
[1] "Hello World!"
> z
[1] 2 3
> list(a,b) %=% list(1,as.Date('2019-01-01'))
> a
[1] 1
> b
```

```
13 [1] "2019-01-01"
```

In the `%=%` operator defined above, we used two functions `substitute, deparse` which are very powerful but less known by R novices. To better understand these functions as well as some other less known R functions, the Rchaeology[9] tutorial is worth reading.

It is also interesting to see that we defined the function `recursive_assign` inside the `%=%` function. Both R and Python support the concept of first class functions. More specifically, a function in R/Python is an object, which can be

1. stored as a variable;
2. passed as a function argument;
3. returned from a function.

The essential idea behind the `recursive_assign` function is a depth-first search (DFS), which is a fundamental graph traversing algorithm[10]. In the context of the `recursive_assign` function, we use DFS to traverse the parse tree of the `left` argument created by calling `substitute(left)`.

1.7 OBJECT-ORIENTED PROGRAMMING (OOP) IN R/PYTHON

All the codes we wrote above follow the procedural programming paradigm[11]. We can also do functional programming (FP) and OOP in R/Python. In this section, let's focus on OOP in R/Python.

Class is the key concept in OOP. In R there are two commonly used built-in systems to define classes, i.e., S3 and S4. In addition, there is an external package R6[12] which defines R6 classes. S3 is a light-weight system but its style is quite different from OOP in many other programming languages. S4 system follows the principles of modern object-oriented programming much better than S3. However, the usage of S4 classes is quite tedious. I would ignore S3/S4 and introduce R6, which is closer to the class in Python.

Let's build a class in R/Python to represent complex numbers.

R

```
1 > library(R6) # load the R6 package
2 >
3 > Complex = R6Class("Complex",
4 + public = list( # only elements declared in this list are
      accessible by the object of this class
```

[9] https://cran.r-project.org/web/packages/rockchalk/vignettes/Rchaeology.pdf
[10] https://en.wikipedia.org/wiki/Depth-first_search
[11] https://en.wikipedia.org/wiki/Comparison_of_programming_paradigms
[12] https://cran.r-project.org/web/packages/R6/index.html

```
+ real = NULL,
+ imag = NULL,
+ # the initialize function would be called automatically when
    we create an object of the class
+ initialize = function(real, imag){
+     # call functions to change real and imag values
+   self$set_real(real)
+   self$set_imag(imag)
+ },
+ # define a function to change the real value
+ set_real = function(real){
+   self$real=real
+ },
+ # define a function to change the imag value
+ set_imag = function(imag){
+   self$imag = imag
+ },
+ # override print function
+ print = function(){
+   cat(paste0(as.character(self$real),'+',as.character(self$
    imag),'j'),'\n')
+ }
+ )
+ )
> # let's create a complex number object based on the Complex
    class we defined above using the new function
> x = Complex$new(1,2)
> x
1+2j
> x$real # the public attributes of x could be accessed by $
    operator
[1] 1
```

Python

```
>>> class Complex:
...   # the __init__ function would be called automatically when
     we create an object of the class
...   def __init__(self, real, imag):
...     self.real = real
...     self.imag = imag
...
...   def __repr__(self):
...     return "{0}+{1}j".format(self.real, self.imag)
```

```
...
>>> x = Complex(1,2)
>>> x
1+2j
>>> x.real # different from the $ operator in R, here we use .
    to access the attribute of an object
1
```

By overriding the print function in the R6 class, we can have the object printed in the format of `real+imag j`. To achieve the same effect in Python, we override the method `__repr__`. In Python, we refer to the functions defined in classes as methods. And overriding a method means changing the implementation of a method provided by one of its ancestors. To understand the concept of ancestors in OOP, one needs to understand the concept of inheritance[13].

You may be curious about the double underscore surrounding the methods, such as `__init__` and `__repr__`. These methods are well-known as special methods[14]. In the definition of the special method `__repr__` in the Python code, the `format` method of `str` object[15] is used.

Special methods can be very handy if we use them in suitable cases. For example, we can use the special method `__add__` to implement the `+` operator for the `Complex` class we defined above.

Python

```
>>> class Complex:
...   def __init__(self, real, imag):
...     self.real = real
...     self.imag = imag
...   def __repr__(self):
...     return "{0}+{1}j".format(self.real, self.imag)
...   def __add__(self, another):
...     return Complex(self.real + another.real, self.imag +
    another.imag)
...
>>> x = Complex(1,2)
>>> y = Complex(2,4)
>>> x+y # + operator works now
3+6j
```

We can also implement the `+` operator for `Complex` class in R as we have done for Python.

[13]https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)
[14]https://docs.python.org/3/reference/datamodel.html#specialnames
[15]https://docs.python.org/3.7/library/string.html

R

```
> `+.Complex` = function(x, y){
+   Complex$new(x$real+y$real, x$imag+y$imag)
+ }
> x=Complex$new(1,2)
> y=Complex$new(2,4)
> x+y
3+6j
```

The most interesting part of the code above is `` `+.Complex` ``. First, why do we use `` ` `` to quote the function name? Before getting into this question, let's have a look at the Python 3's variable naming rules[16].

```
Within the ASCII range (U+0001..U+007F), the valid characters
    for identifiers (also referred to as names) are the same as
    in Python 2.x: the uppercase and lowercase letters A through
     Z, the underscore _ and, except for the first character,
    the digits 0 through 9.
```

According to the rule, we can't declare a variable with name `2x`. Compared with Python, in R we can also use `.` in the variable names[17]. However, there is a workaround to use invalid variable names in R with the help of `` ` ``.

R

```
> 2x = 5
Error: unexpected symbol in
    "2x"
> .x = 3
> .x
[1] 3
> `+2x%` = 0
> `+2x%`
[1] 0
```

Python

```
>>> 2x = 5
  File "<stdin>", line 1
    2x = 5
     ^
SyntaxError: invalid syntax
>>> .x = 3
  File "<stdin>", line 1
    .x = 3
    ^
SyntaxError: invalid syntax
```

Now it is clear the usage of `` ` `` in `` `+.Complex` `` is to define a function with invalid name. Placing `.Complex` after `+` is related to S3 method dispatching, which will not be discussed here.

[16] https://docs.python.org/3.3/reference/lexical_analysis.html
[17] https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Identifiers

1.7.1 Member accessibility in Python

In some programming languages the members (variable or methods) of a class can be declared with access modifiers which specify the accessibility or scope of a member. In Python, class members don't have explicit access modifiers, but it is still possible to specify the accessibility. By default, the class member can be accessed inside or outside the class definition. If the name of the member starts with a single underscore, the member should not be accessed outside the class definition by convention, but it is not enforced. If the name of the member starts with double underscore, the member name is mangled and thus the member can't be accessed by its original member outside the class definition. But inside the class definition these variables can always be accessed. Let's see the example below.

Python

```
>>> class ProtectedMemberClass:
...     """
...     _x and _func1 are protected and are recommended not to be accessed out of the class definition, but not enforced.
...     __y and __func2 are private and the names would be mangled
...     """
...     def __init__(self, val1, val2):
...         self._x = val1
...         self.__y = val2
...     def _func1(self):
...         print("protected _func1 called")
...     def __func2(self):
...         print("private __func2 called")
...     def func3(self):
...         # inside the class definition, we can access all these members
...         print("self._x is {0} and self.__y is{1}".format(
...             self._x, self.__y))
...         self._func1()
...         self.__func2()
...
>>> p = ProtectedMemberClass(0, 1)
>>> p._x
0
>>> p.__y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'ProtectedMemberClass' object has no attribute '__y'
>>> p._func1()
```

```
protected _func1 called
>>> p.__func2()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'ProtectedMemberClass' object has no attribute '
    __func2'
>>> p.func3()
self._x is 0 and self.__y is1
protected _func1 called
private __func2 called
```

In this example, an error is thrown when we try to access `__y` or `__func2` outside the class definition. But they are reachable within the class definition and these fields are usually called private fields.

1.8 MISCELLANEOUS

There are some items that I haven't discussed so far, which are also important in order to master R/Python.

1.8.1 Package/module installation

- Use `install.packages()` function in R
- Use R IDE to install packages
- Use `pip`[18] to install modules in Python

1.8.2 Virtual environment

Virtual environment is a tool to manage dependencies in Python. There are different ways to create virtual environments in Python. I suggest using the `venv` module shipped with Python 3. Unfortunately, there is nothing like a real virtual environment in R as far as I know although there are quite a few management tools/packages.

1.8.3 <- vs. =

If you have known R before, you probably have heard of the advice[19] to use <− to rather than = for value assignment. Let's see an example when <− makes a difference when we do value assignment.

```
> x=1
> a=list(x <- 2)
```

```
> x=1
> a=list(x = 2)
```

[18] https://packaging.python.org/tutorials/installing-packages/
[19] https://google.github.io/styleguide/Rguide.xml

```
> a
[[1]]
[1] 2

> x
[1] 2
```

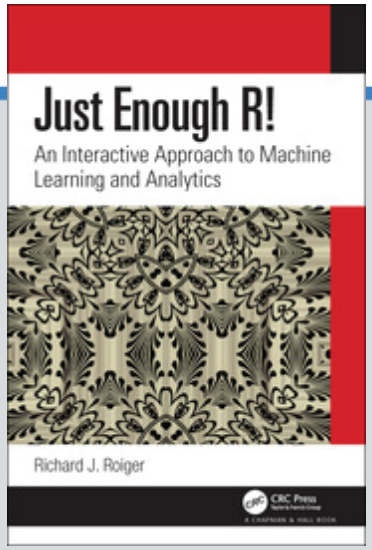
```
> a
$x
[1] 2

> x
[1] 1
```

CHAPTER

2

PREPARING THE DATA



Just Enough R!

An Interactive Approach to Machine Learning and Analytics

By Richard J. Roiger

Learn more

CHAPTER 4

Preparing the Data

In This Chapter

- The KDD Process Model
- Relational Databases
- Data Preprocessing Techniques
- Data Transformation Methods

Many of the packages available thru the Comprehensive R Archive Network (CRAN) repository include datasets which have been at least partially preprocessed. This allows us to concentrate on learning about the various machine learning tools without concerning ourselves with the preprocessing tasks needed to analyze real-world data. However, real data often contains missing values, noise, and requires one or several transformations before it is ready for the model building process. Therefore, before turning our attention to the machine learning algorithms presented in the next chapters, we examine solution strategies for the preprocessing issues often seen with real data.

In Section 4.1, we introduce a formal seven-step process model for knowledge discovery. In Sections 4.2–4.4, we concentrate on the steps involving the creation of initial target data, data preprocessing, and transformation as they are by far the most difficult and time-consuming parts of this process. When careful attention is paid to data preprocessing and data transformation, our chances of success in building useful machine learning models significantly increase.

4.1 A PROCESS MODEL FOR KNOWLEDGE DISCOVERY

Knowledge discovery in data (KDD) is an interactive, iterative procedure that attempts to extract implicit, previously unknown, and potentially useful knowledge from data. Several variations of the KDD process model exist. Variations describe the KDD process from 4

to as many as 12 steps. Although the number of steps may differ, most descriptions show consistency in content. Here is a brief description of a seven-step KDD process model:

1. *Goal identification*. The focus of this step is on understanding the domain being considered for knowledge discovery. We write a clear statement about what is to be accomplished. A hypothesis offering a likely or desired outcome can be stated.
2. *Creating a target dataset*. With the help of one or more human experts and knowledge discovery tools, we choose an initial set of data to be analyzed.
3. *Data preprocessing*. We use available resources to deal with noisy data. We decide what to do about missing data values and how to account for time-sequence information.
4. *Data transformation*. Attributes and instances are added and/or eliminated from the target data. We decide on methods to normalize, convert, and smooth data.
5. *Data mining*. A best model for representing the data is created by applying one or more machine learning algorithms.
6. *Interpretation and evaluation*. We examine the output from step 5 to determine if what has been discovered is both useful and interesting. Decisions are made about whether to repeat previous steps using new attributes and/or instances.
7. *Taking action*. If the discovered knowledge is deemed useful, the knowledge is incorporated and applied directly to appropriate problems.

As you work through the remaining sections of this chapter, we point out several R functions that implement the preprocessing techniques described here. These functions are used to help solve problems throughout the remaining chapters of your text. It is well worth your time to further investigate these functions prior to moving on to Chapters 5–12.

4.2 CREATING A TARGET DATASET

A viable set of resource data is of primary importance for any analytics project to succeed. Target data is commonly extracted from three primary sources—a data warehouse, one or more transactional databases, or one or several flat files. Many machine learning tools require input data to be in a flat file or spreadsheet format (i.e., R's data frame). If the original data is housed in a flat file, creating the initial target data is straightforward. Let's examine the other possibilities.

Database management systems (DBMS) store and manipulate transactional data. The computer programs in a DBMS are able to quickly update and retrieve information from a stored database. The data in a DBMS is often structured using the relational model. A *relational database* represents data as a collection of tables containing rows and columns. Each column of a table is known as an attribute, and each row of the table stores information about one data record. The individual rows are called *tuples*. All tuples in a relational table are uniquely identified by a combination of one or more table attributes.

A main goal of the relational model is to reduce data redundancy so as to allow for quick access to information in the database. A set of normal forms that discourage data redundancy define formatting rules for relational tables. If a relational table contains redundant data, the redundancy is removed by decomposing the table into two or more relational structures. In contrast, the goal of knowledge discovery is to uncover the inherent redundancy in data. Therefore, one or more relational join operations are usually required to restructure data into a form amenable for data mining.

To see this, consider the hypothetical credit card promotion database we defined in Table 2.2 of Chapter 2. Recall the table attributes: *income range*, *magazine promotion*, *watch promotion*, *life insurance promotion*, *credit card insurance*, *gender*, and *age*. The data in Table 2.2 is not a database at all but represents a flat file structure extracted from a database such as the one shown in Figure 4.1. The Acme credit card database contains tables about credit card billing information and orders, in addition to information about credit card promotions. The Promotion-C table creates two one-to-many relationships to resolve the single many-to-many relationship between Customer and Promotion. Therefore, the promotional information shown in Table 2.2 is housed in several relational tables within the database. The next section looks the work needed to extract information from the relational database of Figure 4.1 in order to create a data frame structure similar to Table 2.2.

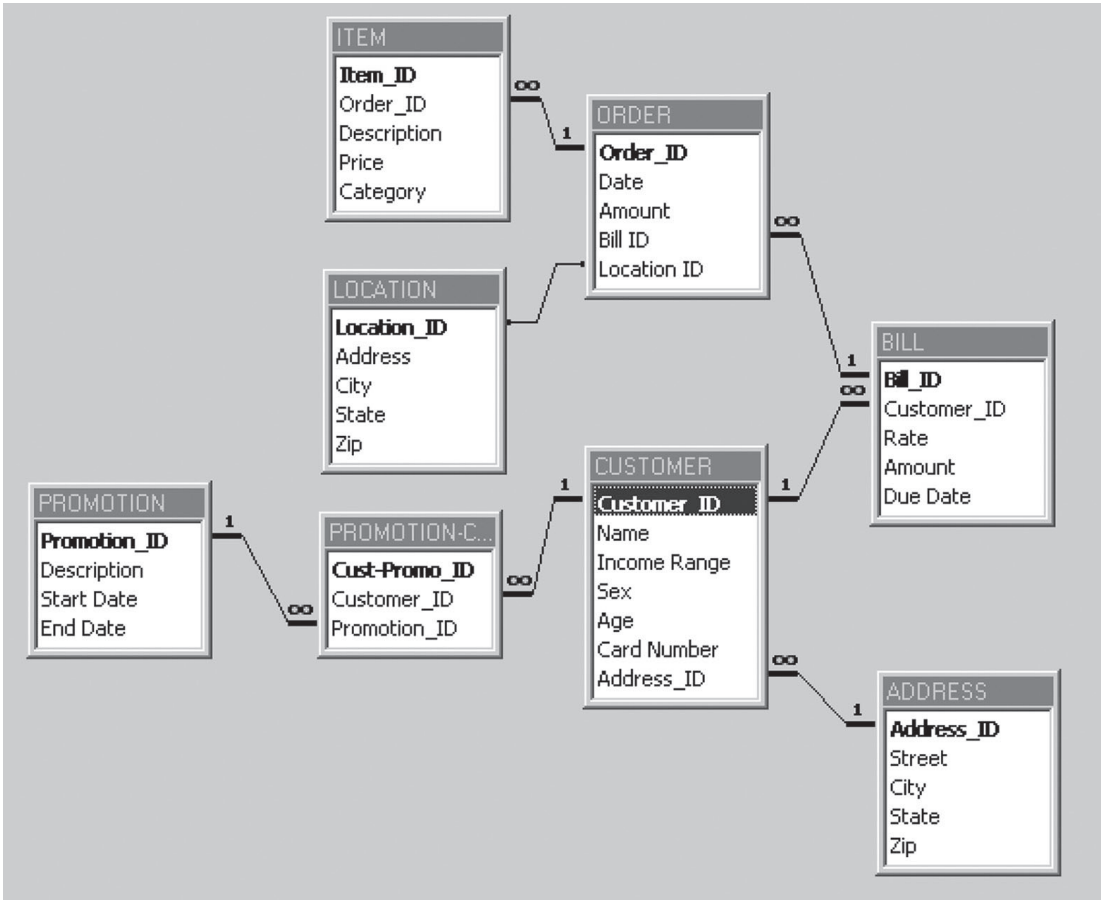


FIGURE 4.1 The Acme credit card database.

4.2.1 Interfacing R with the Relational Model

We have more than one option for connecting R to a relational database. For our example, we chose SQLite for the DBMS and R's DBI package for the database interface. We selected SQLite as it is highly portable, easy to install and use, provides the features of an SQL engine, and most importantly doesn't require a server! In addition, SQLite can be accessed using command line prompts or with a database browser.

The URL for SQLite's home page is https://sqlite.org/index.html. Click *download*, and then select the appropriate download link. For windows users, it is best to select the *sqlite tools* link under Precompiled Binaries for Windows. To avoid using the command line interface, you can download DB Browser—https://sqlitebrowser.org/dl/—which provides a nice interface that works on all major platforms.

We used SQLite to create a subset of the relational database shown in Figure 4.1. The database—CreditCardPromotion.db—is part of your supplementary materials for this chapter. Figure 4.2 displays the database structure as seen with DB Browser. Script 4.1 gives the statements for accessing the database. Prior to executing the script, you must visit the CRAN repository and install two interface packages—RSQLite and DBI. Let's take a look at the statements in Script 4.1.

Script 4.1 Creating a Data Frame with Data from a Relational Database

```
  library(RSQLite)# R interface to SQLite
> library(DBI)    # R database interface
> setwd("C:/Users/richa/desktop/sqlite")#Database is stored here

> dbCon <- dbConnect(SQLite(), dbname = "CreditCardPromotion.db")
> custab <- dbGetQuery(dbCon,"Select CustomerID,
+                 IncomeRange,Gender,Age from Customer")
> custab

  CustomerID IncomeRange Gender Age
1          1      40-50K   Male  45
2          2      30-40K Female  40
3          3      40-50K Female  42

> ccpLife <- dbGetQuery(dbCon,
+      "Select Customer.CustomerID, IncomeRange,
+              Gender,Age, Status,Promotion_C.PromotionID
+       from   Customer, Promotion_C
+       where  PromotionID =10 and
+              Customer.CustomerID =Promotion_C.CustomerID
+              ")
> ccpLife

  CustomerID IncomeRange Gender Age Status PromotionID
1          1      40-50K   Male  45    Yes          10
```

```
2           2      30-40K Female  40    Yes          10
3           3      40-50K Female  42     No          10

> # Change column name from Status to LifeInsPromo
> colnames(ccpLife)[colnames(ccpLife)=="status"]<-"LifeInsPromo"
> ccpLife

  CustomerID IncomeRange Gender Age LifeInsPromo PromotionID
1          1      40-50K   Male  45          Yes          10
2          2      30-40K Female  40          Yes          10
3          3      40-50K Female  42           No          10

> dbDisconnect(dbCon)
```

The first two statements load the installed libraries mentioned above. Modify the *setwd* statement to match the location of your copy of the database. The *dbConnect* function makes the connection to the database. The first *Select* statement obtains information from the *Customer* table. Notice that the database contains three customers.

You can see how the promotion IDs correspond to promotion names with a click within DB Browser on *browse data* or by submitting the following query in your console window:

```
dbGetQuery(dbCon,"Select * from Promotion")

                Description PromotionID
1       Magazine Promotion          20
2          Watch Promotion          30
3 Life Insurance Promotion          10
```

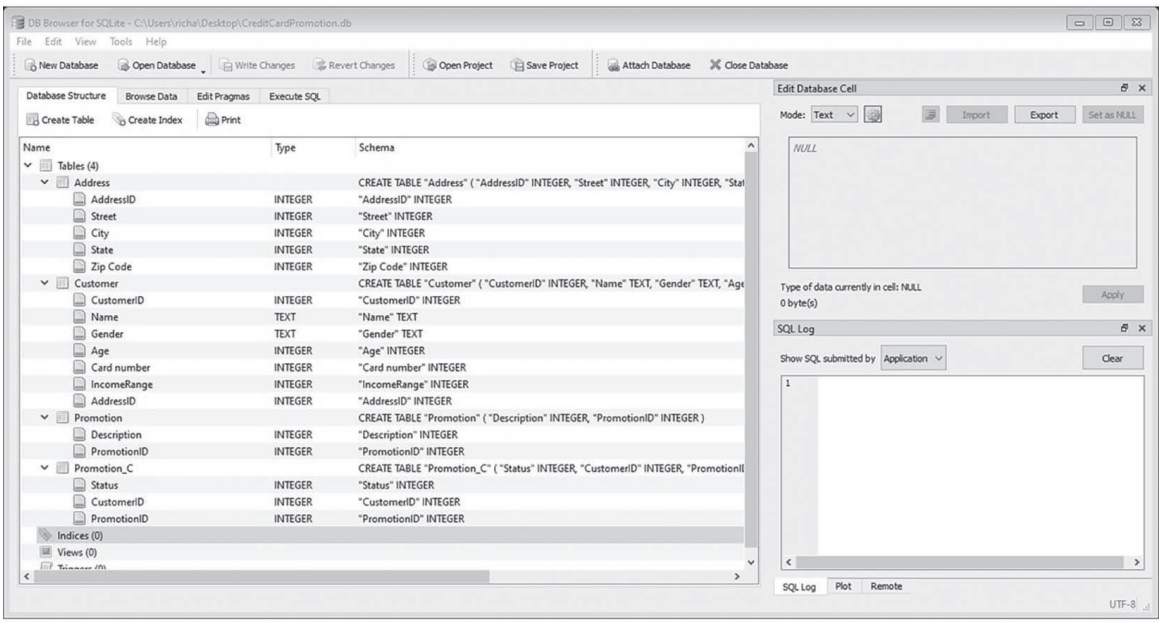


FIGURE 4.2 The CreditCardPromotion database.

Replace *Promotion* with *Promotion_C* to see how all three customers reacted to the three promotions.

The next query is of more interest as its purpose is to create the *Life Insurance Promotion* column shown in Table 2.2. To achieve this, the query joins the *Customer* and *Promotion_C* tables. The join creates a new table that combines the *Status* and *PromotionID* columns from Promotion_C with the information in the *Customer* table. Lastly, we use the *colnames* function to change *Status* to *LifeInsPromo* giving the desired result. Adding the magazine and watch promotions to ccpLife is left as an exercise. As you can see, when the data is stored in a relational database, the task of extracting target data can be a challenge.

4.2.2 Additional Sources for Target Data

The possibility also exists for extracting data from multiple databases. If target data is to be taken from more than one source, the transfer process can be tedious. Consider a simple example where one operational database stores customer gender with the coding *male = 1*, *female = 2*. A second database stores the gender coding as *male = M* and *female = F*. The coding for *male* and *female* must be consistent throughout all records in the target data or the data will be of little use. The process of promoting this consistency when transporting the data is a form of *data transformation*. Other types of data transformations are discussed in Section 4.4.

A third possibility for harvesting target data is the data warehouse. The data warehouse is a historical database designed for decision support rather than transaction processing (Kimball et al., 1998). Thus, only data useful for decision support is extracted from the operational environment and entered into the warehouse database. Data transfer from the operational database to the warehouse is an ongoing process usually accomplished on a daily basis after the close of the regular business day.

A fourth scenario is when data requires a distributed environment supported by a cluster of servers. With distributed data, the KDD process model must be supplemented with the added complexities of data distribution and solution aggregation. Lastly, a special case exists with streaming data where real-time analysis makes data preprocessing at best extremely difficult.

4.3 DATA PREPROCESSING

Most data preprocessing is in the form of data cleaning, which involves accounting for noise and dealing with missing information. Ideally, the majority of data preprocessing takes place before data is permanently stored in a structure such as a data warehouse.

4.3.1 Noisy Data

Noise represents random error in attribute values. In very large datasets, noise can come in many shapes and forms. Common concerns with noisy data include the following:

- How do we find duplicate records?
- How can we locate incorrect attribute values?
- What data smoothing operations should be applied to our data?

- How can we find and process outliers?
- How do we deal with missing data items?

4.3.1.1 Duplicate Records

Suppose a certain weekly publication has 100,000 subscribers and 0.1% of all mailing list entries have erroneous dual listings under a variation of the same name (e.g., Jon Doe and John Doe). Therefore, 100 extra publications are processed and mailed each week. At a processing and mailing cost of $2.00 for each publication, the company spends over $10,000 each year in unwarranted costs.

4.3.1.2 Incorrect Attribute Values

Finding errors in categorical data presents a problem in large datasets. Most data mining tools offer a summary of frequency values for categorical attributes. We should consider attribute values having frequency counts near 0 as errors.

A numeric value of 0 for an attribute such as blood pressure or weight is an obvious error. Such errors often occur when data is missing and default values are assigned to fill in for missing items. In some cases, such errors can be seen by examining class mean and standard deviation scores. However, if the dataset is large and only a few incorrect values exist, finding such errors can be difficult.

4.3.1.3 Data Smoothing

Data smoothing is both a data cleaning and data transformation process. Several data smoothing techniques attempt to reduce the number of values for a numeric attribute. Some classifiers, such as neural networks, use functions that perform data smoothing during the classification process. When data smoothing is performed during classification, the data smoothing is said to be internal. External data smoothing takes place prior to classification. Rounding and computing mean values are two simple external data smoothing techniques. Mean value smoothing is appropriate when we wish to use a classifier that does not support numerical data and would like to retain coarse information about numerical attribute values. In this case, all numerical attribute values are replaced by a corresponding class mean.

Another common data smoothing technique attempts to find atypical (outlier) instances in the data. Outliers often represent errors in the data whereby the items should be corrected or removed. For instance, a credit card application where applicant age is given as -21 is clearly incorrect. In other cases, it may be counterproductive to remove outliers. For example, in credit card fraud detection, the outliers are those items we are most interested in finding.

Unsupervised outlier detection methods often make the assumption that ordinary instances will cluster together. If definite patterns in the data do not exist, unsupervised techniques will flag an undue number of ordinary instances as outliers.

4.3.2 Preprocessing with R

Let's use the *creditScreening.csv* dataset to illustrate some basic preprocessing techniques employing R functions. This dataset is a viable choice as it contains numeric, categorical, and missing data items. The dataset includes information about 690 individuals who

applied for a credit card. The data has 15 input attributes and one output attribute indicating whether an individual credit card application was accepted (+) or rejected (–). Privacy issues prevent knowledge of the semantic meaning of the input attributes. A more complete description of the dataset is given in Section 5.4 of Chapter 5. The output of the *str* function for the first three attributes and the class attribute gives the following:

```
> str(creditScreening)
'data.frame': 690 obs. of 16 variables:
 $ one     : Factor w/ 3 levels "?","a","b": 3 2 2 3 3 3 3 2 3 3
...
 $ two     : num  30.8 58.7 24.5 27.8 20.2 ...
 $ three   : num  0 4.46 0.5 1.54 5.62 ...
 $ class   : Factor w/ 2 levels "-","+": 2 2 2 2 2 2 2 2 2 2 ...
```

The *str* function tells us the file has been imported as a data frame. Also, attribute *one* lists one of its values as "?" indicating a likely unknown attribute value. The subset function can give us the rows of those instances with attribute *one* having a value of "?". Here are the first four such instances obtained with the subset function.

