

```
| sum_squared(4, 6)
```

prints the following to the console:

```
| RSS = 7.211102550927978
```

Alternatively, we could pass a value to `pre` with the call

```
| sum_squared(4, 6, pre="Root sum square =")
```

which prints

```
| Root sum square = 7.211102550927978
```

## 1.10 Branching



There are special statements in all programming languages that allow the programmer to control which portions are to be executed next (or at all); that is, the **control flow**. The primary forms of control flow statements are **branching** and **looping**, and we introduce branching in this section and looping in section 1.11.

### 1.10.1 Branching with `if/elif/else` Statements

Branching control flow statements are based on logical conditions that are tested by the statement. The primary branching statements in Python are the `if/elif/else` statements. For instance, consider the following statements:

```
| if x < 0:  
|     print("negative")  
| elif x == 0:  
|     print("zero")  
| else:  
|     print("positive")
```

If `x` is less than 0, it will print `negative`; if `x` is equal to 0, it will print `zero`, and otherwise (when `x` is positive) it will print `positive`. Note that the blocks of code that follow the branching statements must be indented. The `elif` (i.e., else if) and `else` statements are optional, and there can be multiple `elif` statements. Once a condition is met and the corresponding block executed, the rest of the control statements in the block are skipped.

The conditional expression is evaluated to a `bool` type (class). A `boolean` object can have one of two possible values, `True` and `False`. If the conditional expression of a branching statement evaluates to `True`, its corresponding block of code is executed. Note that Python will evaluate non-`boolean` conditional expression value with the built-in `bool()` function. For instance, if the conditional expression evaluates to a

string `"foo"`, it will be evaluated as `bool("foo")`, which, like all nonempty strings, evaluates to `True`. However, an empty string `""` evaluates to `False`.

### Example 1.5

Write and test a Python program that prints a string variable if it is nonempty, and prints `Empty string` otherwise.

We will want to test our program on a nonempty and an empty string, so we will want to reuse our code; this indicates the use of a function definition. Consider the following program:

```
def print_nonempty(s):
    if s:
        print(s)
    else:
        print("Empty string")

print_nonempty("This should print")
print_nonempty("")      # This should print "Empty string"
```

The `if` statement has conditional expression `s`, which should be a string. Therefore, if it is nonempty, `print(s)` will evaluate. Otherwise (i.e., if `s` is an empty string), the statement `print("Empty string")` will evaluate. As we expect, the program prints the following to the console:

```
This should print
Empty string
```

### 1.10.2 Branching with `match/case` Statements

In Python 3.10, a new kind of branching statement was introduced: `match/case`. Its use is never strictly necessary, but it can make a program more readable. For example,

```
if s == "red":
    print("red")
elif s == "blue":
    print("blue")
else:
    print("other")
```

can be written alternatively as

```
match s:
    case "red":
        print("red")
    case "blue":
        print("blue")
    case _:
        print("other")
```

In the third case `_` matches when there is no other match. Once there is a match, no other cases are tested. If there is no match (and `_` is not given as a case), none of the code blocks are evaluated.

There are more advanced uses of `match/case` statements in which patterns can be matched. See Python Community (2024a; § 4.6) for more details.

### 1.10.3 Branching with `try/except/finally` Statements

Sometimes a statement can yield an **exception**, which is not a syntax error, but has a similar effect in that it can stop the execution of the program. Common exceptions include `ZeroDivisionError`, `NameError` and `TypeError`.

In general, an exception stops the execution of a program; however, certain exceptions can be anticipated and dealt with accordingly, which is called exception handling. One of the primary ways to handle exceptions is to use the `try/except/finally` statements. We can think of these statements as branching statements that branch based on exceptions. For instance consider the following function definition:

```
def plus_7(x):
    try:
        y = x + 7
    except:
        y = x
    return y
```

If we can add 7 to `x`, which is the case when `x` is a number, the `try` statement will execute, the `except` statement will be skipped, and the sum will be returned. If, however, we cannot add 7 to `x`, which is the case when `x` is nonnumeric, the `try` statement will raise an exception, so the `except` statement will be executed; this returns the input without change.

We will later return to exception handling to consider more advanced usage, including the `finally` statement.

## 1.11 Looping



Repeating blocks of code by calling a function more than once, as in example 1.5, can get cumbersome when it needs to be repeated many times. A **loop** repeats a block until some stopping condition is met. One type of loop in Python is a **while** loop, which repeats a block of code while its conditional expression evaluates to **True**. For instance,

```
n = 0          # Initialize n
while n < 5:
    print(n)
    n += 1     # Increment n (i.e., n = n + 1)
```

The loop evaluates the conditional expression `n < 5` and, if in fact `n < 5`, executes the block of code. After the block finishes, the test is repeated and potentially the block of code. This will repeat indefinitely, until the conditional expression evaluates to **False**, in which case the loop exits and execution resumes after the code block. The block will be executed 5 times, printing 0 through 4 to the console.

Another type of Python loop is a **for** loop, which has no explicit conditional expression, instead iterating through an iterable object like a list, , until it reaches the end. For example,

```
l = ["foo", "bar", "baz"]
for s in l:
    print(f"Say {s}")
```

This prints

```
Say foo
Say bar
Say baz
```

It is common to loop through a **range** with a **for** loop, as in the following:

```
for k in range(2, 8):
    print(k, end=" ") # Prints on the same line
```

This prints the following to the console:

```
2 3 4 5 6 7
```

Often, a loop index is required inside a **for** loop. The syntax for this requires an identifier for the index and an **enumerate** type object to be iterated through. The constructor function `enumerate()` assigns an index to each element of its iterable argument (e.g., a list). For instance,