```
subset(creditScreening, one=="?")

    one   two three four five six seven eight nine ten eleven twelve
218   ? 40.83 3.500    u    g   i    bb 0.500    f   f     0       f
237   ? 32.25 1.500    u    g   c     v 0.250    f   f     0       t
265   ? 28.17 0.585    u    g  aa     v 0.040    f   f     0       f
344   ? 29.75 0.665    u    g   w     v 0.250    f   f     0       t
.....
```

The summary function provides information about missing items for both categorical and numeric attributes. Here is summary information for attributes *one* through *six*:

```
summary(creditScreening)

one         two              three       four     five        six
?: 12  Min.   :13.75  Min.   : 0.000    :  6     :  6   c      :137
a:210  1st Qu.:22.60  1st Qu.: 1.000  l:  2   g :519   q      : 78
b:468  Median :28.46  Median : 2.750  u:519   gg:  2   w      : 64
       Mean   :31.57  Mean   : 4.759  y:163   p :163   i      : 59
       3rd Qu.:38.23  3rd Qu.: 7.207                   aa     : 54
       Max.   :80.25  Max.   :28.000                   ff     : 53
       NA's   :12                                     (Other):245
```

The summary information above tells us attribute *two* contains 12 NA's. We also see the "?" for attribute *one*. The *unique* function lists the unique values found in a column of data. Attribute *six* will likely be of little predictive value as it has 16 unique values:

```
length(unique(creditScreening[,6]))
[1] 16
```

Attributes *nine*, *ten*, and *twelve* (not shown) are more likely to be useful as they represent even distributions of true and false values.

4.3.3 Detecting Outliers

Graphical approaches are often employed for outlier detection. Figure 4.3 shows a histogram of creditScreening$fourteen. The histogram shows that the greatest majority of values lie well below 1000. However, the histogram clearly indicates a very small set of values in the 1700–2000 range. Given the summary statement tells us attribute fourteen has 14 missing items, these outliers are likely errors in the data.

There are also functions for detecting outliers. We will investigate the *outlierTest* function available with the *car* package in Chapter 5. Since outlier detection operators do not take attribute significance into account, the outliers detected by an operator may not be useful. When using machine learning methods such as neural networks that do not have attribute selection built into the modeling process, it is best to first apply an attribute selection technique to the data prior to attempting outlier detection.

4.3.4 Missing Data

Imputation is a general term used for replacing missing data with substituted values. In most cases, missing attribute values indicate lost information. For example, a missing value for the attribute *age* certainly indicates a data item that exists but is unaccounted for. However, a missing value for *salary* may be taken as an unentered data item, but it could also indicate an individual who is unemployed. Some machine learning techniques are able to deal directly with missing values. However, many methods require all attributes to contain a value.

The following are possible options for dealing with missing data *before* the data is presented to a learning algorithm.

- *Discard records with missing values*. This method is most appropriate when a small percent of the total number of instances contain missing data and we can be certain that missing values do indeed represent lost information.

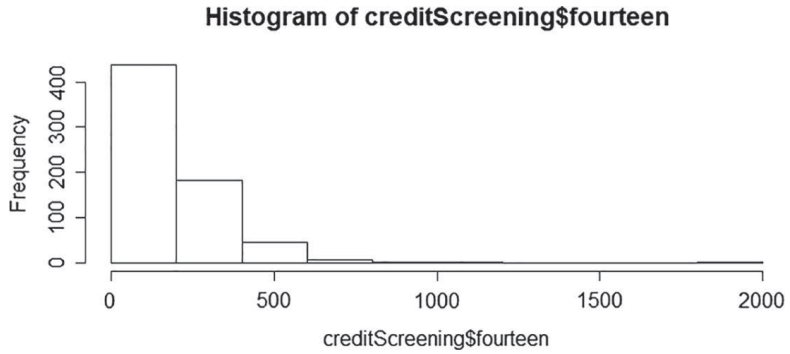


FIGURE 4.3 A histogram of attribute creditScreening$fourteen.

- *For real-valued data, replace missing values with the class mean*. In most cases, this is a reasonable approach for numerical attributes. Options such as replacing missing numeric data with a zero or some arbitrarily large or small value is generally a poor choice.
- *Replace missing attribute values with the values found within other highly similar instances*. This technique is appropriate for either categorical or numeric attributes.

Some machine learning techniques allow instances to contain missing values. Here are three ways that machine learning techniques deal with missing data while learning:

1. *Ignore missing values*. Several machine learning algorithms, including neural networks (Chapter 8) and Bayes classifier (Chapter 5), use this approach.
2. *Treat missing values as equal comparisons*. This approach is dangerous with very noisy data in that dissimilar instances may appear to be very much alike.
3. *Treat missing values as unequal comparisons*. This is a pessimistic approach but may be appropriate. Two similar instances containing several missing values will appear dissimilar.

Finally, a knowledge-based approach for resolving missing information uses supervised learning to determine likely values for missing data. When the missing attribute is categorical, we designate the attribute with missing values as an output attribute. The instances with known values for the attribute are used to build a classification model. The created model is then summoned to classify the instances with missing values. For numerical data, we can use a regression technique or a neural network and apply the same strategy.

Two functions located in *functions.zip* written for your text are designed to help detect and replace missing items. When given a data frame, *removeNAS* lists all instances having at least one NA. It returns a data frame with all instances having missing numeric data removed. The second function, *RepVal*, replaces any item in a given column of a data frame with a specified value. For example, the call

```
y<-repVal(creditScreening, 1, '?','a')
```

creates a new data frame y where all '?' values in column one are replaced with an 'a'.

4.4 DATA TRANSFORMATION

Data transformation can take many forms and is necessary for a variety of reasons. We offer a description of some familiar data transformations in the following sections.

4.4.1 Data Normalization

A common data transformation involves changing numeric values so they fall within a specified range. Classifiers such as neural networks do better with numerical data scaled to a range between 0 and 1. Normalization is particularly appealing with distance-based

classifiers, because by normalizing attribute values, attributes with a wide range of values are less likely to outweigh attributes with smaller initial ranges. Four common normalization methods include:

- *Decimal scaling*. Decimal scaling divides each numerical value by the same power of 10. For example, if we know the values for an attribute range between –1000 and 1000, we can change the range to –1 and 1 by dividing each value by 1000.
- *Min-Max normalization*. Min-Max is an appropriate technique when minimum and maximum values for an attribute are known. The formula is

$$NewValue = \frac{originalValue - oldMin(NewMax - newMin) + newMin}{oldMax - oldMin}$$

  where *oldMax* and *oldMin* represent the original maximum and minimum values for the attribute in question. *NewMax* and *newMin* specify the new maximum and minimum values. *NewValue* represents the transformation of *originalValue*. This transformation is particularly useful with neural networks where the desired range is [0,1]. In this case, the formula simplifies to

$$NewValue = \frac{originalValue - oldMin}{oldMax - oldMin}$$

- *Normalization using Z-scores*. Z-score normalization converts a value to a standard score by subtracting the attribute mean (μ) from the value and dividing by the attribute standard deviation (σ). Specifically,

$$newValue = \frac{originalValue - \mu}{\sigma}$$

This technique is particularly useful when maximum and minimum values are not known.

- *Logarithmic normalization*. The base *b* logarithm of a number *n* is the exponent to which *b* must be raised to equal *n*. For example, the base 2 logarithm of 64 is 6 because $2^6 = 64$. Replacing a set of values with their logarithms has the effect of scaling the range of values without loss of information.

R's generic *scale* function scales a column of numeric data to a mean of 0 and a standard deviation of 1. Execute the following statements to get a clear picture of how the *scale* function normalizes *ccpromo$Age*.

```
my.sc <- scale(ccpromo$Age)
str(my.sc)
my.sc <- data.frame(col1=my.sc)
```

```
str(my.sc)
my.sc<-as.numeric(my.sc$col1)
my.sc
sd(my.sc)
mean(my.sc)
```

4.4.2 Data Type Conversion

Many machine learning tools, including neural networks and some statistical methods, cannot process categorical data. Therefore, converting categorical data to a numeric equivalent is a common data transformation.

4.4.3 Attribute and Instance Selection

Classifiers such as decision trees with built-in attribute selection are less likely to suffer from the effects of datasets containing attributes of little predictive value. Unfortunately, many machine learning algorithms such as neural networks and nearest neighbor classifiers are unable to differentiate between relevant and irrelevant attributes. This is a problem, as these algorithms do not generally perform well with data containing a wealth of attributes that are not predictive of class membership. Furthermore, it has been shown that the number of training instances needed to build accurate supervised models is directly affected by the number of irrelevant attributes in the data. To overcome these problems, we must make decisions about which attributes and instances to use when building our models. The following is a possible algorithm to help us with attribute selection:

1. Given *N* attributes, generate the set *S* of all possible attribute combinations.
2. Remove the first attribute combination from set *S* and generate a machine learning model *M* using these attributes.
3. Measure the goodness of model *M*.
4. Until *S* is empty
   a. Remove the next attribute combination from *S* and build a model using the next attribute combination in *S*.
   b. Compare the goodness of the new model with the saved model *M*. Call the better model *M* and save this model as the best model.
5. Model *M* is the model of choice.

This algorithm will surely give us a best result. The problem with the algorithm lies in its complexity. If we have a total of n attributes, the total number of attribute combinations is $2^n - 1$. The task of generating and testing all possible models for any dataset containing more than a few attributes is not possible. Let's investigate a few techniques we can apply.

4.4.3.1 Wrapper and Filtering Techniques

Attribute selection methods are generally divided into filtering and wrapper techniques. *Filtering* methods select attributes based on some measure of quality independent of the

algorithm used to build a final model. With *wrapper* techniques, attribute goodness is measured in the context of the learning algorithm used to build the final model. That is, the algorithm is wrapped into the attribute selection process. You will see several examples of both methods in the chapters that follow.

4.4.3.2 More Attribute Selection Techniques

In addition to applying wrapper and filtering techniques, several other steps can be taken to help determine which attributes to eliminate from consideration:

1. Highly correlated input attributes are redundant. Most machine learning tools build better models when only one attribute from a set of highly correlated attributes is designated as an input value. Methods for detecting correlated attributes include graphical techniques as well as quantitative measures.
2. *Principal component analysis* is a statistical technique that looks for and removes possible correlational redundancy within the data by replacing the original attributes with a smaller set of artificial values.
3. A *self-organizing map* or SOM is a neural network trained with unsupervised learning that can be used for attribute reduction. SOMs are described in Chapter 8 when we focus on neural network models.

4.4.3.3 Creating Attributes

Attributes of little predictive power can sometimes be combined with other attributes to form new attributes with a high degree of predictive capability. As an example, consider a database consisting of data about stocks. Conceivable attributes include current stock price, 12-month price range, growth rate, earnings, market capitalization, company sector, and the like. The attributes price and earnings are of some predictive value in determining a future target price. However, the ratio of price to earnings (P/E ratio) is known to be more useful. A second created attribute likely to effectively predict a future stock price is the stock P/E ratio divided by the company growth rate. Here are a few transformations commonly applied to create new attributes:

- Create a new attribute where each value represents a ratio of the value of one attribute divided by the value of a second attribute.
- Create a new attribute whose values are differences between the values of two existing attributes.
- Create a new attribute with values computed as the percent increase or percent decrease of two current attributes. Given two values v_1 and v_2 with $v_1 < v_2$, the percent increase of v_2 with respect to v_1 is computed as

$$Percent\ Increase(v_2, v_1) = \frac{v_2 - v_1}{v_1}$$

If $v_1 > v_2$, we subtract v_2 from v_1 and divide by v_1, giving a percent decrease of v_2 with respect to v_1.

New attributes representing differences and percent increases or decreases are particularly useful with time series analysis. *Time series analysis* models changes in behavior over a time interval. For this reason, attributes created by computing differences between one time interval and the next time interval are important.

4.4.4 Creating Training and Test Set Data

Once we have our data preprocessed and transformed, our final step prior to model building is the selection of training and test data. A common scenario for supervised learning is to start by randomizing the data. This is followed by selecting 2/3 of the data for model building and the remaining 1/3 for testing. This is illustrated below using the creditScreening data. Let's take a look at each statement.

```
> set.seed(1000)
> # Randomize and split the data for 2/3 training, 1/3 testing
> credit.data <- creditScreening
> index <- sample(1:nrow(credit.data), 2/3*nrow(credit.data))
> credit.train <- credit.data[index,]
> credit.test <-  credit.data[-index,]
```

The *set.seed* function generates an initial starting point for the pseudo-random number generator. Using the same seed each time makes our experiments reproducible. Next, a copy of the creditScreening data frame is made for the experiment. The *sample* function is given two arguments. The first argument is the total number of rows in the data frame—690 in this case. The second argument specifies the size of the sample. In this case, index will contain a list of 460 randomized positive integer values between 1 and 690.

The statement `credit.data[index,]` copies all of the instances with row numbers specified by index into `credit.train`. Finally, `credit.data[-index,]` specifies `credit.test` is to receive a copy of the remaining 1/3 of the instances. If data normalization is needed, the *scale* function would precede sampling.

We have provided but one method for randomizing and splitting data for training and testing. Countless other techniques can be used to achieve the same result. The next section overviews an alternative to the training–testing scenario when data is lacking.

4.4.5 Cross Validation and Bootstrapping

If ample test data are not available, we can apply a technique known as *cross validation*. With this method, all available data are partitioned into n fixed-size units. $n-1$ of the units are used for training, whereas the nth unit is the test set. This process is repeated until each of the fixed-size units has been used as test data. Model test set correctness is computed as the average accuracy realized from the n training–testing trials. Experimental results have shown a value of 10 for n to be maximal in most situations. Several applications of cross

validation to the data can help ensure an equal distribution of classes within the training and test datasets.

Bootstrapping is an alternative to cross validation. With bootstrapping, we allow the training set selection process to choose the same training instance several times. This happens by placing each training instance back into the data pool after it has been selected for training. It can be shown mathematically that if a dataset containing n instances is sampled n times using the bootstrap technique, the training set will contain approximately two-thirds of the n instances. This leaves one-third of the instances for testing.

4.4.6 Large-Sized Data

While traditional machine learning algorithms assume the entire dataset resides in memory, current datasets are often too large to satisfy this requirement. One possible way to deal with this problem is to process as much of the data as possible while repeatedly retrieving the remaining data from a secondary storage device. Clearly, this solution is not reasonable given the inefficiency of secondary storage data retrieval. Another possibility is to employ a distributed environment where data can be divided among several processors. When this is not feasible, we must limit our list of plausible algorithm choices to those that exhibit the property of scalability.

An algorithm is said to be *scalable* if given a fixed amount of memory, its runtime increases linearly with the number of records in the dataset. The simplest approach to scalability is sampling. With this technique, models are built and tested using a subset of the data that can be efficiently processed in memory. This method works well with supervised learning and unsupervised clustering when the data contains a minimal number of outliers. However, it would be difficult to put our trust in this method as a general approach for handling large-sized data.

Some traditional algorithms such as Naïve Bayes classifier (Chapter 5) are scalable. Cobweb and Classit (Chapter 11) are two scalable unsupervised clustering techniques as data is processed and discarded incrementally.

4.5 CHAPTER SUMMARY

Knowledge discovery can be modeled as a seven-step process that includes goal identification, target data creation, data preprocessing, data transformation, data mining, result interpretation and evaluation, and knowledge application. A clear statement about what is to be accomplished is a good starting point for successful knowledge discovery. Creating a target dataset often involves extracting data from a warehouse, a transactional database, or a distributed environment. Transactional databases do not store redundant data, as they are modeled to quickly update and retrieve information. Because of this, the structure of the data in a transactional database must be modified before data mining can be applied.

Prior to exercising a machine learning tool, the gathered data is preprocessed to remove noise. Missing data is of particular concern because many algorithms are unable to process missing items. In addition to data preprocessing, data transformation techniques can be applied before model building takes place. Data transformation methods such as data normalization and attribute creation or elimination are often necessary for a best result.

4.6 KEY TERMS

- *Attribute filtering*. Attribute filtering methods select attributes based on some measure of quality independent of the algorithm that will be used to build a final model.
- *Bootstrapping*. Allowing instances to appear more than once in a training set.
- *Cross validation*. Partitioning a dataset into *n* fixed-size units. *n* – 1 units are used for training and the *n*th unit is used as a test set. This process is repeated until each of the fixed-size units has been used as test data. Model test set correctness is computed as the average accuracy realized from the *n* training–testing trials.
- *Data normalization*. A data transformation where numeric values are modified to fall within a specified range.
- *Data preprocessing*. The step of the KDD process that deals with noisy and missing data.
- *Data transformation*. The step of the KDD process that deals with data normalization and conversion as well as the addition and/or elimination of attributes.
- *Decimal scaling*. A data transformation technique for a numeric attribute where each value is divided by the same power of 10.
- *Logarithmic normalization*. A data transformation method for a numeric attribute where each numeric value is replaced by its base b logarithm.
- *Min-Max normalization*. A data transformation method that is used to transform a set of numeric attribute values so they fall within a specified numeric range.
- *Noise*. Random error in data.
- *Outlier*. An instance that by some measure deviates significantly from other instances.
- *Relational database*. A database where data is represented as a collection of tables containing rows and columns. Each column of the table is known as an attribute, and each row of the table stores information about one data record.
- *Time series analysis*. Any technique that models changes in behavior over a period of time.
- *Tuple*. An individual row of a table in a relational database.
- *Wrapper technique*. An attribute selection method that bases attribute goodness in the context of the learning algorithm used to build the final model.
- *Z-score normalization*. A data normalization technique for a numeric attribute where each numeric value is replaced by its standardized difference from the mean.

EXERCISES

Review Questions

1. Differentiate between the following terms:
   a. Data cleaning and data transformation
   b. Internal and external data smoothing
   c. Decimal scaling and Z-score normalization
   d. Filter and wrapper attribute selection
2. In Section 4.4, you learned about basic methods machine learning algorithms use to deal with missing data while learning. Decide which technique is best for the following problems. Explain each choice.
   a. A model designed to accept or reject credit card applications.
   b. A model for determining who should receive a promotional flyer in the mail.
   c. A model designed to determine those individuals likely to develop colon cancer.
   d. A model to decide whether to drill for oil in a certain region.
   e. A model for approving or rejecting candidates applying to refinance their home.

Experimenting with R

1. Consider Script 4.1 Copy the script and add two SQL Select clauses to the new script that will create the columns of yes and no responses for *magazine promotion* and *watch promotion*. Use *cbind* to add each column to ccpLife. Also, be sure to remove the PromotionID column from the final table. Your final output will show a single table with the headings: CustomerID, IncomeRange, Gender, Age, LifeInsPromo, MagazinePromo, and WatchPromo.

Computational Questions

1. Set up a general formula for a Min-Max normalization as it would be applied to the attribute *age* for the data in Table 2.2. Transform the data so the new minimum value is 0 and the new maximum value is 1. Apply the formula to determine a transformed value for *age* = 35.
2. Answer the following questions about percent increase and percent decrease.
   a. The price of a certain stock increases from $25.00 to $40.00. Compute the percent increase in the stock price.
   b. The original price of the stock is $40.00. The price decreases by 50%. What is the current stock price?
3. You are to apply a base 2 logarithmic normalization to a certain numeric attribute whose current range of values falls between 2300 and 10,000. What will be the new range of values for the attribute once the normalization has been completed?
4. Apply a base 10 logarithmic normalization to the values for attribute *age* in Table 2.2. Use a table to list the original values as well as the transformed values.

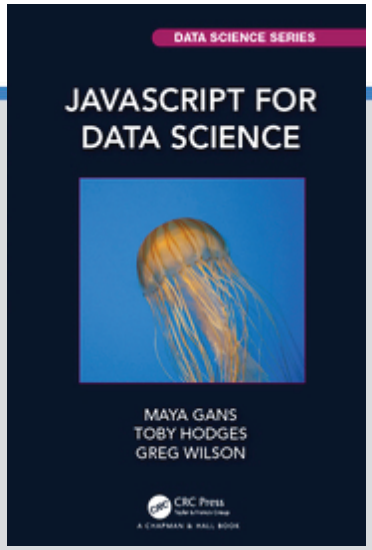
Installed Packages and Functions

Package Name	Function(s)
base / stats	*abs, c, cat, cbind, colnames, data.frame, factor, length, library, ncol, nrow, return, round, sample, scale, set.seed, setwd, str, subset, summary, table, unique*
car	*outlierTest*
DBI & RSQLite	*dbConnect, dbDisconnect, dbGetQuery*

CHAPTER

3

HTML AND CSS



JavaScript for Data Science

By Maya Gans, Toby Hodges, Greg Wilson

Learn more

5 HTML and CSS

HTML is the standard way to represent documents for presentation in web browsers, and CSS is the standard way to describe how it should look. Both are more complicated than they should have been, but in order to create web applications, we need to understand a little of both.

5.1 FORMATTING

An HTML **document** contains **elements** and text (and possibly other things that we will ignore for now). Elements are shown using **tags**: an opening tag `<tagname>` shows where the element begins, and a corresponding closing tag `</tagname>` (with a leading slash) shows where it ends. If there's nothing between the two, we can write `<tagname/>` (with a trailing slash).

A document's elements must form a **tree** (Figure 5.1), i.e., they must be strictly nested. This means that if Y starts inside X, Y must end before X ends, so `<X>...<Y>...</Y></X>` is legal, but `<X>...<Y>...</X></Y>` is not. Finally, every document should have a single **root element** that encloses everything else, although browsers aren't strict about enforcing this. In fact, most browsers are pretty relaxed about enforcing any kind of rules at all, since most people don't obey them anyway.

5.2 TEXT

The text in an HTML page is normal printable text. However, since < and > are used to show where tags start and end, we must use **escape sequences** to represent them, just as we use `\"` to represented a literal double-quote character inside a double-quoted string in JavaScript. In HTML, escape sequences are written `&name;`, i.e., an ampersand, the name of the character, and a semi-colon. A few common escape sequences are shown in Table 5.1.

Name	Escape Sequence	Character
Less than	`<`	<
Greater than	`>`	>
Ampersand	`&`	&
Copyright	`&copy;`	©
Plus/minus	`&plusmn;`	±
Micro	`&micro;`	µ

Table 5.1: HTML Escapes

The first two are self-explanatory, and `&` is needed so that we can write a

literal ampersand (just as \\ is needed in JavaScript strings so that we can write a literal backslash). `&copy;`, `&plusmn;`, and `&micro;` are usually not needed any longer, since most editors will allow us to put non-ASCII characters directly into documents these days, but occasionally we will run into older or stricter systems.

5.3 PAGES

An HTML page should have:

- a single `html` element that encloses everything else
- a single `head` element that contains information about the page
- a single `body` element that contains the content to be displayed.

It doesn't matter whether or how we indent the tags showing these elements and the content they contain, but laying them out on separate lines and indenting to show nesting helps human readers. Well-written pages also use comments, just like code: these start with `<!--` and end with `-->`. Unfortunately, comments cannot be nested, i.e., if you comment out a section of a page that already contains a comment, the results are unpredictable.

Here's an empty HTML page with the structure described above:

```
<html>
  <head>
    <!-- description of page goes here -->
  </head>
  <body>
    <!-- content of page goes here -->
  </body>
</html>
```

Nothing shows up if we open this in a browser, so let's add a little content:

```
<html>
  <head>
    <title>This text is displayed in the browser bar</title>
  </head>
  <body>
    <h1>Displayed Content Starts Here</h1>
    <p>
      This course introduces core features of <em>JavaScript</em>
      and shows where and how to use them.
    </p>
    <!-- The word "JavaScript" is in italics (emphasis) in the preceding -->
        <!-- paragraph. -->
  </body>
</html>
```

- The `title` element inside `head` gives the page a title. This is displayed in the browser bar when the page is open, but is *not* displayed as part of the page itself.

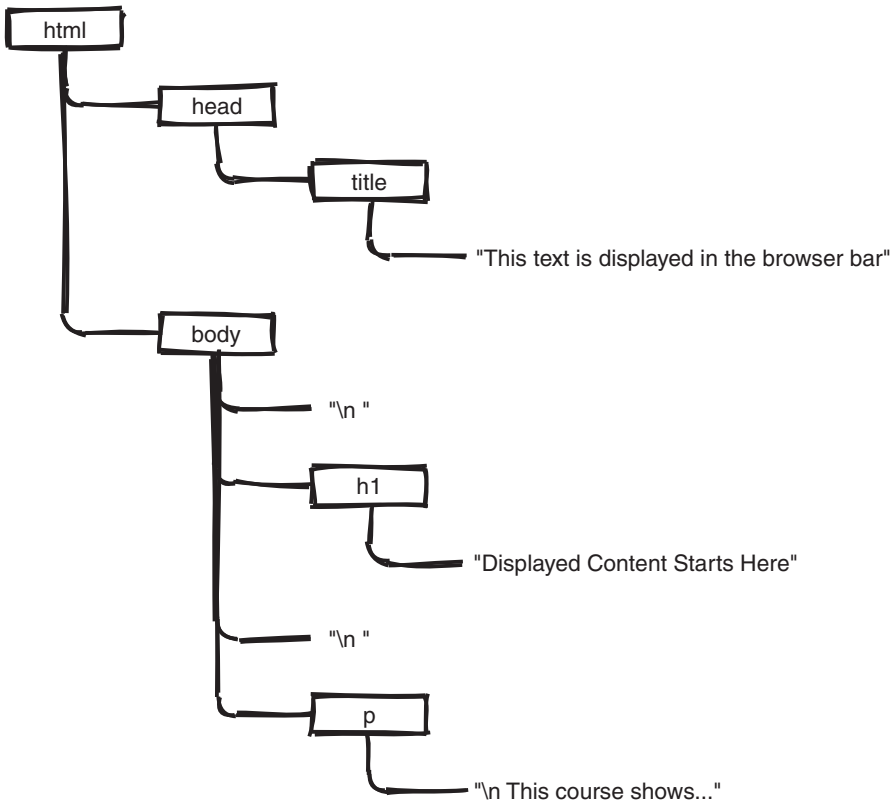


Figure 5.1: HTML as a Tree

- The `h1` element is a level-1 heading; we can use `h2`, `h3`, and so on to create sub-headings.
- The `p` element is a paragraph.
- Inside a heading or a paragraph, we can use `em` to *emphasize* text. We can also use `strong` to make text **stronger**. Tags like these are better than tags like `i` (for italics) or `b` (for bold) because they signal intention rather than forcing a particular layout. Someone who is visually impaired, or someone using a small-screen device, may want emphasis of various kinds displayed in different ways.

5.4 ATTRIBUTES

Elements can be customized by giving them **attributes**, which are written as `name="value"` pairs inside the element's opening tag. For example:

```
<h1 align="center">A Centered Heading</h1>
```

centers the `h1` heading on the page, while:

```
<p class="disclaimer">This planet provided as-is.</p>
```

marks this paragraph as a disclaimer. That doesn't mean anything special to HTML, but as we'll see later, we can define styles based on the `class` attributes of elements.

An attribute's name may appear at most once in any element, just like a key can only appear once in any JavaScript object, so `<p align="left" align="right">...</p>` is illegal. If we want to give an attribute multiple values—for example, if we want an element to have several classes—we put all the values in one string. Unfortunately, as the example below shows, HTML is inconsistent about whether values should be separated by spaces or semi-colons:

```
<p class="disclaimer optional" style="color: blue; font-size: 200%;">
```

However they are separated, values are supposed to be quoted, but in practice we can often get away with `name=value`. And for Boolean attributes whose values are just true or false, we can even sometimes just get away with `name` on its own.

5.5 LISTS

Headings and paragraphs are all very well, but data scientists need more. To create an unordered (bulleted) list, we use a `ul` element, and wrap each item inside the list in `li`. To create an ordered (numbered) list, we use `ol` instead of `ul`, but still use `li` for the list items.

```
<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

- first
- second
- third

```
<ol>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ol>
```

1. first
2. second
3. third

Lists can be nested by putting the inner list's `ul` or `ol` inside one of the outer list's `li` elements:

```
<ol>
  <li>Major A
    <ol>
```

```
      <li>minor p</li>
      <li>minor q</li>
    </ol>
  </li>
  <li>Major B
    <ol>
      <li>minor r</li>
      <li>minor s</li>
    </ol>
  </li>
</ol>
```

1. Major A
    1. minor p
    2. minor q
2. Major B
    1. minor r
    2. minor s

5.6 TABLES

Lists are a great way to get started, but if we *really* want to impress people with our data science skills, we need tables. Unsurprisingly, we use the `table` element to create these. Each row is a `tr` (for "table row"), and within rows, column items are shown with `td` (for "table data") or `th` (for "table heading").

```
<table>
  <tr> <th>Alkali</th>   <th>Noble Gas</th> </tr>
  <tr> <td>Hydrogen</td> <td>Helium</td>    </tr>
  <tr> <td>Lithium</td>  <td>Neon</td>      </tr>
  <tr> <td>Sodium</td>   <td>Argon</td>     </tr>
</table>
```

Alkali	Noble Gas
Hydrogen	Helium
Lithium	Neon
Sodium	Argon

Do *not* use tables to create multi-column layouts: there's a better way.

5.7 LINKS

Links to other pages are what make HTML hypertext (Figure 5.2). Confusingly, the element used to show a link is called `a`. The text inside the element is displayed and (usually) highlighted for clicking. Its `href` attribute specifies what the link is pointing at; both local filenames and URLs are supported. Oh, and we can use `<br/>`

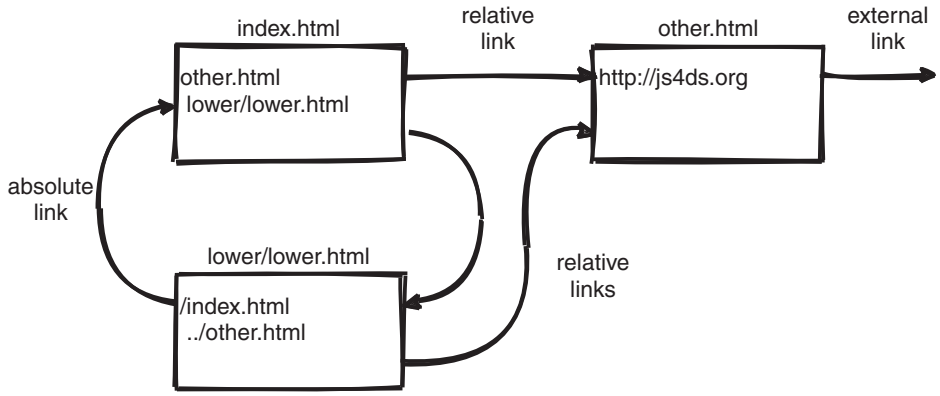


Figure 5.2: Pages and Links

to force a line break in text (with a trailing slash inside the tag, since the `br` element doesn't contain any content):

```
<a href="https://nodejs.org/">Node.js</a>
<br/>
<a href="https://facebook.github.io/react/">React</a>
<br/>
<a href="../index.html">home page (relative path)</a>
```

This appears as:

```
Node.js
React
home page (relative path)
```

with the usual clickability.

5.8 IMAGES

Images can be stored inside HTML pages in two ways: by using SVG (which we will discuss in Chapter 8) or by encoding the image as text and including that text in the body of the page, which is clever, but makes the source of the pages very hard to read.

It is far more common to store each image in a separate file and refer to that file using an `img` element (which also allows us to use the image in many places without copying it). The `src` attribute of the `img` tag specifies where to find the file; as with the `href` attribute of an `a` element, this can be either a URL or a local path. Every `img` should also include a `title` attribute (whose purpose is self-explanatory) and an `alt` attribute with some descriptive text to aid accessibility and search engines. (Again, we have wrapped and broken lines so that they will display nicely in the printed version.)

```
    <img src="./assets/logo.png" title="Book Logo"
         alt="Displays the book logo using a local path" />
    <img src="https://js4ds.org/assets/logo.png"
         title="Book Logo"
         alt="Display the book logo using a URL" />
```

Two things to note here are:

1. Since `img` elements don't contain any text, they are often written with the trailing-slash notation. However, they are also often written improperly as `<img src="...">` without any slashes at all. Browsers will understand this, but some software packages will complain.
2. If an image file is referred to using a path rather than a URL, that path can be either **relative** or **absolute**. If it's a relative path, it's interpreted starting from where the web page is located; if it's an absolute path, it's interpreted relative to wherever the web browser thinks the **root directory** of the filesystem is. As we will see in Chapter 12, this can change from one installation to the next, so you should always try to use relative paths, except where you can't. It's all very confusing...

5.9 CASCADING STYLE SHEETS

When HTML first appeared, people styled elements by setting their attributes:

```
<html>
  <body>
    <h1 align="center">Heading is Centered</h1>
    <p>
      <b>Text</b> can be highlighted
      or <font color="coral">colorized</font>.
    </p>
  </body>
</html>
```

Many still do, but a better way is to use **Cascading Style Sheets** (CSS). These allow us to define a style once and use it many times, which makes it much easier to maintain consistency. (We were going to say "...and keep pages readable", but given how complex CSS can be, that's not a claim we feel we can make.) Here's a page that uses CSS instead of direct styling:

```
<html>
  <head>
    <link rel="stylesheet" href="simple-style.css" />
  </head>
  <body>
    <h1 class="title">Heading is Centered</h1>
    <p>
      <span class="keyword">Text</span> can be highlighted
      or <span class="highlight">colorized</span>.
    </p>
  </body>
</html>
```

The `head` contains a link to an **external style sheet** stored in the same directory as the page itself; we could use a URL here instead of a relative path, but the `link` element *must* have the `rel="stylesheet"` attribute. Inside the page, we then set the `class` attribute of each element we want to style.

The file `simple-style.css` looks like this:

```
h1.title {
  text-align: center;
}
span.keyword {
  font-weight: bold;
}
.highlight {
  color: coral;
}
```

Each entry has the form `tag.class` followed by a group of properties inside curly braces, and each property is a key-value pair. We can omit the class and just write (for example):

```
p {
  font-style: italic;
}
```

in which case the style applies to everything with that tag. If we do this, we can override general rules with specific ones: the style for a disclaimer paragraph is defined by `p` with overrides defined by `p.disclaimer`. We can also omit the tag and simply use `.class`, in which case every element with that class has that style.

As suggested by the earlier discussion of separators, elements may have multiple values for class, as in `<span class="keyword highlight">...</span>`. (The `span` element simply marks a region of text, but has no effect unless it's styled.)

These features are one (but unfortunately not the only) common source of confusion with CSS: if one may override general rules with specific ones but also provide multiple values for class, how do we keep track of which rules will apply to an element with multiple classes? A detailed discussion of the order of precedence for CSS rules is outside the scope of this tutorial. We recommend that those likely to work often with stylesheets read (and consider bookmarking) this W3Schools page[1].

One other thing CSS can do is match specific elements. We can label particular elements uniquely within a page using the `id` attribute, then refer to those elements using `#name` as a **selector**. For example, if we create a page that gives two spans unique IDs:

```
<html>
  <head>
    <link rel="stylesheet" href="selector-style.css" />
  </head>
```

[1] https://www.w3schools.com/css/css_specificity.asp

```
  <body>
    <p>
      First <span id="major">keyword</span>.
    </p>
    <p>
      Full <span id="minor">explanation</span>.
    </p>
  </body>
</html>
```

then we can style those spans like this:

```
#major {
  text-decoration: underline red;
}
#minor {
  text-decoration: overline blue;
}
```

Internal Links

We can link to an element in a page using `#name` *inside the link's* `href`*: for example,* `<a href="page.html#place">text</a>` *refers to the* `#place` *element in* `page.html`*. This is particularly useful* within *pages:* `<a href="#place">jump</a>` *takes us straight to the* `#place` *element within this page. Internal links like this are often used for cross-referencing and to create a table of contents.*

5.10 BOOTSTRAP

CSS can become very complicated very quickly, so most people use a framework to take care of the details. One of the most popular is Bootstrap[2] (which is what we use to style our website). Here's the entire source of a page that uses Bootstrap to create a two-column layout with a banner at the top (Figure 5.3):

```
<html>
  <head>
    <link rel="stylesheet"
          href="https://stackpath.bootstrapcdn.com/bootstrap/\
                4.1.3/css/bootstrap.min.css">
    <style>
      div {
        border: solid 1px;
      }
    </style>
  </head>
  <body>

    <div class="jumbotron text-center">
```

[2] https://getbootstrap.com/

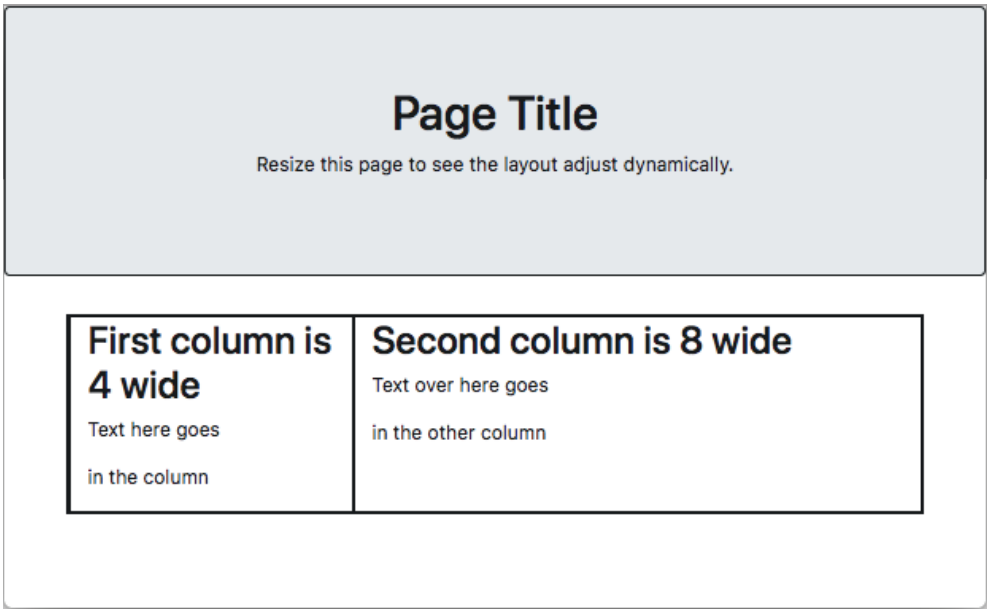


Figure 5.3: Bootstrap Layout

```
    <h1>Page Title</h1>
    <p>Resize this page to see the layout adjust dynamically.</p>
  </div>

  <div class="container">
    <div class="row">
      <div class="col-sm-4">
        <h2>First column is 4 wide</h2>
        <p>Text here goes</p>
        <p>in the column</p>
      </div>
      <div class="col-sm-8">
        <h2>Second column is 8 wide</h2>
        <p>Text over here goes</p>
        <p>in the other column</p>
      </div>
    </div>
  </div>

</body>
</html>
```

The page opens by loading Bootstrap from the web; we can also download `bootstrap.min.css` and refer to it with a local path. (The `.min` in the file's name signals that the file has been **minimized** so that it will load more quickly.)

The page then uses a `style` element to create an **internal style sheet** to put a solid one-pixel border around every `div` so that we can see the regions of the page more clearly. Defining styles in the page header is generally a bad idea, but it's a good way to test things quickly. Oh, and a `div` just marks a region of a page without doing anything to it, just as a `span` marks a region of text without changing its appearance

unless we apply a style.

The first `div` creates a header box (called a "jumbotron") and centers its text. The second `div` is a container, which creates a bit of margin on the left and right sides of our content. Inside that container is a row with two columns, one 4/12 as wide as the row and the other 8/12 as wide. (Bootstrap uses a 12-column system because 12 has lots of divisors.)

Bootstrap is **responsive**: elements change size as the page grows narrower, and are then stacked when the screen becomes too small to display them side by side.

We've left out many other aspects of HTML and CSS as well, such as figure captions, multi-column table cells, and why it's so hard to center text vertically within a `div`. One thing we will return to in Chapter 10 is how to include interactive elements like buttons and forms in a page. Handling those is part of why JavaScript was invented in the first place, but we need more experience before tackling them.

5.11 EXERCISES

CUTTING CORNERS

What does your browser display if you forget to close a paragraph or list item tag like this:

```
<p>This paragraph starts but doesn't officially end.

<p>Another paragraph starts here but also doesn't end.

<ul>
  <li>First item in the list isn't closed.
  <li>Neither is the second.
</ul>
```

1. What happens if you don't close a `ul` or `ol` list?
2. Is that behavior consistent with what happens when you omit `</p>` or `</li>`?

MIX AND MATCH

1. Create a page that contains a 2x2 table, each cell of which has a three-item bullet-point list. How can you reduce the indentation of the list items within their cells using CSS?
2. Open your page in a different browser (e.g., Firefox or Edge). Do they display your indented lists consistently?
3. Why do programs behave inconsistently? Why do programmers do this to us? Why? Why why why why why?

NAMING

What does the `sm` in Bootstrap's `col-sm-4` and `col-sm-8` stand for? What other options could you use instead? Why do web developers still use FORTRAN-style names in the 21st Century?

COLOR

HTML and CSS define names for a small number of colors. All other colors must be specified using **RGB** values. Write a small JavaScript program that creates an HTML page that displays the word `color` in 100 different randomly-generated colors. Compare this to the color scheme used in your departmental website. Which one hurts your eyes less?

UNITS

What different units can you use to specify text size in CSS? What do they mean? What does *anything* mean, when you get right down to it?

KEY POINTS

- HTML is the latest in a long line of markup languages.
- HTML documents contain elements (represented by tags in angle brackets) and text.
- Elements must be strictly nested.
- Elements can contain attributes.
- Use escape sequences beginning with ampersand to represent special characters.
- Every page should have one `html` element containing a `head` and a `body`.
- Use `<!--...-->` to include comments in HTML.
- Use `ul` and `ol` for unordered and ordered lists, and `li` for list elements.
- Use `table` for tables, `tr` for rows, `th` for headings, and `td` for regular data.
- Use `<a href="...">...</a>` to create links.
- Use `<img src="..." title="..." alt="..." />` to include images.
- Use CSS to define appearance of elements.
- Use `class` and `id` to identify elements.
- Use selectors to specify the elements that CSS applies to.

CHAPTER

4

DESIGNING GRAPHICALLY WITH THE LANGUAGE OF CODE

Programming Media Art Using Processing
A Beginner's Guide
By Margaret Noble

Learn more

CHAPTER 1

Designing Graphically with the Language of Code

GETTING STARTED & BASIC OVERVIEW

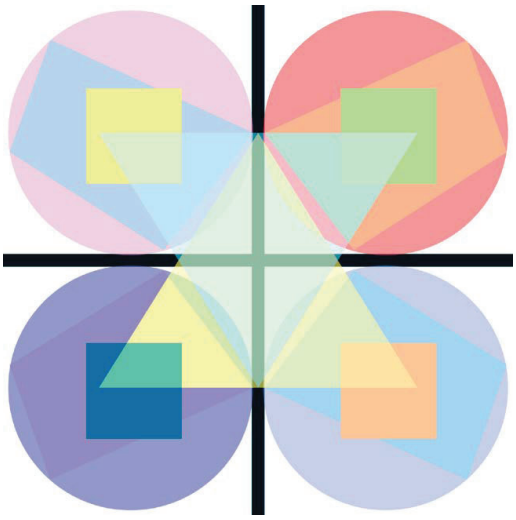


FIGURE 1.1 Student project example: geometric graphic design. (Printed with permission from Michel Yanez.)

Install Processing for free on your computer; it works on Macs, PCs, and Linux.

Link: https://processing.org/download/

FIGURE 1.2

Once on the Processing web page, scroll down to the list of "stable releases" and try to install the most recent version of Processing on your computer. If you have difficulty installing this version, then try one of the earlier releases. The projects in this book will work fine across the various versions of Processing. Once installed, open Processing and press the play button. You should see something like this.

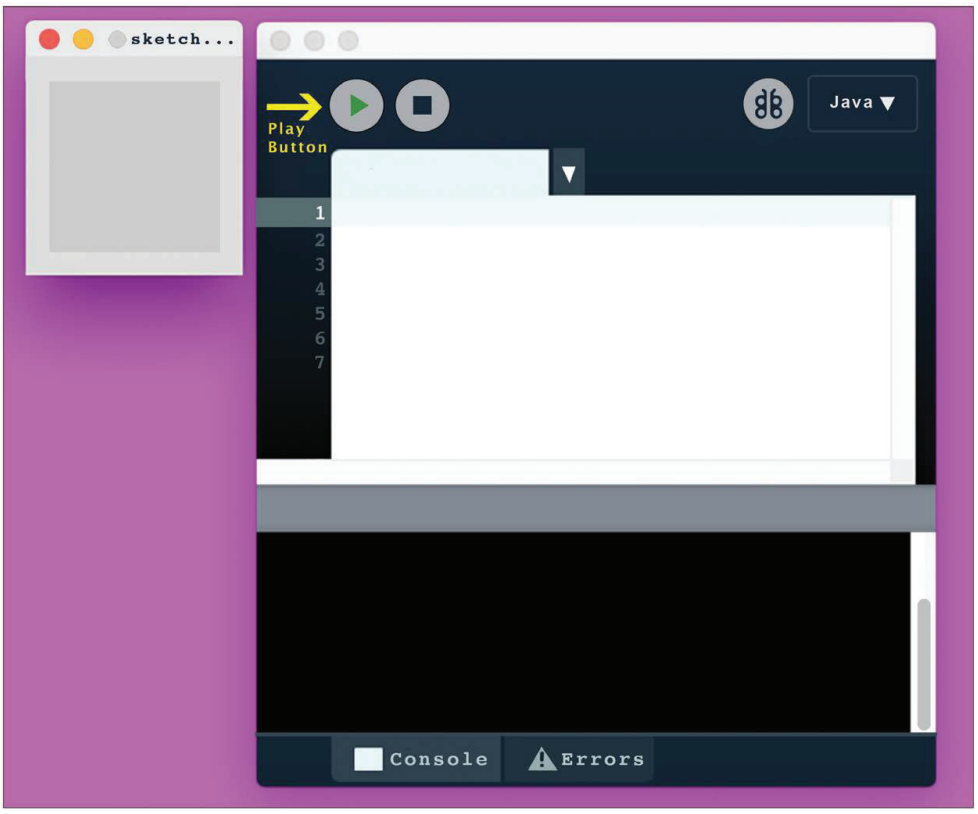


FIGURE 1.3

As shown in Figure 1.3, your Processing workspace has two windows. You enter text commands into the editor window and see your graphic results on the canvas window (also called the sketch window). You push the play button to test for results whenever you change the code.

As you move through these tutorials, it is advised that you save all of your exercises and example files for future reference.

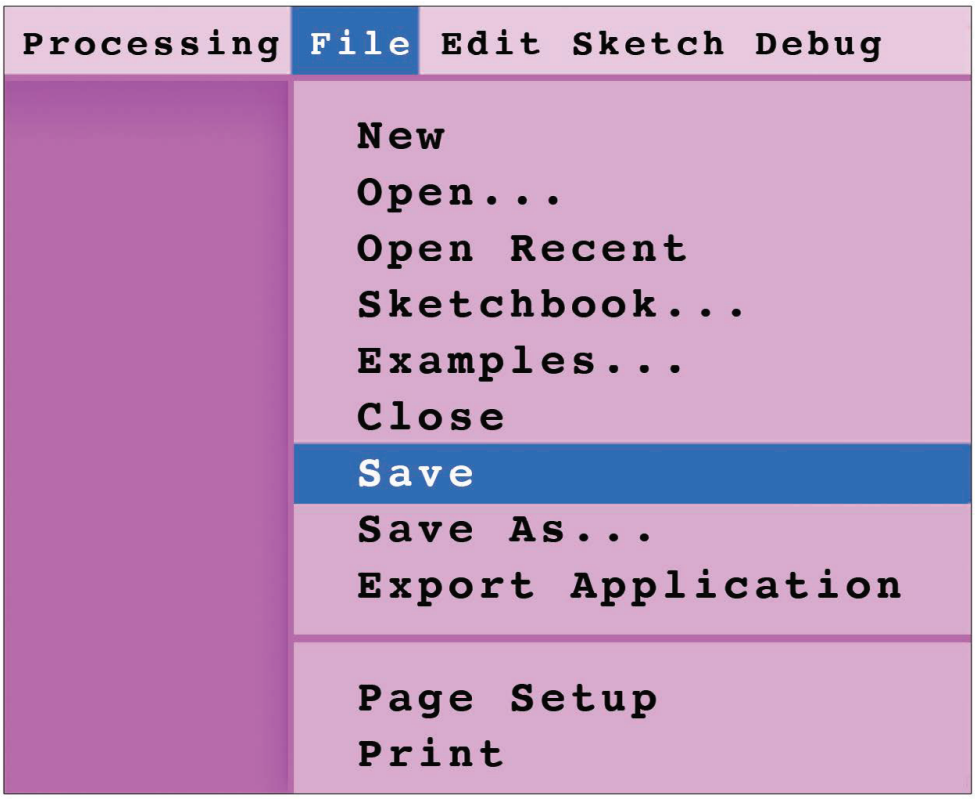


FIGURE 1.4

When you save a Processing file, you will notice that Processing automatically places a .pde file inside of a folder.

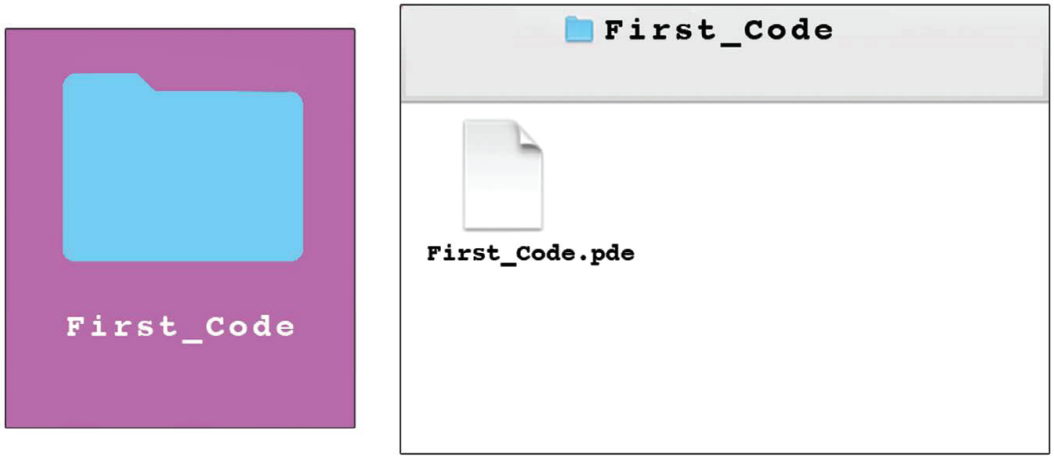


FIGURE 1.5

It is important that you keep this folder with the .pde file inside of it. Processing also requires that the folder and .pde file have the same name. When you have mismatched names between a file and a folder or separate this file path, you will get errors when opening your project.

LESSON 1.1: PIXEL GRID SYSTEM

Processing uses a pixel grid system for plotting shapes on the canvas window. Every point on the screen is a pixel and each point is specified by the locations of x, y (horizontal and vertical placements).

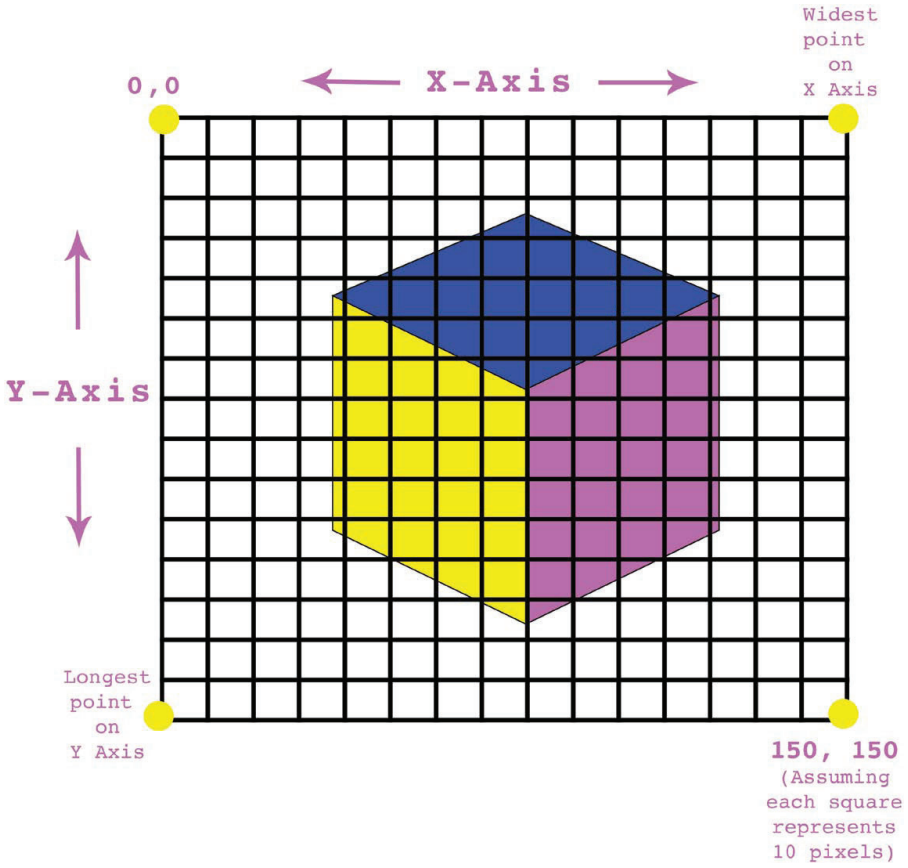


FIGURE 1.6 The longest point on the canvas is also called the height of the canvas.

The pixel grid system has its 0,0 coordinate in the *upper left corner* (this is different from beginning Algebra). The x-axis moves from left to right (0 to the specified width of your canvas). The y-axis moves from the top to the bottom, (0 to the specified height of your canvas.) This can be confusing because the "highest" point of your canvas is at the bottom of your grid.

Exercise 1.1

Grab a piece of paper and draw a quick sketch of a Processing canvas that is 100 × 100 pixels. Label the following coordinates on your paper sketch:

TABLE 1.1 Coordinates for Exercise 1.1

X	Y
0	0
100	100
50	50
0	100
100	0

LESSON 1.2: CODE AND CANVAS WINDOWS

By default, Processing opens the canvas window in the size of 100 pixels by 100 pixels. You can specify your preferred canvas dimensions by typing the command **size()** into the editor window and specifying how many pixels you want for its width and height inside of the parentheses.

```
1 size (400,200);
```

FIGURE 1.7

Important: Details matter! If the line of code you are typing uses punctuations such as parentheses, commas, brackets, and/or semicolons, then copy these details exactly.

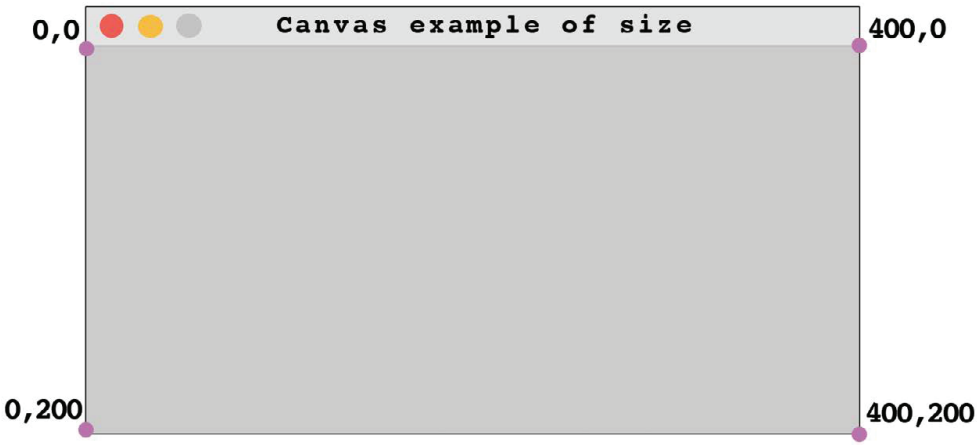


FIGURE 1.8 (Canvas size is 400 × 200).

You must also specify the background color of your canvas or it will default to gray. For now, we will use 255, which is white. More on color options soon.

```
1 size (400,200);
2 background (255);
```

FIGURE 1.9

LESSON 1.3: LINES, WIDTH, AND HEIGHT

Try this code:

```
1 size (200,100);
2 background (255);
3 line (0,0,200,100);
```

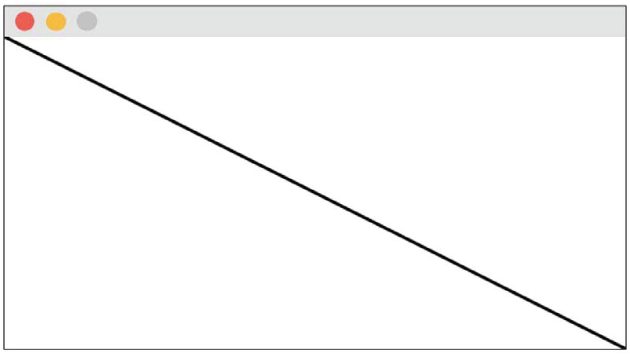


FIGURE 1.10

The words in blue are called functions (or commands). After the blue functions, the numbers inside of the parentheses are called arguments. "Arguments" are the parameters of a function(). Arguments specify the details Processing needs to draw shapes and colors in the canvas window.

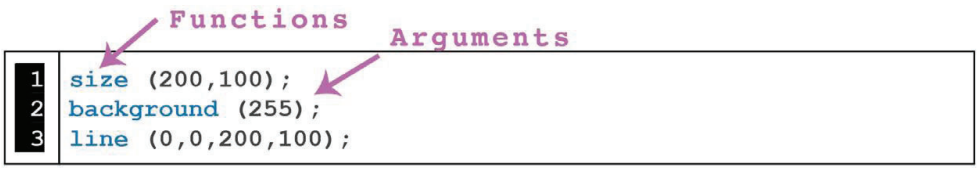


FIGURE 1.11

This is how a line works…

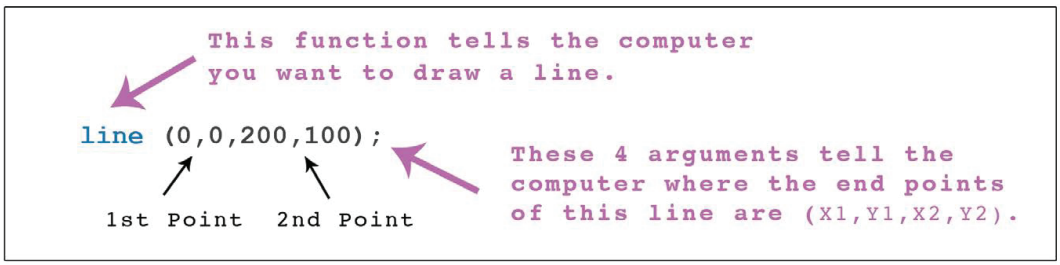


FIGURE 1.12

Try the following code to make a different **line()**. Take note of how the arguments describe each end point of the **line()**.

```
1 size (200,100);
2 background (255);
3 line (100,0,100,100);
```

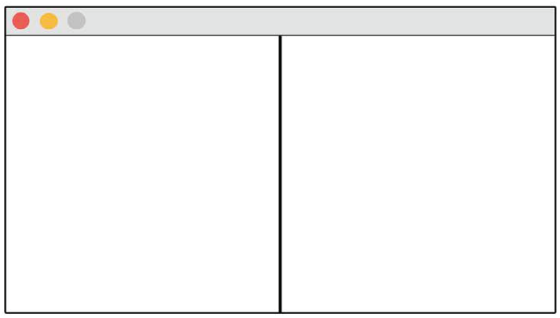


FIGURE 1.13

Try another:

```
1 size (200,100);
2 background (255);
3 line (0,50,200,50);
```

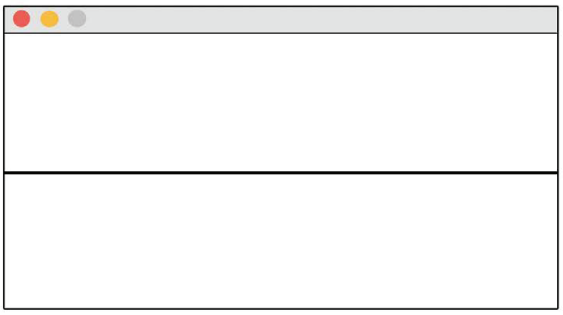


FIGURE 1.14

Make this line:

```
1 size (200,100);
2 background (255);
3 line (0,0,200,100);
```

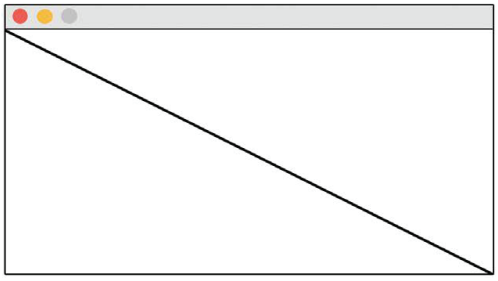


FIGURE 1.15

Now replace the line's third and fourth arguments the words, "width" and "height" – like this:

```
1 size (200,100);
2 background (255);
3 line (0,0,width,height);
```

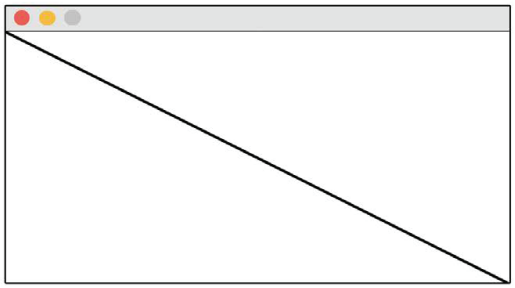


FIGURE 1.16

As you can see, the code in Figures 1.15 and 1.16 give you the same results. What do you think **width** and **height** refer to? In Processing, **width** and **height** refer back to the **size()** of the canvas. They are said to be "built-in variables" and they fluctuate in value depending on the canvas size.

```
1 size (200,100);
2 background (255);
3 line (0,0,width,height);
```

FIGURE 1.17 (In this case, **width** = 200, **height** = 100).

```
1 size (125,150);
2 background (255);
3 line (0,height,width,0);
```

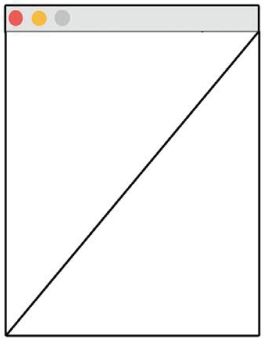


FIGURE 1.18 (In this case, **width** = 125, **height** = 150).

You can also manipulate the values of **width** and **height** with math operations.

```
1 size (200,100);
2 background (255);
3 line (width/2,50,200,height);
```

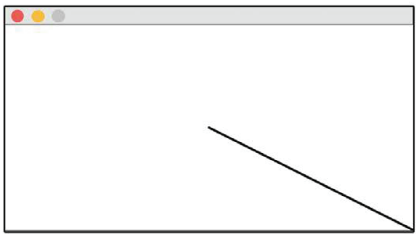


FIGURE 1.19 (**Width** = 100 (200 divided by 2) and **height** = 100).

And so on…

```
1 size (200,100);
2 background (255);
3 line (width/3,10,width/2,height);
```

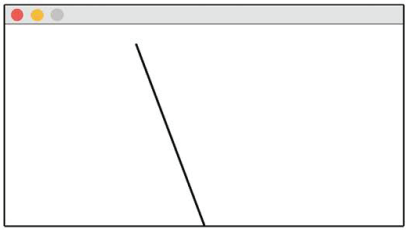


FIGURE 1.20 (**Width** = 66 (200 divided by 3, rounded up) and **height** = 100).

TABLE 1.2 Math Symbols Used in Processing

Application	**Symbol**
division	/
multiplication	*
addition	+
subtraction	-

There are many possible mathematical applications for designing visual art in code and **width** and **height** are very handy for thinking out graphical layouts. You can use **width** and **height** in the same program as many times as you like and in almost any numerical argument.

Exercise 1.2

Code the following picture with a 300 × 300 pixel canvas size using the built-in variables **width** and **height** in some of the **line()** arguments. It might be easiest to first code the lines with numerical arguments and then replace these values with **width** and **height** where appropriate.

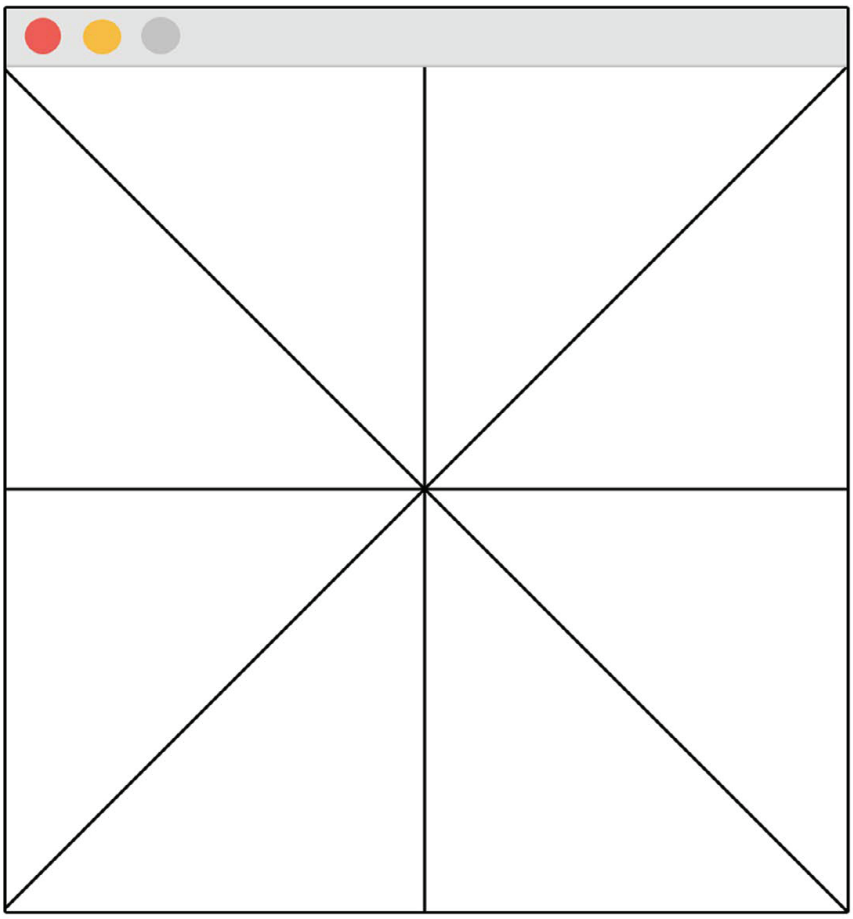


FIGURE 1.21

LESSON 1.4: MORE SHAPES

It is time to add to our graphic design tool kit as there are many shape functions we can use. Like the **line()** function, the arguments of other shape functions control different parameters. See if you can guess what the arguments control for the following shapes?

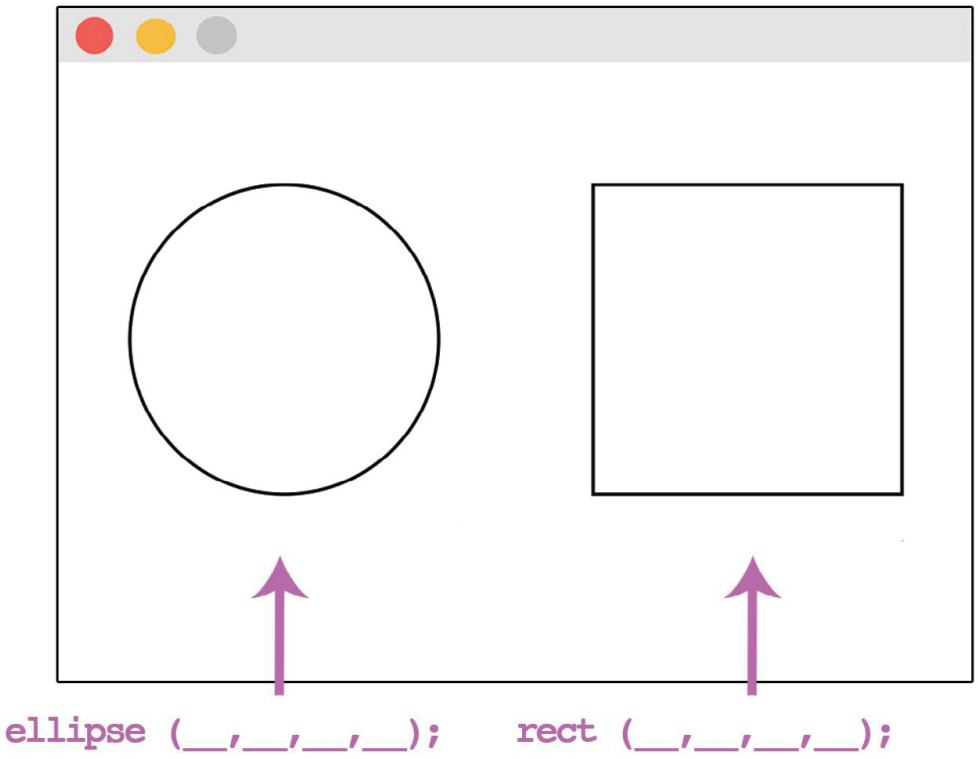


FIGURE 1.22

Answers:

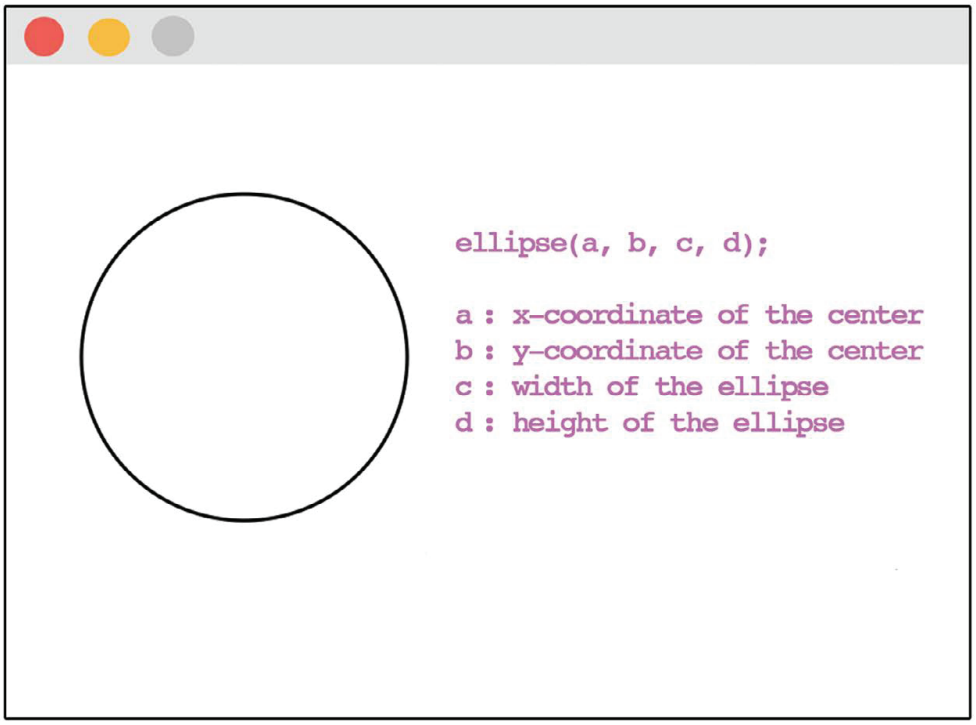


FIGURE 1.23

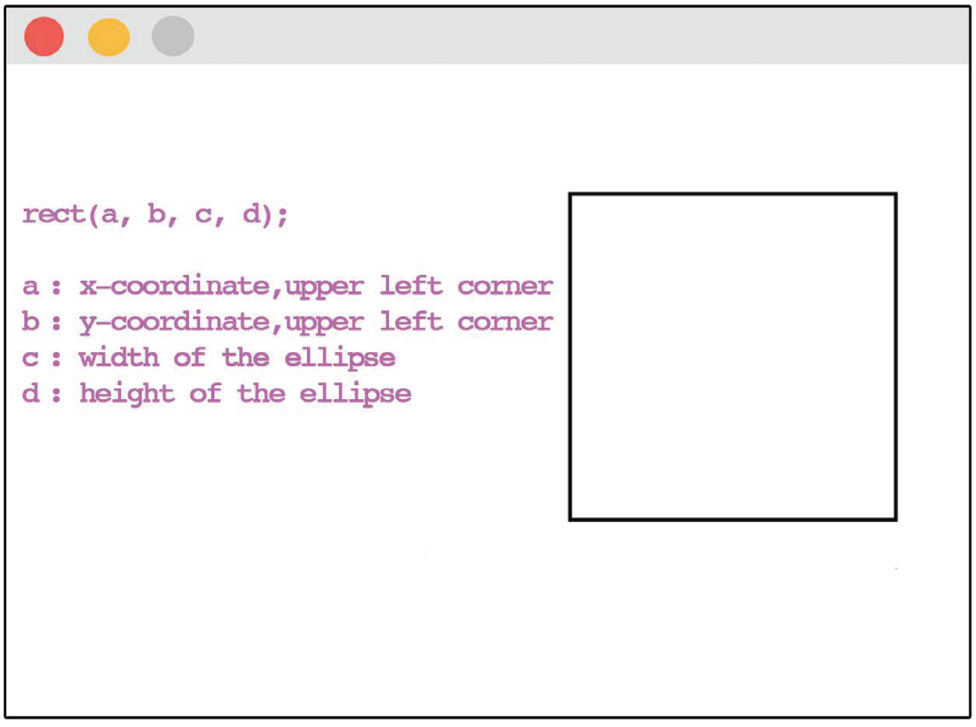


FIGURE 1.24

Play around with the following code to verify your understanding:

```
1 size (300,200);
2 background (255);
3 ellipse (75,100,100,100);
4 rect (175,50,100,100);
```

FIGURE 1.25

Tip: It is sometimes confusing that the x,y anchor points for the **rect()** are positioned in the upper left corner rather than in the center like the **ellipse()**. If you prefer to have your rectangle anchor points in the center, then you can add the following command before your rectangles:

rectMode (CENTER);

Exercise 1.3

Code the following two ellipses, two lines, and one rectangle on a 300 × 200 pixel sized canvas.

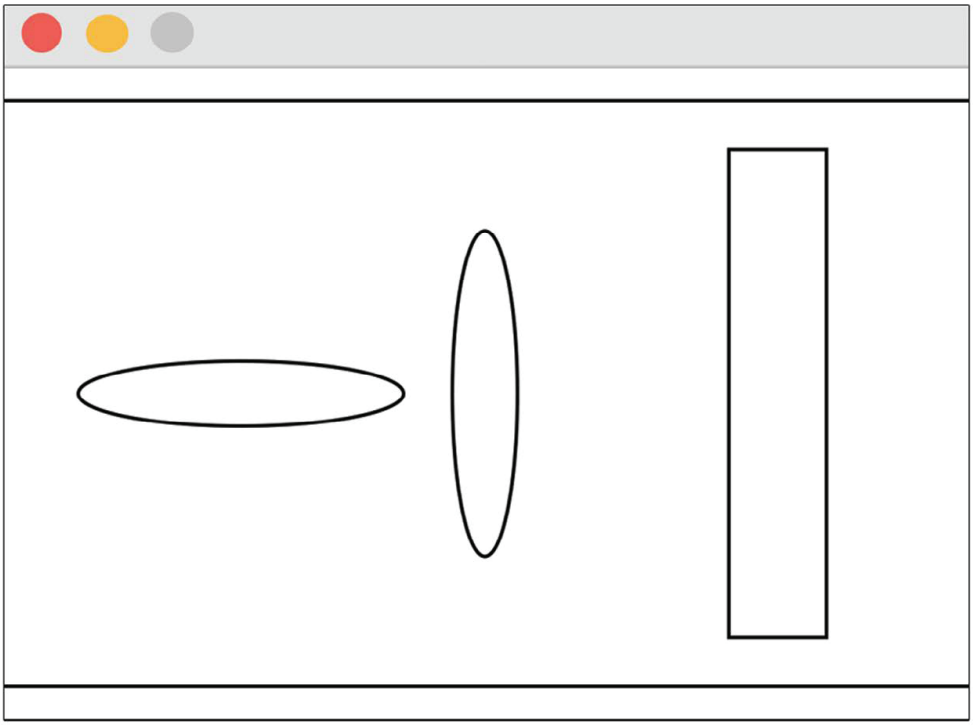


FIGURE 1.26

LESSON 1.5: GRAYSCALE

In Processing, by default lines and outlines are black, while shapes are filled white. But you can designate between grayscale tones very simply by specifying an argument of 0 (the darkest black) all the way up to 255 (the lightest white).

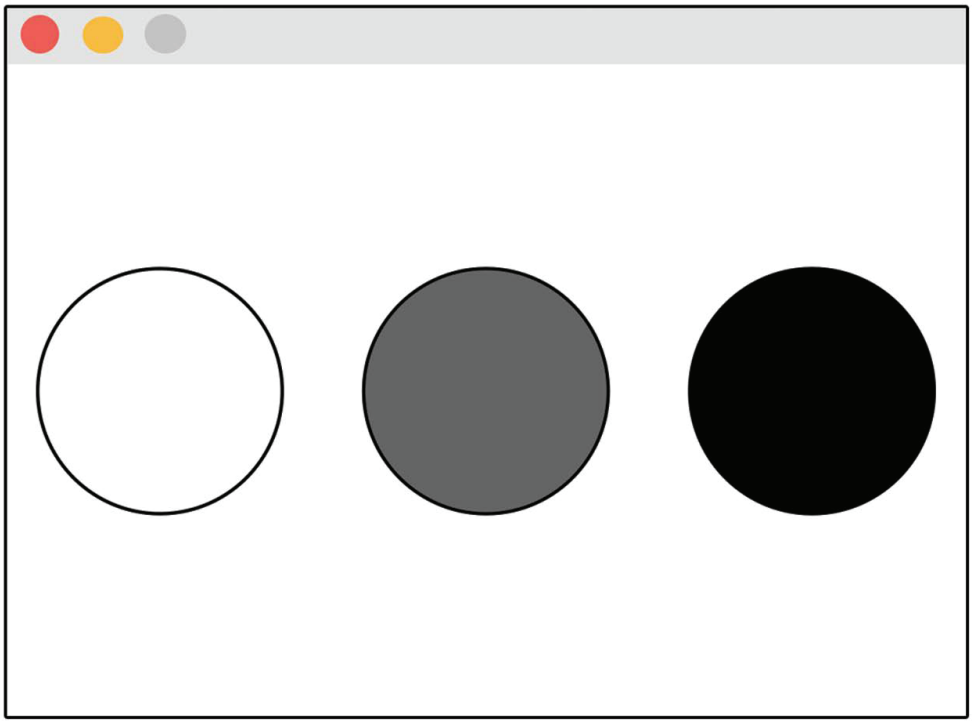


FIGURE 1.27

TABLE 1.3 Black and White Controls in Processing

In Processing	Description
fill()	Fills in the color of shapes.
stroke()	Colors the outlines of shapes and lines.
background()	Colors the background of the canvas.
0 - 255	Grayscale numerical range. 0 is black, 255 is white and all shades of gray are in between.

Exercise 1.4

Use the following starter code and complete the design pictured.

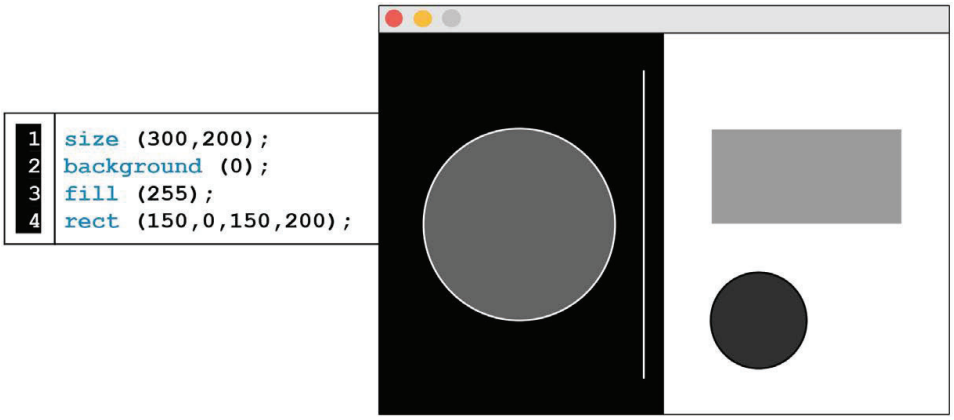


FIGURE 1.28

LESSON 1.6: SYNTAX, COMMENTS, AND ORDER OF CODE

Syntax

When you program the computer, you need to speak in a language that Processing understands. The syntax of a computer language is the set of rules that define the proper order and combinations of letters, numbers, and symbols. The syntax rules must be strictly followed or the computer won't be able to process your code. If you misspell words, miss punctuation, or misuse capitalizations, then your program will have errors.

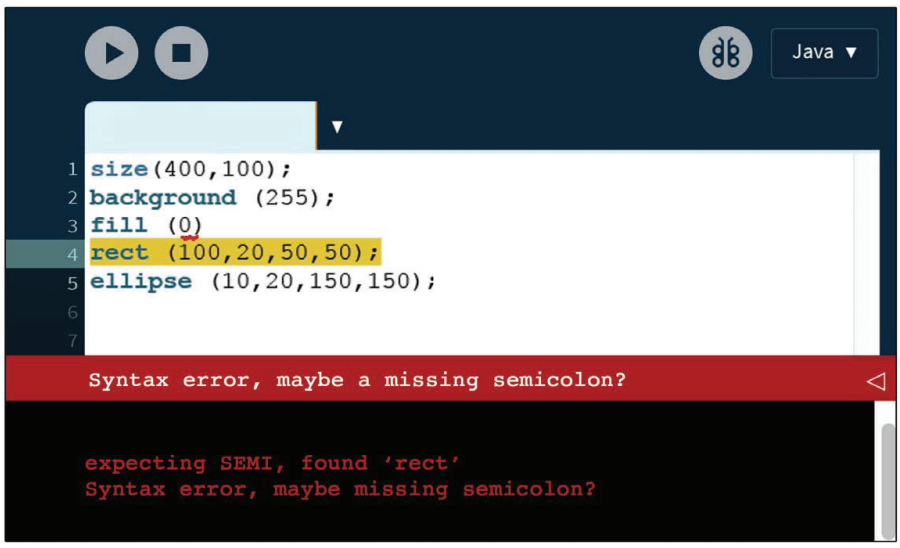


FIGURE 1.29

Processing gives you feedback on broken code in a message area below the editor. Usually, the highlighted line of code is not the problematic line. The malfunctioning line of code is often *before* the highlighted line. In Figure 1.29, the **fill**(0) is missing a semicolon.

Comments

Comments are annotations in the code to help you stay organized in a sea of text. We use two forward slashes//to indicate a comment. The two forward slashes can also be used to turn off a line of code. This is really helpful when you need to troubleshoot a problem with your program.

By isolating different lines of code, it is easier to pinpoint where issues might be.

Forward slashes used to create comments:

```
1 size (400,100);     // size of canvas
2 background (255); // sets background to white
3 stroke (0);       // sets outline color to black
4 fill (150);      // sets color shape fill to gray
```

FIGURE 1.30

Forward slashes to turn off individual lines of code commands. In Figure 1.31 below, **stroke**() and **fill**() are not read by the computer.

```
1 size (400,100);
2 background (255);
3 //stroke (0);
4 //fill (150);
```

FIGURE 1.31

There is a shorthand for turning off large chunks of code. Using /* at the beginning of the section and */ after the last line you want to turn off will render everything in between as unreadable by Processing.

```
/*
size (400,100);
background (255);
fill (0);
rect (100,20,50,50);
ellipse (10,20,150,150);
line (0,0, width, height);
rect (10,20,150,60);
ellipse (30,25,10,15);
line (0,0,width/2, height/2);
*/
```

FIGURE 1.32

Order of Code

The order of code commands impacts how the code will run. Code lines that start at the top will impact the lines of code that come after. On the left, you see that the first gray **fill()** command is overridden by a second black **fill()** command coloring both shapes black. On the right, you see that you need to specify a new color fill before the shapes you want to be in a different color.

```
size (200,100);
background (255);
rectMode ( CENTER );
fill (100);
fill (0);
rect (50,50,50,50);
ellipse (150,50,50,50);
```

```
size (200,100);
background (255);
rectMode ( CENTER );
fill (0);
rect (50,50,50,50);
fill (100);
ellipse (150,50,50,50);
```

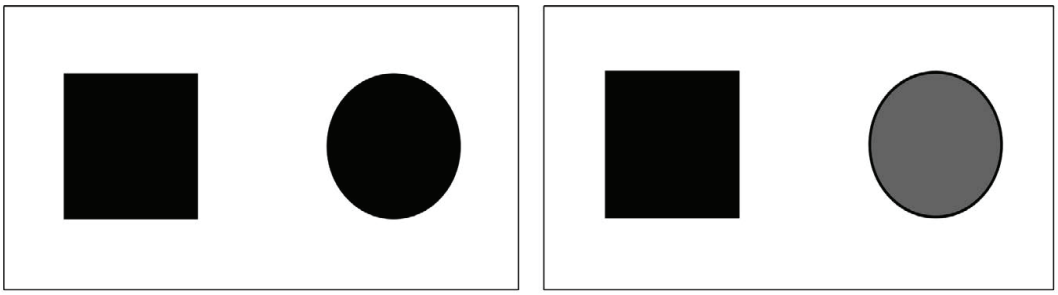


FIGURE 1.33

Exercise 1.5

Code the following on a 300 × 300 canvas.

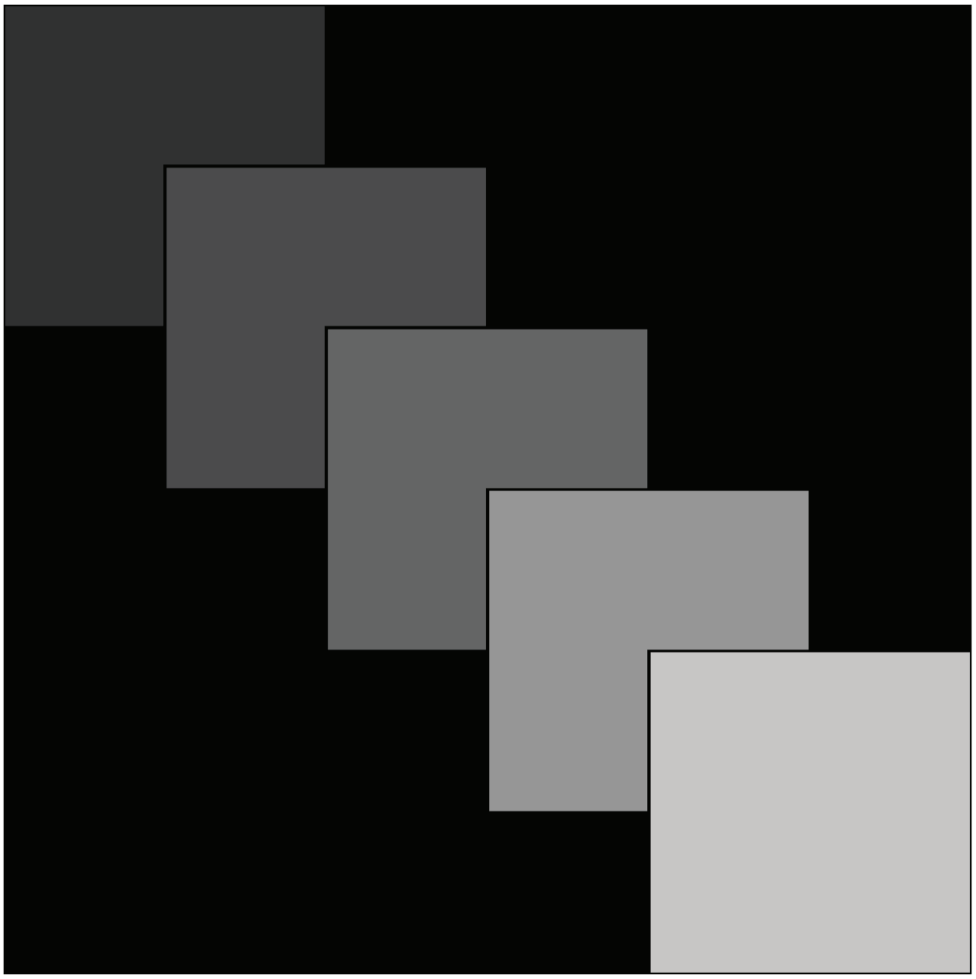


FIGURE 1.34

LESSON 1.7: LINE COMMANDS (STROKE AND NO STROKE)

In your designs, you may want to manipulate the look of lines and outlines. Here are two new commands for modifying lines in your projects.

TABLE 1.4 Line Commands

Command	Description
noStroke()	Removes outlines on shapes.
strokeWeight()	Changes the thickness of lines and shape outlines.

Exercise 1.6

Code the following image on a 200 × 200 pixel canvas and annotate your code with organizational comments.

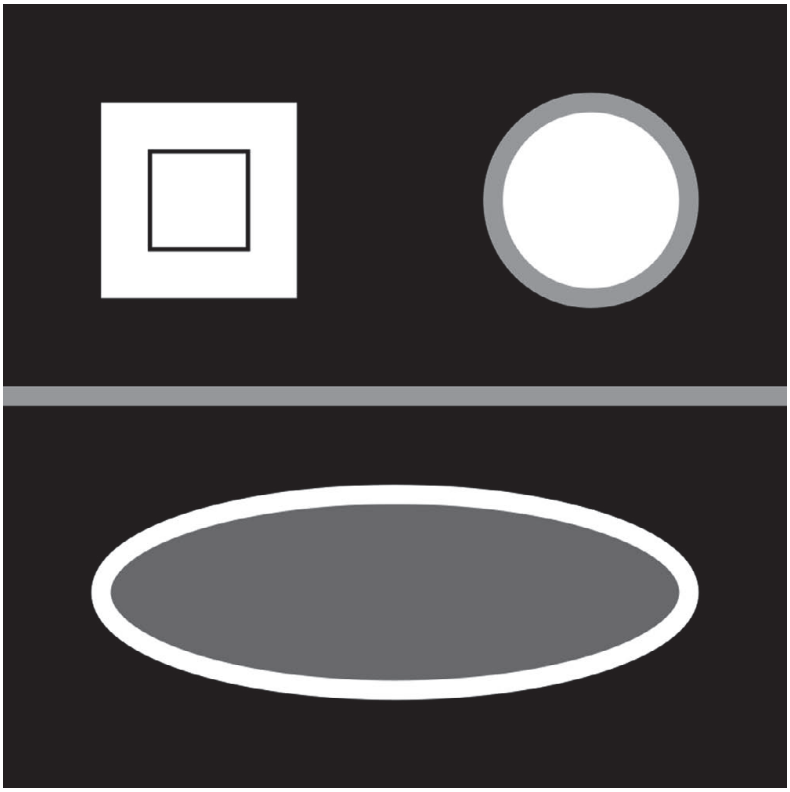
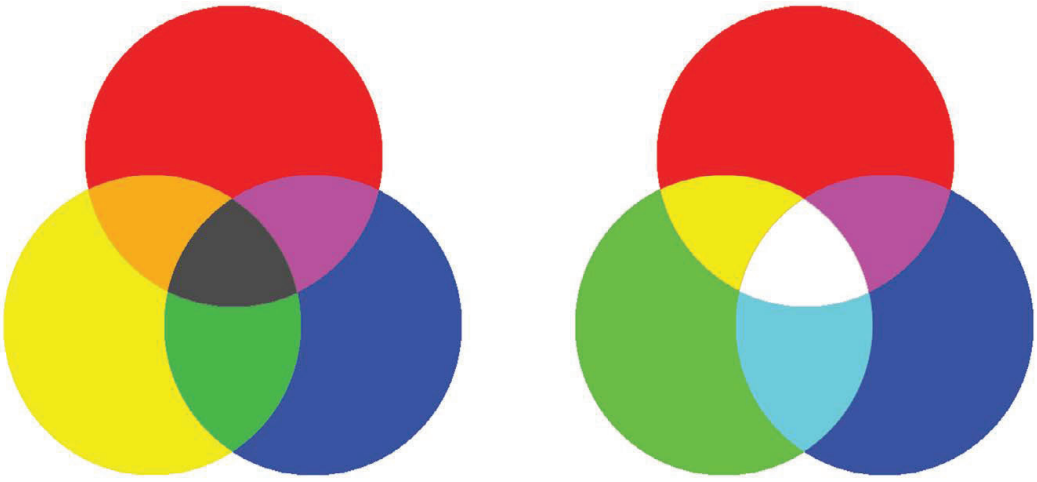


FIGURE 1.35

LESSON 1.8: COLORING PIXELS

In the art class, the primary colors were red, yellow, and blue. In the digital environment, our primary colors are red, green, and blue also known as RGB. These are the colors of light.



Primary analogue color mixing. **RGB digital color mixing.**

FIGURE 1.36

TABLE 1.5 Processing Color Use Overview

Description	Examples	
Color functions commonly used:	**fill()** **stroke()**	//colors the inside of shapes //colors lines and outlines

Code functions using R,G,B color have 3 arguments. One argument for each color (red, green, and blue).	**fill** (255, 0, 0) ;
	stroke (0, 255, 0) ;
	fill (0, 0, 255) ;
		// red
		// green
		// blue

Code functions using grayscale color have 1 argument.	**stroke** (0); **fill** (255):	// black // white
RGB color values range from 0 to 255. 0 is the darkest and 255 is the lightest.	**stroke** (0, 25, 0); **fill** (0, 0, 255);	//dark green // bright blue
Colors that are not primary red, green, and blue are made by various mixes of these 3.	**fill** (150, 0,100); **stroke** (173, 201, 20);	//dark wine color // lime green color

Play with this code to see R, G, B (red, green, blue) arguments in action.

```
1  size (400,200);
2  background (255,0,0);
3  rectMode ( CENTER );
4  fill (0,0,255);
5  rect (200,100,200,80);
6  stroke (255,255,0);
7  strokeWeight (10);
8  fill (0,255,0);
9  ellipse (200,100,300,25);
10 stroke (255);
11 line (0,0,400,200);
12 stroke (0);
13 line (400,0,0,200);
```

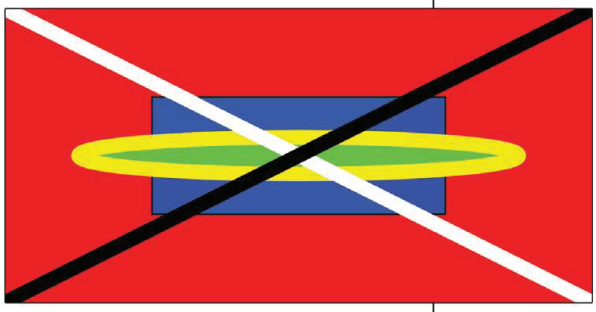


FIGURE 1.37

Tip: Processing has a color selector that gives you the exact RGB mix for a specified color.

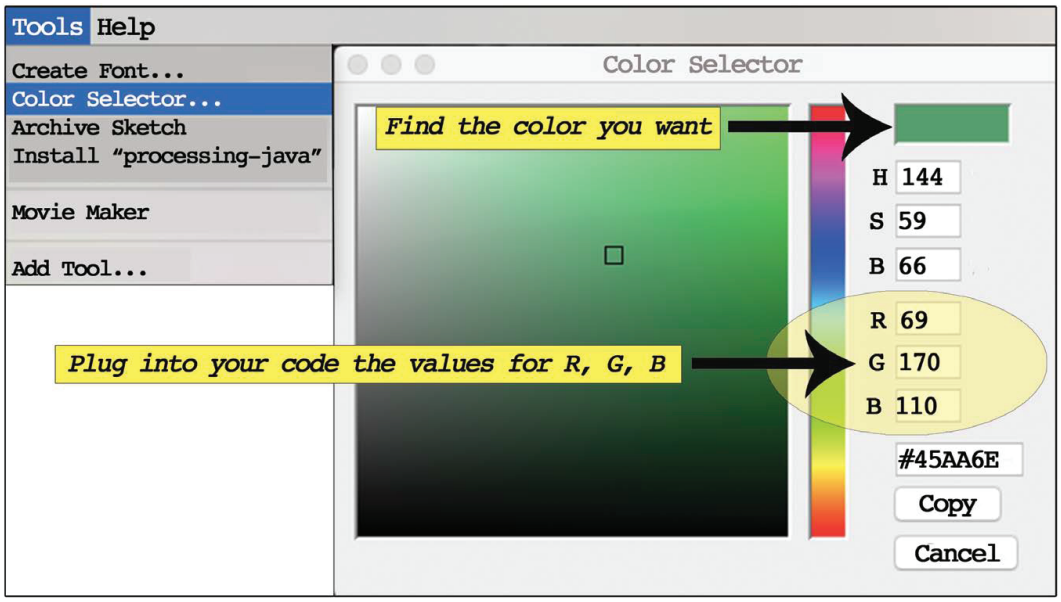


FIGURE 1.38

Exercise 1.7

Code the following sketch on a 300 × 300 canvas.

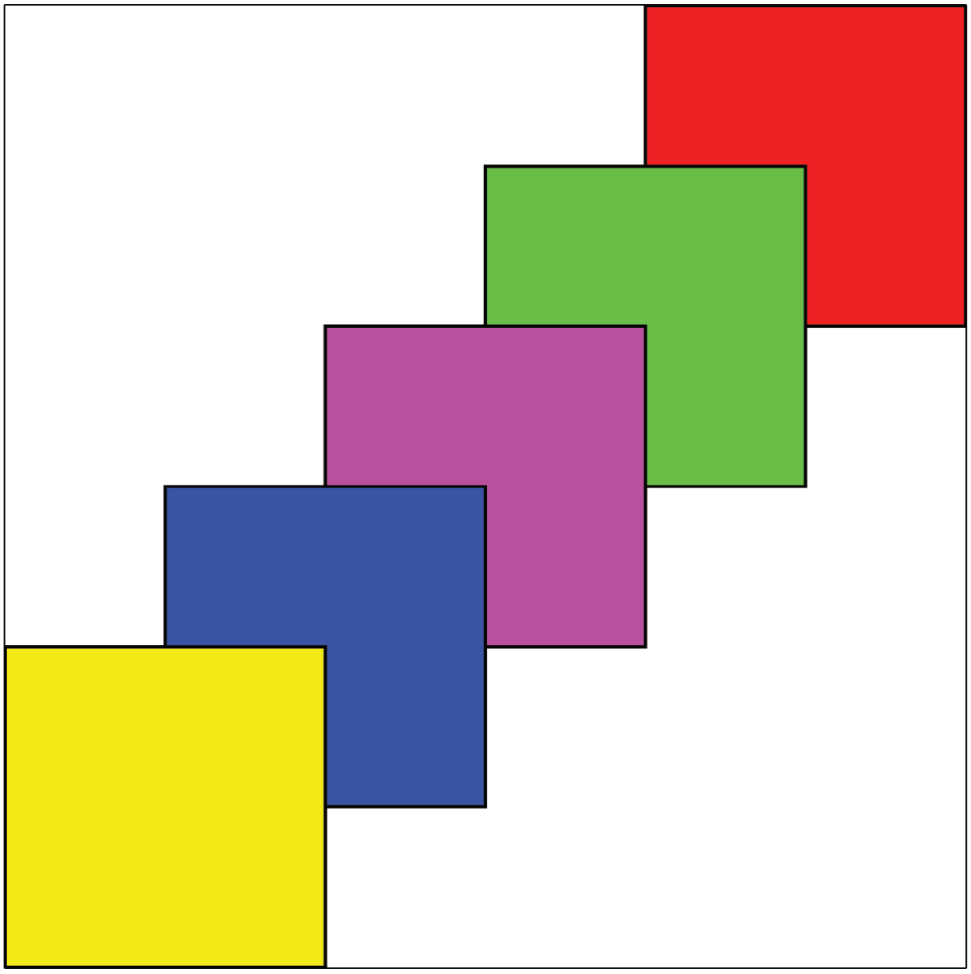


FIGURE 1.39

LESSON 1.9: ADDING TRANSPARENCY VALUES

Some really nice results can be achieved by adding transparency to your designs. For color, add a fourth argument to your R,G,B formulation to specify opacity. For grayscale, add a second argument. The number range for opacity is 0 to 255. Play with the following program to see how transparency works.

```
1  size (200,100);
2  background (255); //solid white background
3  rectMode (CENTER);
4
5  fill (255,0,0,100); //red shape fill w/transparency
6  rect (100, 50, 75, 75);
7
8  strokeWeight (20);
9  stroke (0, 50); //black line/outline stroke w/transparency
10 line (100,0, 100, height);
11
12 fill (0,0,255,175); //blue shape fill w/transparency
13 ellipse (100,50,190,20);
14
15 noStroke ();
16 fill (0,255,0); //green shape fill no transparency
17 ellipse (100,50,25,25);
```

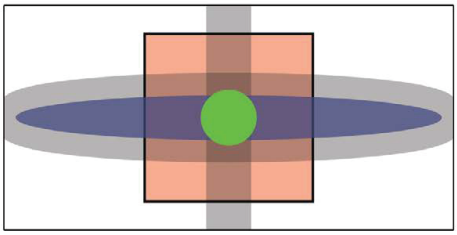


FIGURE 1.40

Exercise 1.8

Type the following starter code:

```
1 size (400,250);
2 background (255);
3 fill (0,0,255,150);
4 ellipse (50,50,100,100);
5 fill (255,0,0,150);
6 ellipse (100,100,100,100);
```

FIGURE 1.41

Next, add four more circles to match the following design:

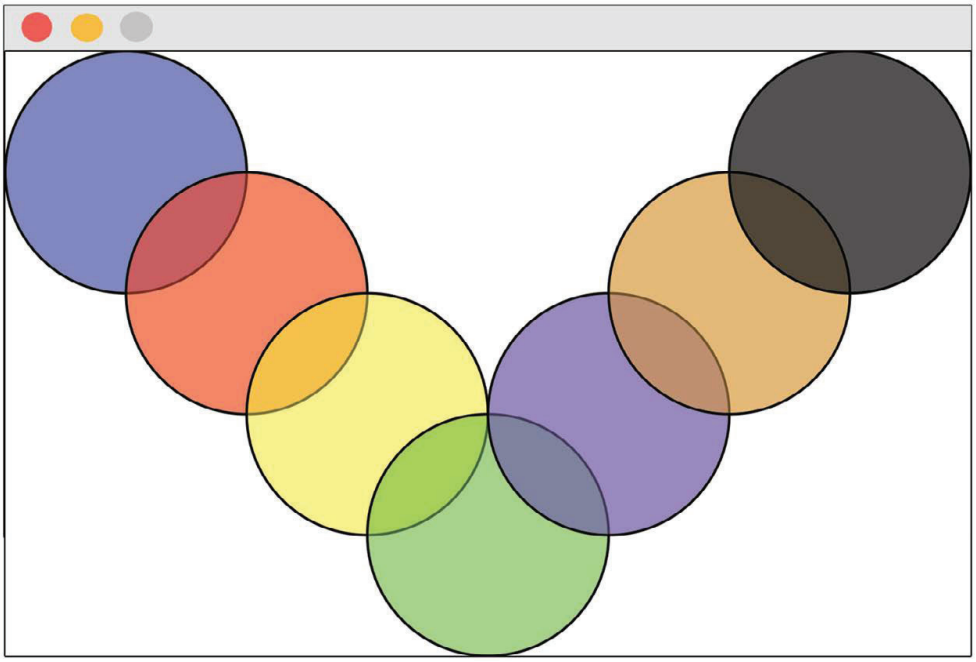


FIGURE 1.42

LESSON 1.10: THE PROCESSING REFERENCE – IMPORTANT RESOURCE!

The reference is a comprehensive, online library of code commands available for use in Processing. It is an important place to figure out things not covered in this book and will also expand your knowledge of things you already know. For the purposes of this chapter, we are going to use the reference to learn about more 2D shapes.

Go to https://processing.org/reference/ to find the following page.

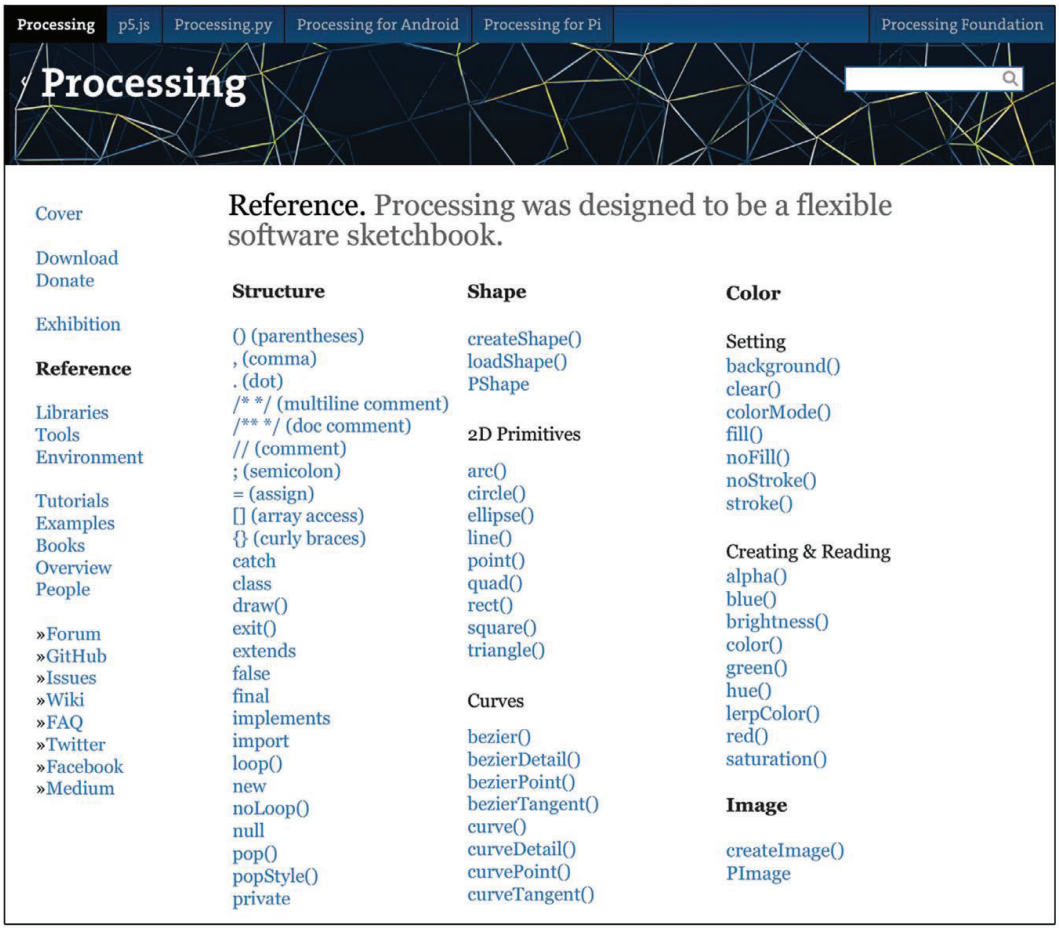


FIGURE 1.43

First, navigate to the "2D Primitives" category. Here you will find many more possible shapes available for your projects. Next, click on the **quad()** function to open its dedicated page. The following diagram uses the **quad()** entry as an example on how to navigate the reference pages.

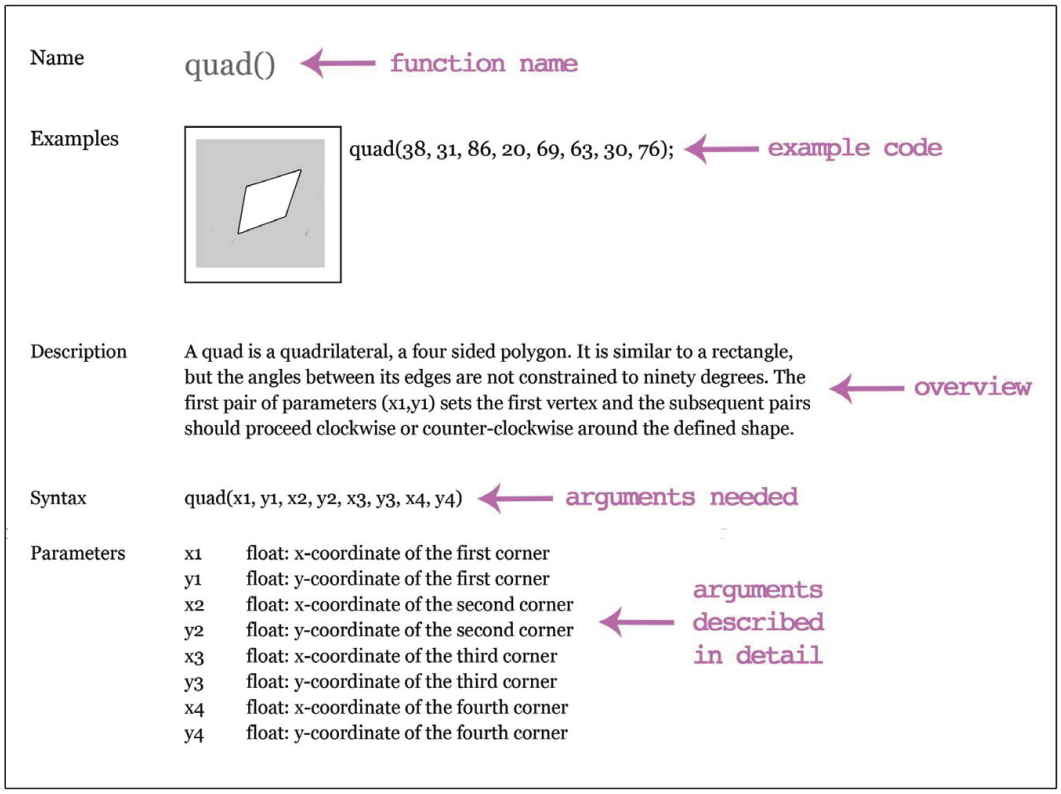


FIGURE 1.44

PROJECT: GEOMETRIC DESIGN

Code a geometrically patterned design with a variety of different shapes using the colors of your choice. Aim to go beyond lines, rects, and ellipses with new 2D shapes from the Processing reference. For project examples, see the download folder available from the publisher's website.

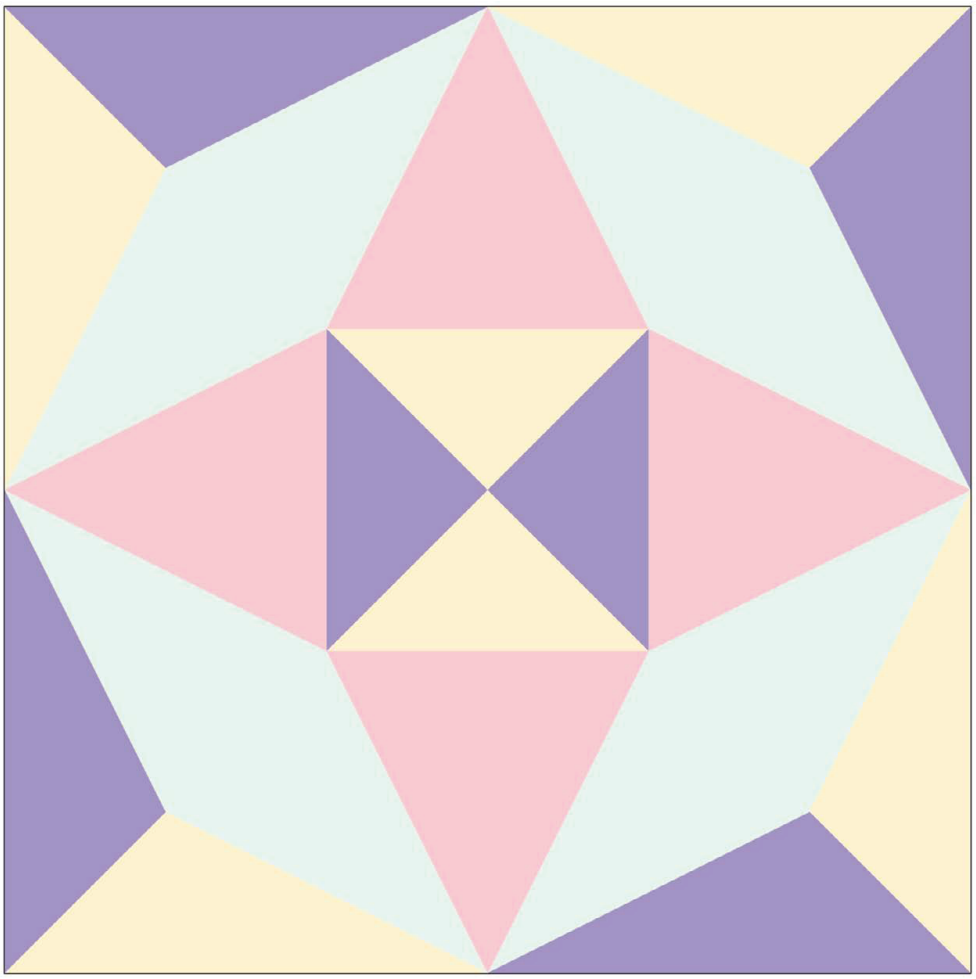
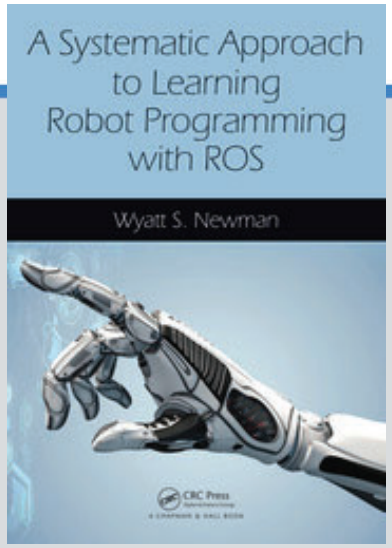


FIGURE 1.45 Student project example: geometric graphic design. (Printed with permission from Flavia Huerta.)

CHAPTER

5

INTRODUCTION TO ROS: ROS TOOLS AND NODES



A Systematic Approach to Learning Robot Programming with ROS

By Wyatt Newman

Learn more

Introduction to ROS: ROS tools and nodes

CONTENTS

1.1 Some ROS concepts 5
1.2 Writing ROS nodes 8
1.2.1 Creating ROS packages 9
1.2.2 Writing a minimal ROS publisher 11
1.2.3 Compiling ROS nodes 14
1.2.4 Running ROS nodes 15
1.2.5 Examining running minimal publisher node 16
1.2.6 Scheduling node timing 18
1.2.7 Writing a minimal ROS subscriber 20
1.2.8 Compiling and running minimal subscriber 22
1.2.9 Minimal subscriber and publisher node summary 23
1.3 More ROS tools: `catkin_ simple`, `roslaunch`, `rqt_ console`, and `rosbag` 24
1.3.1 Simplifying `CMakeLists.txt` with `catkin_ simple` 24
1.3.2 Automating starting multiple nodes 26
1.3.3 Viewing output in a ROS console 27
1.3.4 Recording and playing back data with `rosbag` 28
1.4 Minimal simulator and controller example 30
1.5 Wrap-up 35

INTRODUCTION

This introductory chapter will focus on the concept of `nodes` in ROS, starting with minimal examples. A few ROS tools will be introduced as well to help illuminate the behavior of ROS nodes. The simplest means of ROS communications—publish and subscribe—will be used here. Alternative communications means (`services` and `actionservers`) will be deferred until Chapter 2.

1.1 SOME ROS CONCEPTS

Communications among nodes are at the heart of ROS. A `node` is a ROS program that uses ROS's middleware for communications. A node can be launched independently of other nodes and in any order among launches of other nodes. Many nodes can run on the same computer, or nodes may be distributed across a network of computers. A node is useful only if it can communicate with other nodes and ultimately with sensors and actuators.

Communication among nodes uses the concepts of `messages`, `topics`, `roscore`, `publishers` and `subscribers`. (`Services` are also useful, and these are closely related to publishers and subscribers). All communications among nodes is serialized network communications. A `publisher` publishes a `message`, which is a packet of data that is interpretable using an associated key. Since each message is received as a stream of bits, it is necessary to consult the key (the message type description) to know how to parse the bits and reconstruct the corresponding data structure. A simple example of a message is `Float64`, which is defined in the package `std_msgs`, which comes with ROS. The message type helps the publisher pack a floating-point number into the defined stream of 64 bits, and it also helps the subscriber interpret how to unpack the bitstream as a representation of a floating-point number.

A more complex example of a message is a `twist`, which consists of multiple fields describing three-dimensional translational and rotational velocities. Some messages also accommodate optional extras, such as time stamps and message identifier codes.

When data is published by a publisher node, it is made available to any interested subscriber nodes. Subscriber nodes must be able to make connections to the published data. Often, the published data originates from different nodes. This can happen because these publishers have changed due to software evolution or because some publisher nodes are relevant in some contexts and other nodes in other contexts. For example, a publisher node responsible for commanding joint velocities may be a stiff position controller, but in other scenarios a compliant-motion controller may be needed. This hand-off can occur by changing the node publishing the velocity commands. This presents the problem that the subscriber does not know who is publishing its input. In fact, the need to know what node is publishing complicates construction of large systems. This problem is addressed by the concept of a `topic`.

A topic may be introduced and various publishers may take turns publishing to that topic. Thus a subscriber only needs to know the name of a topic and does not need to know what node or nodes publish to that topic. For example, the topic for commanding velocities may be `vel_cmd`, and the robot's low-level controller should subscribe to this named topic to receive velocity commands. Different publishers may be responsible for publishing velocity-command messages on this topic, whether these are nodes under experimentation or trusted nodes that are swapped in to address specific task needs.

Although creating the abstraction of a `topic` helps some, a publisher and a subscriber both need to know how to communicate via a topic. This is accomplished through communications middleware in ROS via the provided executable node `roscore`. The `roscore` node is responsible for coordinating communications, like an operator. Although there can be many ROS nodes distributed across multiple networked computers, there can be only one instance of `roscore` running, and the machine on which `roscore` runs establishes the `master` computer of the system.

A publisher node initiates a topic by informing `roscore` of the topic (and the corresponding message type). This is called `advertising` the topic. To accomplish this, the publisher instantiates an object of the class `ros::Publisher`. This class definition is part of the ROS distribution, and using publisher objects allows the designer to avoid having to write communications code. After instantiating a publisher object, the user code invokes the member function `advertise` and specifies the message type and declares the desired topic name. At this point, the user code can start sending messages to the named topic using the publisher member function `publish`, which takes as an argument the message to be published.

Since a publisher node communicates with `roscore`, `roscore` must be running before any ROS node is launched. To run `roscore`, open a terminal in Linux and enter `roscore`. The response to this command will be a confirmation "started core services." It will also print

the `ROS_MASTER_URI`, which is useful for informing nodes running on non-master computers how to reach `roscore`. The terminal running `roscore` will be dedicated to `roscore`, and it will be unavailable for other tasks. The `roscore` node should continue running as long as the robot is actively controlled (or as long as desired to access the robot's sensors).

After `roscore` has been launched, a publisher node may be launched. The publisher node will advertise its topic and may start sending messages (at any rate convenient to the node, though at a frequency limited by system capacity). Publishing messages at 1 kHz rate is normal for low-level controls and sensor data.

Introducing a sensor to a ROS system requires specialized code (and possibly specialized hardware) that can communicate with the sensor. For example, a LIDAR sensor may require RS488 communications, accept commands in a specific format, and start streaming data in a predefined format. A dedicated microcontroller (or a node within the main computer) must communicate with the LIDAR, receive the data, then publish the data with a ROS message type on a ROS topic. Such specialized nodes convert the specifics of individual sensors into the generic communications format of ROS.

When a publisher begins publishing, it is not necessary that any nodes are listening to the messages. Alternatively, there may be many subscribers to the same topic. The publisher does not need to be aware of whether it has any subscribers nor how many subscribers there may be. This is handled by the ROS middleware. A subscriber may be receiving messages from a publisher, and the publisher node may be halted and possibly replaced with an alternative publisher of the same topic, and the subscriber will resume receiving messages with no need to restart the subscriber.

A ROS subscriber also communicates with `roscore`. To do so, it uses an object of class `ros::Subscriber`. This class has a member function called `subscribe` that requires an argument of the named topic. The programmer must be aware that a topic of interest exists and know the name of the topic. Additionally, the subscriber function requires the name of a `callback` function. This provides the necessary hook to the ROS middleware, such that the callback function will start receiving messages. The callback function suspends until a new message has been published, and the designer may include code to operate on the newly received message.

Subscriber functions can be launched before the corresponding publisher functions. ROS will allow the subscriber to register its desire to receive messages from a named topic, even though that topic does not exist. At some point, if or when a publisher informs `roscore` that it will publish to that named topic, the subscriber's request will be honored, and the subscriber will start receiving the published messages.

A node can be both a subscriber and a publisher. For example, a control node would need to receive sensor signals as a subscriber and send out control commands as a publisher. This only requires instantiating both a subscriber object and a publisher object within the node. It is also useful to pipeline messages for sequential processing. For example, a low-level image processing routine (*e.g.* for edge finding) could subscribe to raw camera data and publish low-level processed images. A higher-level node might subscribe to the edge-processed images, look for specific shapes within those images, and publish its results (*e.g.* identified shapes) for further use by still higher-level processes. A sequence of nodes performing successive levels of processing can be modified incrementally by replacing one node at a time. To replace one such link in a chain, the new node needs only to continue to use the same input and output topic names and message types. Although the implementation of algorithms within the modified node may be dramatically different, the other nodes within the chain will be unaffected.

The flexibility to launch publisher and subscriber nodes in any order eases system design. Additionally, individual nodes may be halted at any time and additional nodes may be hot-swapped into the running system. This can be exploited, for example, to launch some

specialized code when it is needed and then halt the (potentially expensive) computations when no longer needed. Additionally, diagnostic nodes (*e.g.* interpreting and reporting on published messages) may be run and terminated *ad hoc*. This can be useful to examine the output of selected sensors to confirm proper functioning.

It should be appreciated that the flexibility of launching and halting publishers and subscribers at any time within a running system can also be a liability. For time-critical code—particularly control code that depends on sensor values to generate commands to actuators—an interruption of the control code or of the critical sensor publishers could result in physical damage to the robot or its surroundings. It is up to the programmer to make sure that time-critical nodes remain viable. Disruptions of critical nodes should be tested and responded to appropriately (*e.g.* with halting all actuators).

From the system architecture point of view, ROS helps implement a desired software architecture and supports teamwork in building large systems. Starting from a predetermined software architecture, one can construct a skeleton of a large system constructed as a collection of nodes. Initially, each of the nodes might be dummy nodes, capable of sending and receiving messages via predefined topics (the software interfaces). Each module in the architecture could then be upgraded incrementally by swapping out an older (or dummy) node for a newer replacement and no changes would be required elsewhere throughout the system. This supports distributed software development and incremental testing, which are essential for building large, complex systems.

1.2 WRITING ROS NODES

In this section, design of a minimal publisher node and a minimal subscriber node will be detailed. The concept of a ROS `package` is introduced, along with instructions on how to compile and link the code via the associated files `package.xml` and `CMakeLists`. Several ROS tools and commands are introduced, including `rosrun`, `rostopic`, `rosnode`, and `rqt_graph|`. Specific C++ code examples for a publisher and a subscriber are detailed, and results of running the compiled nodes are shown.

The code examples used in this section are contained in the accompanying code repository, within the directory package `minimal_nodes` under `learning_ros/Part_1/minimal_nodes`. This introduction begins with instructions on how the example code was created. In following these instructions it is important to avoid ambiguity from naming conflicts. In this section, the names used will be altered (as necessary) from the provided code examples for the purpose of illustrating to how the example code was created. In subsequent sections, the example code may be used verbatim.

Before creating new ROS code, one must establish a directory (a ROS workspace) where ROS code will reside. One creates this directory somewhere convenient in the system (for example, directly under `home`). A subdirectory called `src` must exist, and this is where source code (packages) will reside. The operating system must be informed of the location of your ROS workspace (typically done automatically through edits to the start-up script `.bashrc`). Setting up a ROS workspace (and automating defining ROS paths) needs to be done only once. The process is described at: http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment. It is important that the necessary environment variables be set in Linux, or the OS will not know where to find your code for compilation and execution. Formerly (ROS Fuerte and older), ROS used its own build system called `rosbuild` that was replaced by the `path{catkin}`indexcatkin build system, which is faster, but can be more complex. A useful simplification is `catkin_simple`, which reduces the detail required of the programmer to specify how to build a project.

For the following, it is assumed that you already have ROS Indigo installed, that you have a ROS workspace defined (called `ros_ws` in the examples to follow), that it has

a src subdirectory, and that the OS has been informed of the path to this workspace (via environment variables). These requirements will satisfied if your setup uses the learning_ros_setup_scripts recommended in the preface. We proceed with creating new code within this workspace.

1.2.1 Creating ROS packages

The first thing to do when starting to design new ROS code is to create a package. A package is a ROS concept of bundling multiple, necessary components together to ease building ROS code and coupling it with other ROS code. Packages should be logical groups of code, *e.g.* separate packages for low-level joint control, for high-level planning, for navigation, for mapping, for vision, etc. Although these packages normally would be developed separately, nodes from separate packages would ultimately be run together simultaneously on a robot (or on a network of computers collectively controlling the robot).

One creates a new package using the catkin_create_pkg command (or the alternative cs_create_pkg, which will be the preferred approach in this presentation). The catkin_create_pkg command is part of the ROS installation. For a given package, package-creation needs to be done only once. One can go back to this package and add more code incrementally (without needing to create the package again). However, the code added to a package should logically belong to that package (*e.g.* avoid putting low-level joint-control code in the same package as mapping code).

New packages should reside under the src directory of your catkin workspace (*e.g.* ros_ws/src). As a specific example, consider creation of a new package called my_minimal_nodes, which will contain source code in C++ and depend on the basic, pre-defined message types contained in std_msgs. To do so, open a terminal and navigate (cd) to the ros_ws directory. A shortcut for this is roscd, which will bring you to ~/ros_ws. From here, move to the subdirectory /src and enter the following command:

```
catkin_create_pkg    my_minimal_nodes    roscpp    std_msgs
```

The effect of this command is to create and populate a new directory: ~/ros_ws/src/my_minimal_nodes.

Every package name in ROS must be unique. By convention, package names should follow common C variable naming conventions: lower case, start with a letter, use underscore separators, *e.g.* grasp_planner. (See http://wiki.ros.org/ROS/Patterns/Conventions.) Every package used within your system must be uniquely named. As noted at http://wiki.ros.org/ROS/Patterns/Conventions, you can check whether a name is already taken via http://www.ros.org/browse/list.php. For the present illustration, the name my_minimal_nodes was chosen so as not to conflict with the package minimal_nodes, which resides in the example code repository (under ~/ros_ws/src/learning_ros/Part_1/minimal_nodes).

Moving to the newly created package directory, ~/ros_ws/src/my_minimal_nodes, we see that it is already populated with package.xml, CMakeLists and the subdirectories src and include. The catkin_create_pkg command just created a new package by the name of my_minimal_nodes, which will reside in a directory of this name.

As we create new code, we will depend on some ROS tools and definitions. Two dependencies were listed during the catkin_create_pkg command: roscpp and std_msgs. The roscpp dependency establishes that we will be using a C++ compiler to create our ROS code, and we will need C++ compatible interfaces (such as the classes ros::Publisher and ros::Subscriber, referred to earlier). The std_msgs dependency says that we will need to

rely on some datatype definitions (standard messages) that have been predefined in ROS. (an example is `std_msgs::Float64`).

The package.xml file: A ROS package is recognized by the build system by virtue of the fact that it has a `package.xml` file. A compatible `package.xml` file has a specific structure that names the package and lists its dependencies. For the new package `my_minimal_nodes`, a `package.xml` file was auto-generated, and its contents are shown in Listing 1.1.

Listing 1.1: Contents of `package.xml` for minimal nodes package

```
<?xml version="1.0"?>
<package>
  <name>my_minimal_nodes</name>
  <version>0.0.0</version>
  <description>The my_minimal_nodes package</description>

  <!-- One maintainer tag required, multiple allowed, one person per tag -->
  <!-- Example:  -->
  <!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
  <maintainer email="wyatt@todo.todo">wyatt</maintainer>


  <!-- One license tag required, multiple allowed, one license per tag -->
  <!-- Commonly used license strings: -->
  <!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
  <license>TODO</license>


  <!-- Url tags are optional, but multiple are allowed, one per tag -->
  <!-- Optional attribute type can be: website, bugtracker, or repository -->
  <!-- Example: -->
  <!-- <url type="website">http://wiki.ros.org/my_miminal_nodes</url> -->


  <!-- Author tags are optional, multiple are allowed, one per tag -->
  <!-- Authors do not have to be maintainers, but could be -->
  <!-- Example: -->
  <!-- <author email="jane.doe@example.com">Jane Doe</author> -->


  <!-- The *_depend tags are used to specify dependencies -->
  <!-- Dependencies can be catkin packages or system dependencies -->
  <!-- Examples: -->
  <!-- Use build_depend for packages you need at compile time: -->
  <!--   <build_depend>message_generation</build_depend> -->
  <!-- Use buildtool_depend for build tool packages: -->
  <!--   <buildtool_depend>catkin</buildtool_depend> -->
  <!-- Use run_depend for packages you need at runtime: -->
  <!--   <run_depend>message_runtime</run_depend> -->
  <!-- Use test_depend for packages you need only for testing: -->
  <!--   <test_depend>gtest</test_depend> -->
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>std_msgs</run_depend>


  <!-- The export tag contains other, unspecified, tags -->
  <export>
    <!-- Other tools can request additional information be placed here -->

  </export>
</package>
```

The `package.xml` file is merely ASCII text using XML formatting, and thus you can open it with any editor (`gedit` will do). In Listing 1.1, most of the lines are merely comments, such as `<!-- Example:  -->`, where each comment is delimited by an opening of `<!--`

and closing of `-->`. The comments instruct you how to edit this file appropriately. It is recommended that you edit the values to enter your name and e-mail address as author of the code, particularly if you intend to share your contribution publicly.

The line `<name>my_minimal_nodes</name>` corresponds to the name of the new package. It is important that this name correspond to the name of your package. You cannot merely create a new directory (with a new name) and copy over the contents of another package. Because of the mismatch between your directory name and the package name in the `package.xml` file, ROS will be confused.

Within the `package.xml` file, the lines:

```
<build_depend>roscpp</build_depend>
<build_depend>std_msgs</build_depend>
<run_depend>roscpp</run_depend>
<run_depend>std_msgs</run_depend>
```

explicitly declare dependency on the package `roscpp` and on the package `std_msgs`, both of which were explicitly listed as dependencies upon creation of this package. Eventually, we will want to bring in large bodies of third-party code (other packages). In order to integrate with these packages (*e.g.* utilize objects and datatypes defined in these packages), we will want to add them to the `package.xml` file. This can be done by editing our package's `package.xml` file and adding `build_depend` and `run_depend` lines naming the new packages to be utilized, emulating the existing lines that declare dependence on `roscpp` and `std_msgs`.

The `src` directory is where we will put our user-written C++ code. We will write illustrative nodes `minimal_publisher.cpp` and `minimal_subscriber.cpp` as examples. It will be necessary to edit the `CMakeLists.txt` file to inform the compiler that we have new nodes to be compiled, which will be described further later.

1.2.2 Writing a minimal ROS publisher

In a terminal window, move to the `src` directory within the `my_minimal_nodes` package that has been created. Open an editor, create a file called `minimal_publisher.cpp` and enter the following code. (Note: if you attempt to copy/paste from an electronic version of this text, you likely will get copying errors, including undesired newline symbols, which will confuse the C++ compiler. Rather, refer to the corresponding example code on the associated github repository at https://github.com/wsnewman/learning_ros, under package `~/ros_ws/src/learning_ros/Part_1/minimal_nodes`.)

Listing 1.2: Minimal Publisher

```
1  #include <ros/ros.h>
2  #include <std_msgs/Float64.h>
3
4  int main(int argc, char **argv) {
5      ros::init(argc, argv, "minimal_publisher"); // name of this node will be "↩
           minimal_publisher"
6      ros::NodeHandle n; // two lines to create a publisher object that can talk to ROS
7      ros::Publisher my_publisher_object = n.advertise<std_msgs::Float64>("topic1", 1);
8      //"topic1" is the name of the topic to which we will publish
9      // the "1" argument says to use a buffer size of 1; could make larger, if expect ↩
           network backups
10
11     std_msgs::Float64 input_float; //create a variable of type "Float64",
12     // as defined in: /opt/ros/indigo/share/std_msgs
13     // any message published on a ROS topic must have a pre-defined format,
14     // so subscribers know how to interpret the serialized data transmission
15
16     input_float.data = 0.0;
```

```
17
18      // do work here in infinite loop (desired for this example), but terminate if ↩
            detect ROS has faulted
19      while (ros::ok())
20      {
21          // this loop has no sleep timer, and thus it will consume excessive CPU time
22          // expect one core to be 100% dedicated (wastefully) to this small task
23          input_float.data = input_float.data + 0.001; //increment by 0.001 each ↩
                iteration
24          my_publisher_object.publish(input_float); // publish the value--of type ↩
                Float64--
25          //to the topic "topic1"
26      }
27  }
```

The above code is dissected here. On line 1,

```
#include <ros/ros.h>
```

is needed to bring in the header files for the core ROS libraries. This should be the first line of any ROS source code written in C++.

Line 2,

```
#include <std_msgs/Float64.h>
```

brings in a header file that describes objects of type: `std_msgs::Float64`, which is a message type we will use in this example code.

As you incorporate use of more ROS message types or ROS libraries, you will need to include their associated header files in your code, just as we have done with `std_msgs`.

Line 4,

```
int main ( int argc , char ** argv )
```

declares a `main` function. For all ROS nodes in C++, there must be one and only one `main()` function per node. Our `minimal_publisher.cpp` file has `main()` declared in the standard "C" fashion with generic `argc`, `argv` arguments. This gives the node the opportunity to use command-line options, which are used by ROS functions (and thus these arguments should always be included in `main()`).

The code lines 5 through 7 all refer to functions or objects defined in the core ROS library.

Line 5:

```
ros::init(argc, argv, "minimal_publisher");
```

is needed in every ROS node. The argument `minimal_publisher` will be the name that the new node will use to register itself with the ROS system upon start-up. (This name can be overridden, or `remapped`, upon launch, but this detail is deferred for now.) The node name is required, and every node in the system must have a unique name. ROS tools take advantage of the node names, *e.g.* to monitor which nodes are active and which nodes are publishing or subscribing to which topics.

Line 6 instantiates a ROS `NodeHandle` object with the declaration:

```
    ros::NodeHandle n;
```

A `NodeHandle` is required for establishing network communications among nodes. The

`NodeHandle` name `n` is arbitrary. It will be used infrequently (typically, for initializing communications objects). This line can simply be included in every node's source code (and no harm is done in the rare instances in which it is not needed).

Line 7:

```
ros::Publisher my_publisher_object = n.advertise<std_msgs::Float64>("topic1", 1);
```

instantiates an object to be called `my_publisher_object` (the name is the programmer's choice). In instantiating this object, the ROS system is informed that the current node (here called `minimal_publisher`) intends to publish messages of type `std_msgs::Float64` on a topic named `topic1`. In practice, one should choose topic names that are helpful and descriptive of the type of information carried via that topic.

On line 11:

```
std_msgs::Float64 input_float;
```

the program creates an object of type `std_msgs:Float64` and calls it `input_float`. One must consult the message definition in `std_msgs` to understand how to use this object. The object is defined as having a member called `data`. Details of this message type can be found by looking in the corresponding directory with: `roscdstd_msgs`. The subdirectory `msg` contains various files defining the structures of numerous standard messages, including `Float64.msg`. Alternatively, one can examine the details of any message type with the command `rosmsg show ...`, *e.g.* from a terminal, entering the command:

```
rosmsg show std_msgs/Float64
```

will display the fields of this message, which results in the response:

```
float64 data
```

In this case, there is only a single field, named `data`, which holds a value of type `float64` (a ROS primitive).

On line 16,

```
input_float.data = 0.0;
```

the program initializes the `data` member of `input_float` to the value 0.0. An infinite loop is then entered, that will self terminate upon detecting that the ROS system has terminated, which is accomplished using the function `ros::ok()` in line 19:

```
while (ros::ok())
```

This approach can be convenient for shutting down a collection of nodes by merely halting the ROS system (*i.e.*, by killing `roscore`).

Inside the `while` loop, the value of `input_float.data` is incremented by 0.001 per iteration. This value is then published (line 24) using:

```
my_publisher_object.publish(input_float);
```

It was previously established (upon instantiation of the object `my_publisher_object` from the class `ros::Publisher`) that the object `my_publisher_object` would publish

messages of type `std_msg::Float64` to the topic called `topic1`. The publisher object, `my_publisher_object` has a member function `publish` to invoke publications. The publisher expects an argument of compatible type. Since the object `input_float` is of type `std_msgs::Float64`, and since the publisher object was instantiated to expect arguments of this type, the `publish` call is consistent.

The example ROS node has only 14 active lines of code. Much of this code is ROS-specific and may seem cryptic. However, most of the lines are common boilerplate, and becoming familiar with these common lines will make other ROS code easier to read.

1.2.3 Compiling ROS nodes

ROS nodes are compiled by running `catkin_make`. This command must be executed from a specific directory. In a terminal, navigate to your ROS workspace (`~/ros_ws`). Then enter `catkin_make`.

This will compile all packages in your workspace. Compiling large collections of code can be time consuming, but compilation will be faster on subsequent edit, compile and test iterations. Although compilation is sometimes slow, the compiled code can be very efficient. Particularly for CPU-intensive operations (*e.g.* for point-cloud processing, image processing or intensive planning computations), compiled C++ code typically runs faster than Python code.

After building a catkin package, the executables will reside in a folder in `ros_ws/devel/lib` named according to the source package.

Before we can compile, however, we have to inform `catkin_make` of the existence of our new source code, `minimal_publisher.cpp`. To do so, edit the file `CMakeLists.txt`, which was created for us in the package `my_minimal_nodes` when we ran `catkin_create_pkg`. This file is quite long (187 lines for our example), but it consists almost entirely of comments.

The comments describe how to modify `CmakeLists.txt` for numerous variations. For the present, we only need to make sure we have our package dependencies declared, inform the compiler to compile our new source code, and link our compiled code with any necessary libraries.

`catkin_package_create` already fills in the fields:

Listing 1.3: Snippet from `CMakeLists.txt`

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  std_msgs
)

include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

However, we need to make two modifications, as follows:

Listing 1.4: Adding the new node and linking it with libraries

```
## Declare a cpp executable
add_executable(my_minimal_publisher src/minimal_publisher.cpp)

## Specify libraries to link a library or executable target against
target_link_libraries(my_minimal_publisher ${catkin_LIBRARIES} )
```

These modifications inform the compiler of our new source code, as well as which libraries with which to link.

In the above, the first argument to `add_executable` is a chosen name for the executable to be created, here chosen to be `my_minimal_publisher`, which happens to be the same root name as the source code.

The second argument is where to find the source code relative to the package directory. The source code is in the `src` subdirectory and it is called `minimal_publisher.cpp`. (It is typical for the source code to reside in the `src` sub-directory of a package.)

There are a few idiosyncrasies regarding the node name. In general, one cannot run two nodes with the same name. The ROS system will complain, kill the currently running node, and start the new (identically named) node. ROS does allow for different packages to re-use node names, although only one of these at a time may be run. Although (executable) node names are allowed to be duplicated across packages, the `catkin_make` build system gets confused by such duplication (though the build can be forced by compiling packages one at a time). For simplicity, it is best to avoid replicating node names.

Having edited the `CMakeLists` file, we can compile our new code. To do so, from a terminal, navigate to the `ros_ws` directory and enter:

```
catkin_make
```

This will invoke the C++ compiler to build all packages, including our new `my_minimal_nodes` package. If the compiler output complains, find and fix your bugs.

Assuming compilation success, if you look in the directory `ros_ws/devel/lib/my_minimal_nodes`, you will see a new, executable file there named `my_minimal_publisher`. This is the name that was chosen for the output file (executable node) with the addition of `add_executable(my_minimal_publisher src/minimal_publisher.cpp)` in `CMakeLists`.

1.2.4 Running ROS nodes

As noted in Section 1.1, there must be one and only one instance of `roscore` running before any nodes can be started. In a terminal, enter:

```
roscore
```

This should respond with a page of text concluding with "started core services." You can then shrink this window and leave it alone. ROS nodes can be started and stopped at random without needing to start a new `roscore`. (If you kill `roscore` or the window running `roscore`, however, all nodes will stop running.)

Next, start the new publisher node by entering (from a new terminal):

```
rosrun    my_minimal_nodes    my_minimal_publisher
```

The arguments to the command `rosrun` are the package name (`my_minimal_nodes`) and the executable name (`my_minimal_publisher`).

The `rosrun` command can seem confusing at times due to re-use of names. For example, if we wanted to make a LIDAR publisher node, we might have a package called `lidar_publisher`, a source file called `lidar_publisher.cpp`, an executable called `lidar_publisher`, and a node name (declared within the source code) of `lidar_publisher`. To run this node, we would type:

```
rosrun    lidar_publisher    lidar_publisher
```

This may seem redundant, but it still follows the format:

```
rosrun    package_name    executable_name
```

Once the command has been entered, the ROS system will recognize a new node by the name of `lidar_publisher`. This name re-use may seem to lead to confusion, but in many instances, there is no need to invent new names for the package, the source code, the executable name and the node name. In fact, this can help simplify recognizing named entities–as long as the context is clear (package, source code, executable, node name).

1.2.5 Examining running minimal publisher node

After entering: `rosrun my_minimal_nodes my_minimal_publisher`, the result may seem disappointing. The window that launched this node seems to hang and provides no feedback to the user. Still, `minimal_publisher` is running. To see this, we can invoke some ROS tools.

Open a new terminal and enter: `rostopic`. You will get the following response:

```
rostopic is a command-line tool for printing information about ROS Topics.

Commands:
rostopic bw display bandwidth used by topic
rostopic echo print messages to screen
rostopic find find topics by type
rostopic hz display publishing rate of topic
rostopic info print information about active topic
rostopic list list active topics
rostopic pub publish data to topic
rostopic type print topic type

Type rostopic <command> -h for more detailed usage, e.g. 'rostopic echo -h'
```

This shows that the command `rostopic` has eight options. If we type:

```
rostopic list
```

the result is:

```
/rosout
/rosout_agg
/topic1
```

We see that there are three active topics—two that ROS created on its own and the topic created by our publisher, `topic1`.

Entering:

```
 rostopic   hz   topic1
```

results in the following output:

```
average rate: 38299.882
min: 0.000s max: 0.021s std dev: 0.00015s window: 50000
average rate: 38104.090
min: 0.000s max: 0.024s std dev: 0.00016s window: 50000
```

This output shows that our minimal publisher (on this particular computer) is publishing its data at roughly 38 kHz (with some jitter). Viewing the system monitor would show that one CPU core is fully saturated just running the minimal publisher. This is because the `while` loop within our ROS node has no pauses. It is publishing as fast as it can.

Entering:

```
rostopic bw topic1
```

yields the following output:

```
average: 833.24KB/s
mean: 0.01KB min: 0.01KB max: 0.01KB window: 100
average: 1.21MB/s
mean: 0.00MB min: 0.00MB max: 0.00MB window: 100
average: 746.32KB/s
mean: 0.01KB min: 0.01KB max: 0.01KB window: 100
```

This display shows how much of our available communications bandwidth is consumed by our minimal publisher (nominally 1 MB/s). This rostopic option can be useful for identifying nodes that are over-consuming communications resources.

Entering:

```
rostopic info topic1
```

yields:

```
Type: std_msgs/Float64

Publishers:
 * /minimal_publisher (http://Wall-E:56763/)

Subscribers: None
```

This tells us that `topic1` involves messages of type `std_msgs/Float64`. At present, there is a single publisher to this topic (which is the norm), and that publisher has a node name of `minimal_publisher`. As noted above, this is the name we assigned to the node within the source code on line 5:

```
ros::init(argc,argv,"minimal_publisher");
```

Entering:

```
rostopic echo topic1
```

causes `rostopic` to try to print out everything published on `topic1`. A sample of the output is:

```
data: 860619.606909
---
data: 860619.608909
---
data: 860619.609909
---
data: 860619.612909
---
```

In this case, the display has no hope of keeping up with the publishing rate, and most of the messages are dropped between lines of display. If the echo could keep up, we would expect to see values that increase by increments of 0.001, which is the increment used in the `while` loop of our source code.

The `rostopic` command tells us much about the status of our running system, even though there are no nodes receiving the messages sent by our minimal publisher. Additional handy ROS commands are summarized in the "ROS cheat sheet" (see http://www.ros.org/news/2015/05/ros-cheatsheet-updated-for-indigo-igloo.html).

For example, entering:

```
rosnode list
```

results in the following output:

```
/minimal_publisher
/rosout
```

We see that there are two nodes running: `rosout` (a generic process used for nodes to display text to a terminal, launched by default by `roscore`) and `minimal_publisher` (our node).

Although `minimal_publisher` does not take advantage of the capability of displaying output to its terminal, the link is nonetheless available through the topic `rosout`, which would get processed by the display node `rosout`. Using `rosout` can be helpful, since one's code does not get slowed down by output (*e.g.* `cout`) operations. Rather, messages get sent rapidly by publishing the output to the `rosout` topic, and a separate node (`rosout`) is responsible for user display. This can be important, *e.g.* in time-critical code where some monitoring is desired, but not at the expense of slowing the time-critical node.

1.2.6 Scheduling node timing

We have seen that our example publisher is abusive of both CPU capacity and communications bandwidth. In fact, it would be unusual for a node within a robotic system to require updates at 30 kHz. A more reasonable update rate for even time-critical, low-level nodes is 1 kHz. In the present example, we will slow our publisher to 1 Hz using a ROS timer.

A modified version of source code for `minimal_publisher.cpp`, called `sleepy_minimal_publisher.cpp`, is shown below:

Listing 1.5: Minimal Publisher with Timing: `sleepy_minimal_publisher.cpp`

```
1  #include <ros/ros.h>
2  #include <std_msgs/Float64.h>
3
4  int main(int argc, char **argv) {
5      ros::init(argc, argv, "minimal_publisher2"); // name of this node will be "↩
           minimal_publisher2"
6      ros::NodeHandle n; // two lines to create a publisher object that can talk to ROS
7      ros::Publisher my_publisher_object = n.advertise<std_msgs::Float64>("topic1", 1);
8      //"topic1" is the name of the topic to which we will publish
9      // the "1" argument says to use a buffer size of 1; could make larger, if expect ↩
           network backups
10
11     std_msgs::Float64 input_float; //create a variable of type "Float64",
12     // as defined in: /opt/ros/indigo/share/std_msgs
13     // any message published on a ROS topic must have a pre-defined format,
14     // so subscribers know how to interpret the serialized data transmission
15
16    ros::Rate naptime(1.0); //create a ros object from the ros Rate class;
17    //set the sleep timer for 1Hz repetition rate (arg is in units of Hz)
```

```
18
19      input_float.data = 0.0;
20
21      // do work here in infinite loop (desired for this example), but terminate if ↩
            detect ROS has faulted
22      while (ros::ok())
23      {
24          // this loop has no sleep timer, and thus it will consume excessive CPU time
25          // expect one core to be 100% dedicated (wastefully) to this small task
26          input_float.data = input_float.data + 0.001; //increment by 0.001 each ↩
                iteration
27          my_publisher_object.publish(input_float); // publish the value--of type ↩
                Float64--
28          //to the topic "topic1"
29      //the next line will cause the loop to sleep for the balance of the desired period
30          // to achieve the specified loop frequency
31      naptime.sleep();
32      }
33  }
```

There are only two new lines in the above program: line 16

```
ros::Rate naptime(1); //set the sleep timer for 1Hz repetition rate
```

and line 31:

```
 naptime.sleep();
```

The ROS class `Rate` is invoked to create a `Rate` object that was named "naptime". In doing so, `naptime` is initialized with the value "1", which is a specification of the desired frequency (1 Hz). After creating this object, it is used within the `while` loop, invoking the member function `sleep()`. This causes the node to suspend (thus ceasing to consume CPU time) until the balance of the desired period (1 second) has expired.

After re-compiling the modified code (with `catkin_make`), we can run it (with `rosrun`) and examine its behavior by entering (from a new terminal):

```
rostopic hz topic1
```

which produces the display below:

```
average rate: 1.000
min: 1.000s max: 1.000s std dev: 0.00000s window: 2
average rate: 1.000
min: 1.000s max: 1.000s std dev: 0.00006s window: 3
average rate: 1.000
min: 1.000s max: 1.000s std dev: 0.00005s window: 4
average rate: 1.000
```

This output indicates that `topic1` is being updated at 1 Hz with excellent precision and very low jitter. Further, an inspection of the system monitor shows negligible CPU time consumed by our modified publisher node.

If we enter

```
rostopic echo topic1
```

from a terminal, example output looks like the following:

```
data: 0.153
---
```

```
data: 0.154
---
data: 0.155
---
```

Each message sent by the publisher is displayed by `rostopic echo`, as evidenced by the increments of 0.001 between messages. This display is updated once per second, since that is the rate at which new data is now published.

1.2.7 Writing a minimal ROS subscriber

The complement to the publisher is a subscriber (a listener node). We will create this node in the same package, `my_minimal_nodes`. The source code will go in the subdirectory `src`.

Open an editor and create the file `minimal_subscriber.cpp` in the directory `~/ros_ws/src/my_minimal_nodes/src`. Enter the following code (which may be found in `~/ros_ws/src/learning_ros/Part_1/minimal_nodes/src/minimal_subscriber.cpp`):

Listing 1.6: Minimal Subscriber

```
1  #include<ros/ros.h>
2  #include<std_msgs/Float64.h>
3  void myCallback(const std_msgs::Float64& message_holder)
4  {
5    // the real work is done in this callback function
6    // it wakes up every time a new message is published on "topic1"
7    // Since this function is prompted by a message event,
8    //it does not consume CPU time polling for new data
9    // the ROS_INFO() function is like a printf() function, except
10   // it publishes its output to the default rosout topic, which prevents
11   // slowing down this function for display calls, and it makes the
12   // data available for viewing and logging purposes
13   ROS_INFO("received value is: %f",message_holder.data);
14   //really could do something interesting here with the received data...but all we do ↩
         is print it
15 }
16
17 int main(int argc, char **argv)
18 {
19   ros::init(argc,argv,"minimal_subscriber"); //name this node
20   // when this compiled code is run, ROS will recognize it as a node called "↩
         minimal_subscriber"
21   ros::NodeHandle n; // need this to establish communications with our new node
22   //create a Subscriber object and have it subscribe to the topic "topic1"
23   // the function "myCallback" will wake up whenever a new message is published to ↩
         topic1
24   // the real work is done inside the callback function
25
26   ros::Subscriber my_subscriber_object= n.subscribe("topic1",1,myCallback);
27
28   ros::spin(); //this is essentially a "while(1)" statement, except it
29   // forces refreshing wakeups upon new data arrival
30   // main program essentially hangs here, but it must stay alive to keep the callback ↩
         function alive
31   return 0; // should never get here, unless roscore dies
32 }
33 }
```

Most of the code within the minimal subscriber is identical to that of the minimal publisher (though the node name in line 19 has been changed to "`minimal_subscriber`"). There are four important new lines to examine.

Most notably, the subscriber is more complex than the publisher, since it requires a `callback`, which is declared in line 3,

```
void myCallback(const std_msgs::Float64& message_holder)
```

This function has an argument of a reference pointer (indicated by the & sign) to an object of type `std_msgs::Float64`. This is the message type associated with `topic1`, as published by our minimal publisher.

The importance of the callback function is that it is awakened when new data is available on its associated topic (which is set to `topic1` in this example). When new data is published to the associated topic, the callback function runs, and the published data appears in the argument `message_holder`. (This message holder must be of a type compatible with the message type published on the topic of interest, in this case `std_msgs::Float64`).

Within the callback function, the only action taken is to display the received data, implemented on line 13.

```
 ROS_INFO("received value is: %f",message_holder.data);
```

Display is performed using `ROS_INFO()` instead of `cout` or `printf`. `ROS_INFO()` uses message publishing, which avoids slowing time-critical code for display driving. Also, using `ROS_INFO()` makes the data available for logging or monitoring. However, as viewed from the terminal from which this node is run, the output is displayed equivalently to using `cout` or `printf`. The argument to `ROS_INFO()` is the same as `printf` in C.

An alternative to using `ROS_INFO()` is `ROS_INFO_STREAM()`. Line 13 could be replaced with:

```
ROS_INFO_STREAM("received value is: "<<message_holder.data<<std::endl);
```

which produces the same output, but uses syntax of `cout`.

Once the callback function executes, it goes back to sleep, ready to be re-awakened by arrival of a new message on `topic1`.

In the main program, a key new concept is on line 26:

```
ros::Subscriber my_subscriber_object= n.subscribe("topic1",1,myCallback);
```

The use of `ros::Subscriber` is similar to the use of `ros::Publisher` earlier. An object of type `Subscriber` is substantiated; `Subscriber` is a class that exists within the ROS distribution. There are three arguments used in instantiating the subscriber object. The first argument is the topic name; `topic1` is chosen as the topic to which our minimal publisher publishes. (For this example, we want our subscriber node to listen to the output of our example publisher node.)

The second argument is the queue size. If the callback function has trouble keeping up with published data, the data may be queued. In the present case, the queue size is set to one. If the callback function cannot keep up with publications, messages will be lost by being overwritten by newer messages. (Recall that in the first example, `rostopic echo topic1` could not keep up with the 30 kHz rate of the original minimal publisher. Values displayed skipped many intermediate messages.) For control purposes, typically only the most recent sensor value published is of interest. If a sensor publishes faster than the callback function can respond, there is no harm done in dropping messages, as only the most recent message would be needed. In this (and many cases) a queue size of one message is all that is needed.

The third argument for instantiating the `Subscriber` object is the name of the callback function that is to receive data from `topic1`. This argument has been set to `myCallback`, which is the name of our callback function, described earlier. Through this line of code, we associate our callback function with messages published on `topic1`.

Finally, line 28:

```
ros::spin();
```

introduces a key ROS concept that is non-obvious but essential. The callback function should awaken whenever a new message appears on `topic1`. However, the main program must yield some time for the callback function to respond. This is accomplished with a `spin()` command. In the present case, a `spin` causes the main program to suspend, but keeps the callback function alive. If the main program were to run to conclusion, the callback function would no longer be poised to react to new messages. The `spin()` command keeps `main()` alive without consuming significant CPU time. As a result, the minimal subscriber is quite efficient.

1.2.8 Compiling and running minimal subscriber

For our new node to get compiled, we must include reference to it in `CMakeLists`. This requires adding two lines, very similar to what we did to enable compiling the minimal publisher. The first new line is simply:

```
add_executable(my_minimal_subscriber src/minimal_subscriber.cpp)
```

The arguments are the desired executable name (chosen to be `my_minimal_subscriber`), and the relative path to the source code (`src/minimal_subscriber.cpp`).

The second line added is:

```
target_link_libraries(my_minimal_subscriber  ${catkin_LIBRARIES} )
```

which informs the compiler to link our new executable with the declared libraries.

After updating `CMakeLists`, the code is newly compiled with the command `catkin_make` (which must be run from the `ros_ws` directory).

The code example should compile without error, after which a new executable, `my_minimal_subscriber`, will appear in the directory: `~/ros_ws/devel/lib/my_minimal_nodes`. Note: it is not necessary to recall this lengthy path to the executable file. The executable will be found when running `rosrun` with the package name and node name as arguments.

After recompiling, we now have two nodes to run. In one terminal, enter:

```
rosrun my_minimal_nodes sleepy_minimal_publisher
```

and in a second terminal enter:

```
rosrun my_minimal_nodes my_minimal_subscriber
```

It does not matter which is run first. An example of the display in the terminal of the `my_minimal_subscriber` node is shown below:

```
[ INFO] [1435555572.972403158]: received value is: 0.909000
[ INFO] [1435555573.972261535]: received value is: 0.910000
[ INFO] [1435555574.972258968]: received value is: 0.911000
```

This display was updated once per second, since the publisher published to `topic1` at

1 Hz. The messages received differ by increments of 0.001, as programmed in the publisher code.

In another terminal, entering:

```
rosnode list
```

results in the output below.

```
/minimal_publisher2
/minimal_subscriber
/rosout
```

This shows that we now have three nodes running: the default `rosout`, our minimal publisher (which we named node `minimal_publisher2` for the timed version) and our minimal subscriber, `minimal_subscriber`.

In an available terminal, entering:

```
rqt_graph
```

produces a graphical display of the running system, which is shown in Fig 1.1.

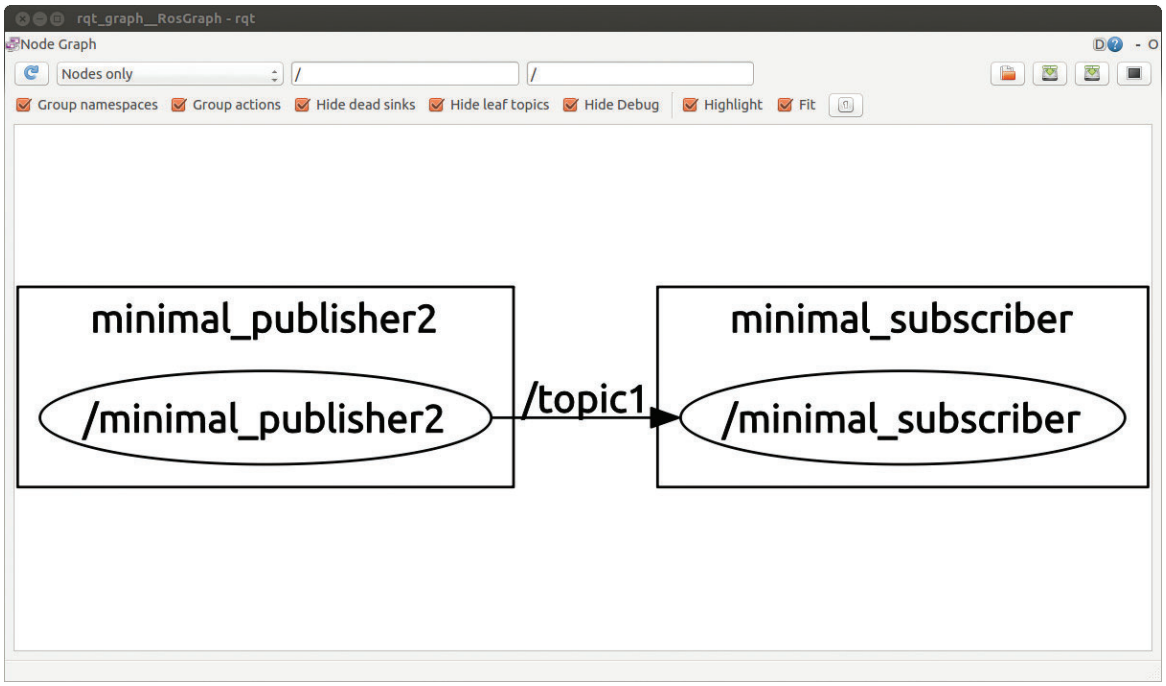


Figure 1.1: Node topology as illustrated by `rqt_graph`

The graphical display shows our two nodes. Our minimal publisher node is shown publishing to `topic1` and our minimal subscriber is shown subscribing to this topic.

1.2.9 Minimal subscriber and publisher node summary

We have seen the basics for how to create our own publisher and subscriber nodes. A single node can subscribe to multiple topics by replicating the corresponding lines of code to create additional subscriber objects and callback functions. Similarly, a node can publish

to multiple topics by instantiating multiple publisher objects. A node can also be both a subscriber and a publisher.

In a ROS-controlled robot system, custom device-driver nodes must be designed that can read and publish sensor information and one or more nodes that subscribe to actuator (or controller setpoint) commands and impose these on actuators. Fortunately, there is already a large body of existing ROS drivers for common sensors, including LIDARs, cameras, the Kinect sensor, inertial measurement units, encoders, etc. These may be imported as packages and used as-is in your system (though perhaps requiring some tweaking to reference specifics of your system). There are also packages for driving some servos (*e.g.* hobby-servo style RCs and Dynamixel motors). There are also some ROS-Industrial interfaces to industrial robots, which only require publishing and subscribing to and from robot topics. In some cases, the user may need to design device driver nodes to interface to custom actuators. Further, hard real-time, high-speed servo loops may require a non-ROS dedicated controller (although this might be as simple as an Arduino microcontroller). The user then assumes the responsibility for designing the hard-real-time controller and writing a ROS-compatible subscriber interface to run on the control computer.

1.3 MORE ROS TOOLS: `catkin_simple`, `roslaunch`, `rqt_console`, AND `rosbag`

Having introduced minimal ROS talkers (publishers) and listeners (subscribers), one can already begin to appreciate the value of some additional ROS tools. The tools and facilities introduced here are `catkin_simple`, `roslaunch`, `rqt_console` and `rosbag`.

1.3.1 Simplifying `CMakeLists.txt` with `catkin_simple`

As seen in Section 1.2.3, the `CMakeLists.txt` file generated with a new package creation is quite long. While the required edits to this file were relatively simple, introducing additional features can require tedious, non-obvious changes. A package called `catkin_simple` helps to simplify the `CMakeLists.txt` file. This package can be found at https://github.com/catkin/catkin_simple.git. A copy has been cloned into our external-packages repository at https://github.com/wsnewman/learning_ros_external_packages.git, which should already be cloned into your `~/ros_ws/src` directory.

Additionally, the external-packages repository has a Python script that assists with creating new packages that use `catkin_simple`. To run this script, it is convenient to define an alias to point to this script as a command. Within a terminal, enter:

```
alias cs_create_pkg='~/ros_ws/src/learning_ros_external_packages/cs_create_pkg.py'
```

Subsequently, from this same terminal, you can create a package that uses `catkin_simple` with the command `cs_create_pkg`. For example, navigate to `~/ros_ws/src` and create a new package called `my_minimal_nodes2` by entering:

```
cs_create_pkg my_minimal_nodes2 roscpp std_msgs
```

Note that this command will be recognized only in the terminal from which the alias `cs_create_pkg` was defined. More conveniently, the alias definition should be including in your `.bashrc` file. To do so, navigate to your home directory and edit the (hidden) file `.bashrc`. Append the line

```
alias cs_create_pkg='~/ros_ws/src/learning_ros_external_packages/cs_create_pkg.py'
```

to this file and save it. Subsequently, this alias will be recognized by all new terminals that

are opened. If you have used the recommended setup scripts, this `.bashrc` file edit will already be performed for you.

After having invoked our `cs_create_pkg` command, the new package `my_minimal_nodes2` contains the expected structure, including subdirectories `src` and `include`, the `package.xml` file, a `CMakeLists.txt` file and a new file, `README.md`. The README file should be edited to describe the purpose of the new package and how to run examples within the package. The README file uses `markdown` formatting (see https://guides.github.com/features/mastering-markdown/ for a description of `markdown`). Such formatting allows your README file to be displayed with attractive formatting when viewed from your repository via a browser.

The `package.xml` file is similar to that created by `catkin_create_pkg`, except it includes the additional dependency:

```
<buildtool_depend>catkin_simple</buildtool_depend>
```

The `CMakeLists.txt` file is considerably simplified. A copy of the default generated file is:

Listing 1.7: `CMakeLists.txt` with `catkin_simple`

```
cmake_minimum_required(VERSION 2.8.3)
project(my_minimal_nodes2)

find_package(catkin_simple REQUIRED)

#uncomment next line to use OpenCV library
#find_package(OpenCV REQUIRED)

#uncomment the next line to use the point-cloud library
#find_package(PCL 1.7 REQUIRED)

#uncomment the following 4 lines to use the Eigen library
#find_package(cmake_modules REQUIRED)
#find_package(Eigen3 REQUIRED)
#include_directories(${EIGEN3_INCLUDE_DIR})
#add_definitions(${EIGEN_DEFINITIONS})

catkin_simple()

# example boost usage
# find_package(Boost REQUIRED COMPONENTS system thread)

# C++0x support - not quite the same as final C++11!
# use carefully;  can interfere with point-cloud library
# SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++0x")

# Libraries: uncomment the following and edit arguments to create a new library
# cs_add_library(my_lib src/my_lib.cpp)

# Executables: uncomment the following and edit arguments to compile new nodes
# may add more of these lines for more nodes from the same package
# cs_add_executable(example src/example.cpp)

#the following is required, if desire to link a node in this package with a library ↩
    created in this same package
# edit the arguments to reference the named node and named library within this package
# target_link_libraries(example my_lib)

cs_install()
cs_export()
```

The line `catkin_simple()` invokes actions to automatically perform much of the tedious editing of `CMakeLists.txt`. The commented lines are reminders of how to exploit CMake variations, including linking with popular libraries, such as Eigen, PCL and OpenCV. To modify `CMakeLists.txt` to compile our desired code, uncomment and edit the line:

```
cs_add_executable(example src/example.cpp)
```

By copying `minimal_publisher.cpp` to the new package, `my_minimal_nodes2`, we can specify that this node should be compiled by modifying the above line in `CMakeLists.txt` to:

```
cs_add_executable(minimal_publisher3 src/minimal_publisher.cpp)
```

This indicates that we wish to compile `minimal_publisher.cpp` and call the executable `minimal_publisher3` (a name that will not conflict with other instances of minimal publisher nodes). We can add commands to compile additional nodes by inserting more `cs_add_executable` lines of the same form. It is not necessary to specify linking with libraries. This may not seem to be much of a simplification at this point, but `catkin_simple` will become much more valuable when we start to link with more libraries, create libraries, and create custom messages.

1.3.2 Automating starting multiple nodes

In our minimal example, we ran two nodes: one publisher and one subscriber. To do so, we opened two separate terminals and typed in two `rosrun` commands. Since a complex system may have hundreds of nodes running, we need a more convenient way to bring up a system. This can be done using `launch` files and the command `roslaunch`. (See http://ros.org/wiki/roslaunch for more details and additional capabilities, such as setting parameters.)

A launch file has the suffix `.launch`. It is conventionally named the same as the package name (though this is not required). It is also conventionally located in a subdirectory of the package by the name of `launch` (also not required). A launch file can also invoke other launch files to bring up multiple nodes from multiple packages. However, we will start with a minimal launch file. In our package `my_minimal_nodes`, we may create a subdirectory `launch` and within this directory create a `my_minimal_nodes.launch` file containing the following lines:

```
<launch>
<node name="publisher" pkg="my_minimal_nodes" type="sleepy_minimal_publisher"/>
<node name="subscriber" pkg="my_minimal_nodes" type="my_minimal_subscriber"/>
</launch>
```

In the above, using XML syntax, we use the keyword "node" to tell ROS that we want to launch a ROS node (an executable program compiled by `catkin_make`). We must specify three key/value pairs to launch a node: the package name of the node (value of "pkg"), the binary executable name of the node (value of "type"), and the name by which the node will be recognized by ROS when launched (value of "name"). In fact, we already specified node names within the source code (*e.g.* `ros::init(argc,argv, "minimal_publisher2")` within `sleepy_minimal_publisher.cpp`). The launch file gives you the opportunity to rename the node when it is launched. For example, by setting: `name="publisher"` in the launch file, we would still start running an instance of the executable called `sleepy_minimal_publisher` within the package `my_minimal_nodes`, but it will be known to `roscore` by the name `publisher`. Similarly, by assigning `name="subscriber"` to the executable `my_minimal_subscriber`, this node will be known to `roscore` as `subscriber`.

We can execute the launch file by typing:

```
roslaunch my_minimal_nodes my_minimal_nodes.launch
```

Recall that for using `rosrun` to start nodes, it was necessary that `roscore` be running first. When using `roslaunch`, it is not necessary to start `roscore` running. If `roscore` is already running, the `roslaunch` will launch the specified nodes. If `roscore` is not already running, `roslaunch` will detect this and will start `roscore` before launching the specified nodes.

The terminal in which we executed `roslaunch` displays the following:

```
SUMMARY
========

PARAMETERS
 * /rosdistro: indigo
 * /rosversion: 1.11.13

NODES
  /
    publisher (my_minimal_nodes/sleepy_minimal_publisher)
    subscriber (my_minimal_nodes/my_minimal_subscriber)

ROS_MASTER_URI=http://localhost:11311

core service [/rosout] found
process[publisher-1]: started with pid [18793]
process[subscriber-2]: started with pid [18804]
```

From another terminal, entering:

```
rosnode list
```

produces the output:

```
/publisher
/rosout
/subscriber
```

which shows that nodes known as "publisher" and "subscriber" are running (using the new names assigned to these nodes by the launch file).

Most often, we will not want to change the name of the node from its original specification, but ROS launch files nonetheless require that this option be used. To decline changing the node name, the default name (embedded in the source code) may be used as the desired node name.

After launching our nodes, `rostopic list` shows that `topic1` is alive, and the command `rostopic info topic1` shows that the node `publisher` is publishing to this topic and the node `subscriber` has subscribed to this topic. Clearly, this will be useful when we have many nodes to launch.

One side effect, though, is that we no longer see the output from our subscriber, which formerly appeared in the terminal from which the subscriber was launched. However, since we used `ROS_INFO()` instead of `printf` or `cout`, we can still observe this output using the `rqt_console` tool.

1.3.3 Viewing output in a ROS console

A convenient tool to monitor ROS messages is `rqt_console`, which can be invoked from a terminal by entering: `rqt_console`. With our two nodes running, an example of using this tool is shown in Fig 1.2.

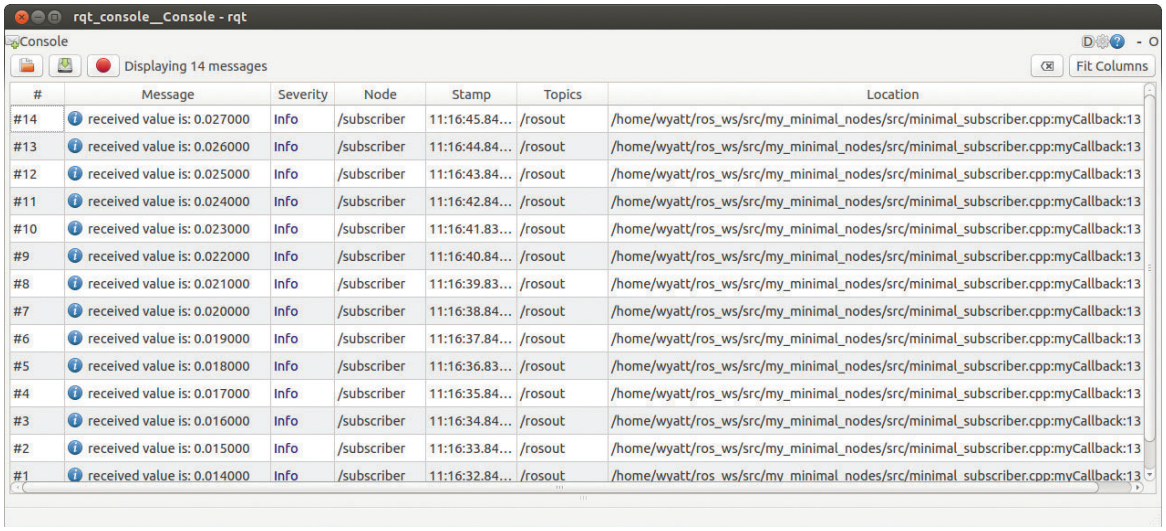


Figure 1.2: Output of `rqt_console` with minimal nodes launched

In this instance, `rqt_console` shows values output by the minimal subscriber from the time `rqt_console` was started and until `rqt_console` was paused (using the `rqt_console` "pause" button). The lines displayed show that the messages are merely informational and not warnings or errors. The console also shows that the node responsible for posting the information is our minimal subscriber (known by the node name "subscriber"). `rqt_console` also notes the time stamp at which the message was sent.

Multiple nodes using `ROS_INFO()` may be run simultaneously, and their messages may be viewed with `rqt_console`, which will also note new events, such as starting and stopping new nodes. Another advantage of using `ROS_INFO()` instead of `printf()` or `cout` is that the messages can be logged and run in playback. A facility for doing this is `rosbag`.

1.3.4 Recording and playing back data with `rosbag`

The `rosbag` command is extremely useful for debugging complex systems. One can specify a list of topics to record while the system is running, and `rosbag` will subscribe to these topics and record the messages published, along with time stamps, in a "bag" file. `rosbag` can also be used to play back bag files, thus recreating the circumstances of the recorded system. (This playback occurs at the same clock rate at which the original data was published, thus emulating the real-time system.)

When running `rosbag`, the resulting log (bag) files will be saved in the same directory from which `rosbag` was launched. For our example, move to the `my_minimal_nodes` directory and create a new subdirectory `bagfiles`. With our nodes still running (which is optional; nodes can be started later), navigate to the `bagfile` directory (wherever you chose to store your bags) and enter:

```
rosbag record topic1
```

With this command, we have asked to record all messages published on `topic1`. Running `rqt_console` will display data from `topic1`, as reported by our subscriber node using `ROS_INFO()`. In the screenshot of `rqt_console` shown in Fig 1.3, the `rosbag` recording startup is noted at line number 34; at this instant, the value output from our subscriber node is 0.236 (i.e., 236 seconds after the nodes were launched).

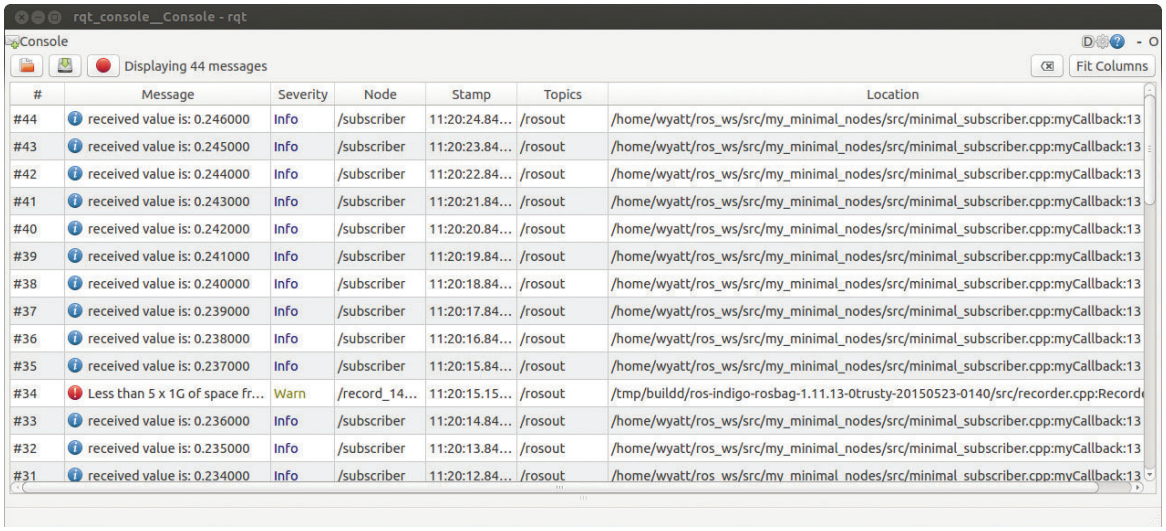


Figure 1.3: Output of `rqt_console` with minimal nodes running and `rosbag` running

---

The `rosbag` node was subsequently halted with a control-C in the terminal from which it was launched. Looking in the `bagfiles` directory (from which we launched `rosbag`), we see there is a new file, named according to the date and time of the recording, and with the suffix `.bag`.

We can play back the recording using `rosbag` as well. To do so, first kill the running nodes, then type:

```
rosbag play fname.bag
```

where "fname" is the file name of the recording. The `rosbag` terminal shows a playback time incrementing, but there is otherwise no noticeable effect. The screen output is:

```
rosbag play 2016-01-07-11-20-15.bag
[ INFO] [1452184943.589921994]: Opening 2016-01-07-11-20-15.bag

Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to step.
wyatt@Wall-E:~/ros_ws$ 452183621.541102   Duration: 5.701016 / 750.000001
```

The `rqt_console` display indicates that the bagfile has been opened, but no other information is displayed. What is happening at this point is that `rosbag` is publishing the recorded data (recorded from `topic1`) to `topic1` at the same rate as the data was recorded.

To see that this is taking place, halt all nodes (including `rosbag`). Run `rqt_console`. In a terminal window, start the subscriber, but not the publisher, using:

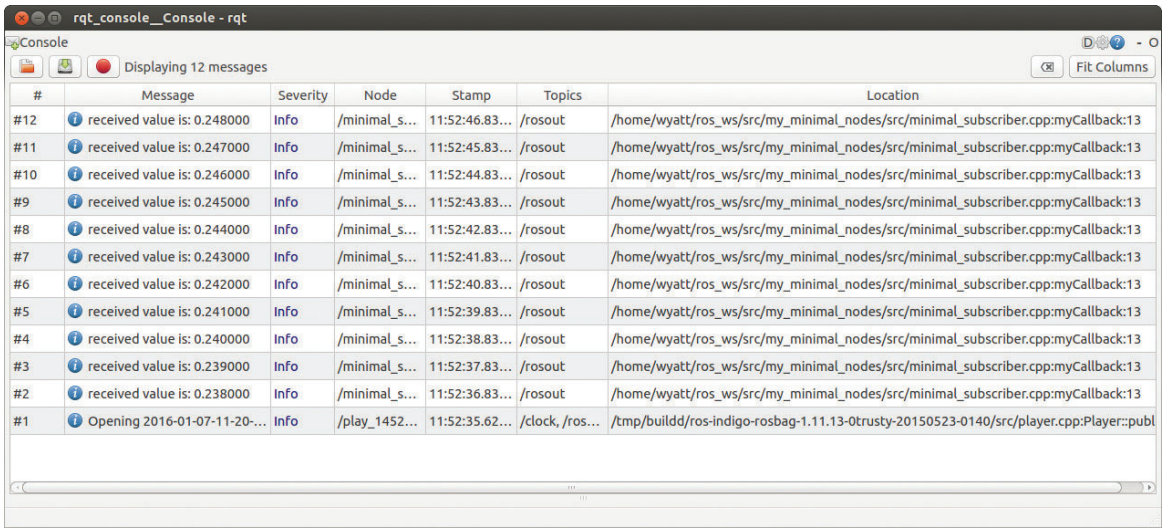
```
rosrun my_minimal_nodes my_minimal_subscriber
```

At this point, this terminal is suspended, because `topic1` is not yet active.

From another terminal, navigate to the `bagfiles` directory and enter:

```
rosbag play fname.bag
```

where "fname" is (again) the name of the recording that was bagged previously. `rosbag` now assumes the role formerly taken by `sleepy_minimal_publisher`, recreating the messages that were formerly published. The `my_minimal_subscriber` window reports the recorded data, updating once per second. `rqt_console` also shows the data posted by the minimal subscriber. As can be seen from the screenshot in Fig 1.4, the playback corresponds to the original recording, with the first output (console line 2) displaying a value of 0.238 from the subscriber node.



#	Message	Severity	Node	Stamp	Topics	Location
#12	received value is: 0.248000	Info	/minimal_s...	11:52:46.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#11	received value is: 0.247000	Info	/minimal_s...	11:52:45.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#10	received value is: 0.246000	Info	/minimal_s...	11:52:44.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#9	received value is: 0.245000	Info	/minimal_s...	11:52:43.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#8	received value is: 0.244000	Info	/minimal_s...	11:52:42.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#7	received value is: 0.243000	Info	/minimal_s...	11:52:41.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#6	received value is: 0.242000	Info	/minimal_s...	11:52:40.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#5	received value is: 0.241000	Info	/minimal_s...	11:52:39.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#4	received value is: 0.240000	Info	/minimal_s...	11:52:38.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#3	received value is: 0.239000	Info	/minimal_s...	11:52:37.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#2	received value is: 0.238000	Info	/minimal_s...	11:52:36.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#1	Opening 2016-01-07-11-20-...	Info	/play_1452...	11:52:35.62...	/clock, /ros...	/tmp/buildd/ros-indigo-rosbag-1.11.13-0trusty-20150523-0140/src/player.cpp:Player::publ

Figure 1.4: Output of `rqt_console` with `minimal_subscriber` and `rosbag play` of recorded (bagged) data

Dynamically, these values are posted at the original 1 Hz rate `sleepy_minimal_publisher` published them. The `rosbag` player terminates when it gets to the end of the recorded data.

Note that our subscriber is oblivious to what entity is publishing to `topic1`. Consequently, playback of previously recorded data is indistinguishable from receiving live data. This is very useful for development. For example, a robot may be teleoperated through an environment of interest while it publishes sensor data from cameras, LIDAR, etc. Using `rosbag`, this data may be recorded verbatim. Subsequently, sensor processing may be performed on the recorded data to test machine vision algorithms, for example. Once a sensory-interpretation node is shown to be effective on the recorded data, the same node may be tried verbatim on the robot system. Note that no changes to the developed node are needed. In live experiments, this node would merely receive messages published by the real-time system instead of by `rosbag playback`.

1.4 MINIMAL SIMULATOR AND CONTROLLER EXAMPLE

Concluding this introduction, we consider a pair of nodes that both publish and subscribe. One of these nodes is a minimal simulator, and the other is a minimal controller. The minimal simulator simulates $F = ma$ by integrating acceleration to update velocities. The input force is published to the topic `force_cmd` by some entity (eventually the controller). The resulting system state (velocity) is published to the topic `velocity` by the minimal simulator.

The simulator code is shown in Listing 1.8, and the source code is in the accompanying repository, in https://github.com/wsnewman/learning_ros/tree/master/Part_1/minimal_nodes/src.

The `main()` function initializes both a publisher and a subscriber. Formerly, we saw nodes as dedicated publishers or dedicated subscribers, but a node can (and often does) perform both actions. Equivalently, the simulator node behaves like a link in a chain, processing incoming data and providing timely output publications.

Another difference from our previous publisher and subscriber nodes is that both the callback function and the `main()` routine perform useful actions. The minimal simulator has a callback routine that checks for new data on the topic `force_cmd`. When the callback receives new data, it copies it to a global variable, `g_force`, so that the `main()` program has access to it. The `main()` function iterates at a fixed rate (set to 100 Hz). In order for the callback function to respond to incoming data, the `main()` function must provide `spin` opportunities. Formerly, our minimal subscriber used the `spin()` function, but this resulted in the `main()` function ceasing to contribute new computations.

An important new feature in the minimal simulator example is the use of the ROS function: `ros::spinOnce()`. This function, executed within the 100 Hz loop of the simulator, allows the callback function to process incoming data at 10 ms intervals. If a new input (a force stimulus) is received, it is stored in `g_force` by the callback function. In the complementary minimal controller node, values of force stimulus are published at only 10 Hz. Consequently, 9 out of 10 times there will be no new messages on the `force_command` topic. The callback function will not block, but neither will it update the value of `g_force` when there is no new transmission available. The main loop will still repeat its iterations at 100 Hz, albeit re-using stale data in `g_force`. This behavior is realistic, since a simulator should emulate realistic physics. For an actuated joint that is digitally controlled, controller effort (force or torque) commands typically behave with a sample-and-hold output between controller updates (sample periods).

Listing 1.8: Minimal Simulator

```
1  // minimal_simulator node:
2  // wsn example node that both subscribes and publishes
3  // does trivial system simulation, F=ma, to update velocity given F specified on topic↩
       "force_cmd"
4  // publishes velocity on topic "velocity"
5  #include<ros/ros.h>
6  #include<std_msgs/Float64.h>
7  std_msgs::Float64 g_velocity;
8  std_msgs::Float64 g_force;
9  void myCallback(const std_msgs::Float64& message_holder)
10 {
11 // checks for messages on topic "force_cmd"
12 ROS_INFO("received force value is: %f",message_holder.data);
13 g_force.data = message_holder.data; // post the received data in a global var for ↩
       access by
14 // main prog.
15 }
16 int main(int argc, char **argv)
17 {
18 ros::init(argc,argv,"minimal_simulator"); //name this node
19 // when this compiled code is run, ROS will recognize it as a node called "↩
       minimal_simulator"
20 ros::NodeHandle nh; // node handle
21 //create a Subscriber object and have it subscribe to the topic "force_cmd"
22 ros::Subscriber my_subscriber_object= nh.subscribe("force_cmd",1,myCallback);
23 //simulate accelerations and publish the resulting velocity;
24 ros::Publisher my_publisher_object = nh.advertise<std_msgs::Float64>("velocity",1);
25 double mass=1.0;
26 double dt = 0.01; //10ms integration time step
27 double sample_rate = 1.0/dt; // compute the corresponding update frequency
```

```
28 ros::Rate naptime(sample_rate);
29 g_velocity.data=0.0; //initialize velocity to zero
30 g_force.data=0.0; // initialize force to 0; will get updated by callback
31 while(ros::ok())
32 {
33 g_velocity.data = g_velocity.data + (g_force.data/mass)*dt; // Euler integration of
34 //acceleration
35 my_publisher_object.publish(g_velocity); // publish the system state (trivial--1-D)
36 ROS_INFO("velocity = %f",g_velocity.data);
37 ros::spinOnce(); //allow data update from callback
38 naptime.sleep(); // wait for remainder of specified period; this loop rate is faster ↩
       than
39 // the update rate of the 10Hz controller that specifies force_cmd
40 // however, simulator must advance each 10ms regardless
41 }
42 return 0; // should never get here, unless roscore dies
43 }
```

The minimal simulator may be compiled and run. Running `rqt_console` shows that the velocity has a persistent value of 0.

The result can be visualized graphically with the ROS tool `rqt_plot`. To do so, use command-line arguments for the values to be plotted, *e.g.*:

```
rqt_plot velocity/data
```

will plot the velocity command against time. This output will be boring, at present, since the velocity is always zero.

One can manually publish values to a topic from a command line. For example, enter the following command in a terminal:

```
rostopic pub -r 10 force_cmd std_msgs/Float64 0.1
```

This will cause the value 0.1 to be published repeatedly on the topic `force_cmd` at a rate of 10 Hz using the consistent message type `std_msgs/Float64`. This can be confirmed (from another terminal) with:

```
rostopic echo force_cmd
```

which will show that the `force_cmd` topic is receiving the prescribed value.

Additionally, invoking:

```
rqt_plot velocity/data
```

will show that the velocity is increasing linearly, and `rqt_console` will print out the corresponding values (for both force and velocity).

Instead of publishing force-command values manually, these can be computed and published by a controller. Listing 1.9 displays a compatible minimal controller node. (This code is also contained in the accompanying examples repository under https://github.com/wsnewman/learning_ros/tree/master/Part_1/minimal_nodes/src.)

The minimal controller subscribes to two topics, (`velocity` and `vel_cmd`), and publishes to the topic `force_cmd`. At each control cycle (set to 10 Hz), the controller checks for the latest system state (velocity), checks for any updates to the commanded velocity, and computes a proportional error feedback to derive (and publish) a force command. This simple controller attempts to drive the simulated system to the user-commanded velocity setpoint.

Again, the `ros::spinOnce()` function is used to prevent blocking in the timed main loop. Callback functions put received message data in the global variables `g_velocity` and `g_vel_cmd`.

Listing 1.9: Minimal Controller

```
// minimal_controller node:
// wsn example node that both subscribes and publishes--counterpart to minimal_simulator
// subscribes to "velocity" and publishes "force_cmd"
// subscribes to "vel_cmd"
#include<ros/ros.h>
#include<std_msgs/Float64.h>
//global variables for callback functions to populate for use in main program
std_msgs::Float64 g_velocity;
std_msgs::Float64 g_vel_cmd;
std_msgs::Float64 g_force; // this one does not need to be global...
void myCallbackVelocity(const std_msgs::Float64& message_holder)
{
// check for data on topic "velocity"
ROS_INFO("received velocity value is: %f",message_holder.data);
g_velocity.data = message_holder.data; // post the received data in a global var for access by
//main prog.
}
void myCallbackVelCmd(const std_msgs::Float64& message_holder)
{
// check for data on topic "vel_cmd"
ROS_INFO("received velocity command value is: %f",message_holder.data);
g_vel_cmd.data = message_holder.data; // post the received data in a global var for access by
//main prog.

 }
int main(int argc, char **argv)
{
ros::init(argc,argv,"minimal_controller"); //name this node
// when this compiled code is run, ROS will recognize it as a node called "minimal_controller"
ros::NodeHandle nh; // node handle
//create 2 subscribers: one for state sensing (velocity) and one for velocity commands
ros::Subscriber my_subscriber_object1= nh.subscribe("velocity",1,myCallbackVelocity);
ros::Subscriber my_subscriber_object2= nh.subscribe("vel_cmd",1,myCallbackVelCmd);
//publish a force command computed by this controller;
ros::Publisher my_publisher_object = nh.advertise<std_msgs::Float64>("force_cmd",1);
double Kv=1.0; // velocity feedback gain
double dt_controller = 0.1; //specify 10Hz controller sample rate (pretty slow, but
//illustrative)
double sample_rate = 1.0/dt_controller; // compute the corresponding update frequency
ros::Rate naptime(sample_rate); // use to regulate loop rate
g_velocity.data=0.0; //initialize velocity to zero
g_force.data=0.0; // initialize force to 0; will get updated by callback
g_vel_cmd.data=0.0; // init velocity command to zero
double vel_err=0.0; // velocity error
// enter the main loop: get velocity state and velocity commands
// compute command force to get system velocity to match velocity command
// publish this force for use by the complementary simulator
while(ros::ok())
{
vel_err = g_vel_cmd.data - g_velocity.data; // compute error btwn desired and actual
//velocities
g_force.data = Kv*vel_err; //proportional-only velocity-error feedback defines commanded
//force
my_publisher_object.publish(g_force); // publish the control effort computed by this
//controller
ROS_INFO("force command = %f",g_force.data);
ros::spinOnce(); //allow data update from callback;
naptime.sleep(); // wait for remainder of specified period;
}
return 0; // should never get here, unless roscore dies
}
```

Once the two nodes are compiled with `catkin_make` (which requires editing `CMakeLists.txt` to add these executables to the package), they can be run (with `rosrun`) from separate terminal windows (assuming `roscore` is running). Running `rqt_console` reveals that the force command is updated once for every 10 updates of velocity (as expected for the simulator at 100 Hz and the controller at 10 Hz).

The velocity command may be input from another terminal using a command line, *e.g.*:

```
rostopic pub r 10 vel_cmd  std_msgs/Float64 1.0
```

publishes the value 1.0 to the topic `vel_cmd` repeatedly at a rate of 10 Hz. Watching the output on `rqt_console` shows the velocity converge exponentially on the desired value of `vel_cmd`.

The result can be visualized graphically with the ROS tool `rqt_plot`. To do so, use command line arguments for the values to be plotted, for example:

```
rqt_plot vel_cmd/data,velocity/data,force_cmd/data
```

will plot the velocity command, the actual velocity and the force commands on the same plot versus time. For the minimal simulator and minimal controller, the velocity command was initially set to 0.0 via `rostopic pub`. Subsequently, the command was set to 2.0. A screenshot of the resulting `rqt_plot` is shown in Fig 1.5. The control effort (in red) reacts to accelerate the velocity closer to the goal of 2.0, then the control effort decreases. Ultimately, the system velocity converges on the goal value and the required control effort decreases to zero.

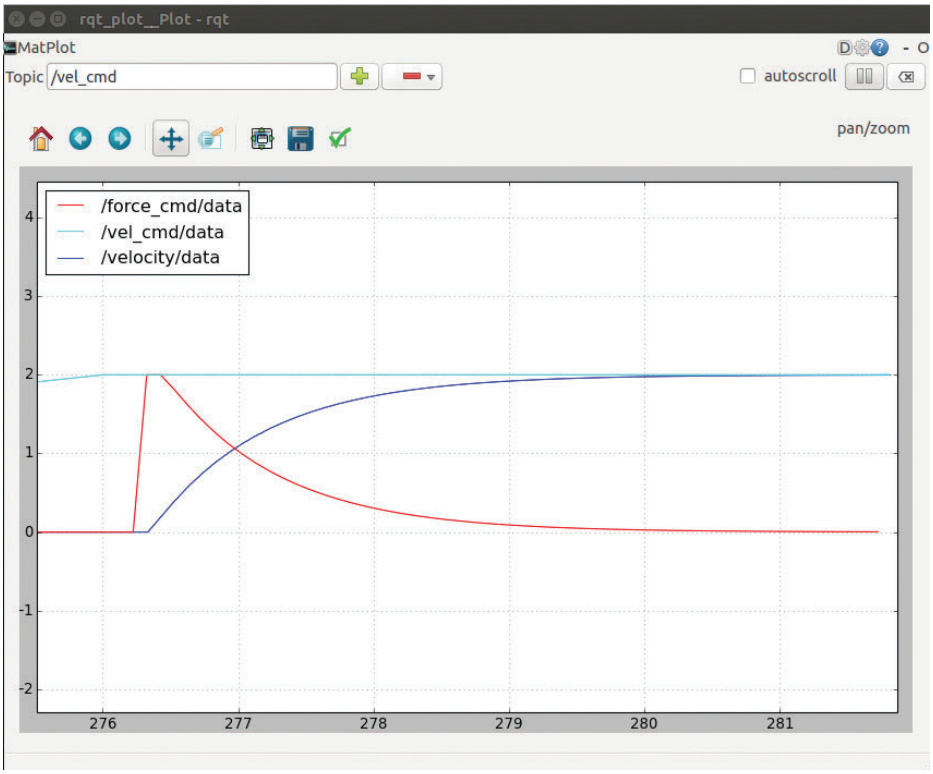


Figure 1.5: Output of `rqt_plot` with `minimal_simulator`, `minimal_controller` and step velocity-command input via console

1.5 WRAP-UP

This chapter has introduced the reader to some basics of ROS. It has been shown that multiple asynchronous nodes can communicate with each other through the mechanisms of publish and subscribe of messages on topics, as coordinated by `roscore`. Some ROS tools were introduced for the purpose of compiling nodes (`catkin_create_pkg`, `cs_create_pkg` and `catkin_make`), for logging and replaying published information (`rosbag`) and for visualizing ROS output (`rqt_console` and `rqt_plot`).

In the next chapter, additional communications topics are introduced, including defining custom message types, using client–server interactions, the common design paradigm of action servers, and the ROS parameter server.