Chapter 15
Introduction to Functional Programming

15.1 Introduction

There has been much hype around Functional Programming in recent years. However, Functional Programming is not a new idea and indeed goes right back to the 1950s and the programming language LISP. However, many people are not clear as to what Functional Programming is and instead jump into code examples and never really understand some of the key ideas associated with Functional Programming such as Referential Transparency.

This chapter introduces Functional Programming (also known as FP) and the key concept of Referential Transparency (or RT).

One idea to be aware of is that Functional Programming is a software coding style or approach and is separate from the concept of a function in Python.

Python functions can be used to write functional programs but can also be used to write procedural style programs; so do not get too hung up on the syntax that might be used or the fact that Python has functions just yet. Instead explore the idea of defining a functional approach to your software design.

15.2 What is Functional Programming?

Wikipedia describes Functional Programming as:

> … a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids state and mutable data.

There are a number of points to note about this definition. The first is that it is focused on the *computational side* of computer programming. This might seem obvious but most of what we have looked at so far in Python would be considered procedural in nature.

© Springer Nature Switzerland AG 2023

J. Hunt, *A Beginners Guide to Python 3 Programming*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-35122-8_15

Another thing to note is that the way in which the computations are represented emphasizes functions that generate results based purely on the data provided to them. That is these functions only rely on their inputs to generate a new output. They do not rely on any *side effects* and do not depend on the *current state* of the program. As an example of a side effect, if a function stored a running total in a global variable and another function used that total to perform some calculation, then the first function has a side effect of modifying a global variable and the second relies on some global state for its result.

Taking each of these in turn:

1. **Functional Programming aims to avoid side effects**. A function should be replaceable by taking the data it receives and in lining the result generated (this is referred to as Referential Transparency). This means that there should be no hidden side effects of the function. Hidden side effects make it harder to understand what a program is doing and thus make comprehension, development and maintenance harder. Pure functions have the following attributes:

   2. The only observable output is the return value.
   3. The only output dependency is the arguments.
   4. Arguments are fully determined before any output is generated.

5. **Functional Programming avoids concepts such as state**. Let us take these as separate issues. If some operation is dependent upon the (potentially hidden) state of the program or some element of a program, then its behaviour may differ depending upon that state. This may make it harder to comprehend, implement, test and debug. As all of these impact on the stability and probably reliability of a system, state-based operations may result in less reliable software being developed. As functions do not (should not) rely on any given state (only upon the data they are given) they should as a result be easier to understand, implement, test and debug.
6. **Functional Programming promotes immutable data**. Functional Programming also tends to avoid concepts such as mutable data. Mutable data is data that can change its state. By contrast *immutability* indicates that once created, data cannot be changed. In Python strings are immutable. Once you create a new string you cannot modify it. Any functions that apply to a string that might conceptually alter the contents of the string and result in a new string being generated. Many developers take this further by having a presumption of immutability in their code; that means that by default all data holding types are implemented as immutable. This ensures that functions cannot have hidden side effects and thus simplifies programming in general.
7. **Functional Programming promotes declarative programming** which means that programming is oriented around expressions that describe the solution rather than focus on the imperative approach of most procedural programming languages. Imperative languages emphasize aspects of how the solution is

derived. For example, an imperative approach to looping through some container and printing out each result in turn would look like this:

```
int sizeOfContainer = container.length
for (int i = 1 to sizeOfContainer) do
   element = container.get(i)
   print(element)
enddo
```

Whereas a Functional Programming approach would look like:

```
container.foreach(print)
```

Functional Programming has its roots in the lambda calculus, originally developed in the 1930s to explore computability. Many Functional Programming languages can thus be considered as elaborations on this lambda calculus. There have been numerous pure Functional Programming languages including Common Lisp, Clojure and Haskell. Python provides some support for writing in the functional style, particularly where the benefits of it are particularly strong (such as in processing various different types of data).

Indeed, when used judiciously, Functional Programming can be a huge benefit for, and an enhancement to, the toolkit available to developers.

To summarize then:

- **Imperative Programming** is what is currently perceived as traditional programming. That is, it is the style of programming used in languages such as C, C++, Java and C#. In these languages a programmer tells the computer what to do. It is thus oriented around control statements, looping constructs and assignments.
- **Functional Programming** aims to describe the solution, that is, *what* the program needs to do (rather than *how* it should be done).

15.3 Advantages to Functional Programming

There are a number of significant advantages to Functional Programming compared to imperative programming. These include:

1. **Less code**. Typically, a Functional Programming solution will require less code to write than an equivalent imperative solution. As there is less code to write, there is also less code to understand and to maintain. It is therefore possible that functional programs are not only more elegant to read but easier to update and maintain. This can also lead to enhanced programmer productivity as they spend less time writing reams of code as well as less time reading those reams of code.
2. **Lack of (hidden) side effects (Referential Transparency)**. Programming without side effects is good as it makes it easier to reason about functions (that is a function is completely described by the data that goes in and the results that come back). This also means that it is safe to reuse these functions in different

situations (as they do not have unexpected side effects). It should also be easier to develop, test and maintain such functions.

3. **Recursion is a natural control structure**. Functional languages tend to emphasize recursion as a way of processing structures that would use some form of looping constructs in an imperative language. Although you can typically implement recursion in imperative languages, it is often easier to do in functional languages. It is also worth noting that although recursion is very expressive and a great way for a programmer to write a solution to a problem, it is not as efficient at runtime as iteration. However, any expression that can be written as a recursive routine can also be written using looping constructs. Functional Programming languages often incorporate *tail end recursive optimizations* to convert recursive routines into iterative ones at runtime. A util end recursive function is one in which the last thing a function does before it returns to call itself. This means that rather than actually invoking the function and having to set up the context for that function, it should be possible to reuse the current context and to treat it in an iterative manner as a loop around that routine. Thus the programmer benefits from the expressive recursive construct and the runtime benefits of an iterative solution using the same source code. This option is typically not available in imperative languages.
4. **Good for prototyping solutions**. Solutions can be created very quickly for algorithmic or behaviour problems in a functional language, thus allowing ideas and concepts to be explored in a rapid application development style.
5. **Modular functionality**. Functional Programming is modular in terms of functionality (where object-oriented languages are modular in the dimension of components). They are thus well suited to situations where it is natural to want to reuse or componentize the *behaviour* of a system.
6. **The avoidance of state-based behaviour**. As functions only rely on their inputs and outputs (and avoid accessing any other stored state) they exhibit a cleaner and simpler style of programming. This avoidance of state-based behaviour makes many difficult or challenging areas of programming simpler (such as those in concurrent applications).
7. **Additional control structures**. A strong emphasis is on additional control structures such as pattern matching, managing variable scope, tail recursion optimizations, etc.
8. **Concurrency and immutable data**. As Functional Programming systems advocate immutable data structures it is simpler to construct concurrent systems. This is because the data being exchanged and accessed is immutable. Therefore, multiple executing thread or processes cannot affect each other adversely. The Akka Actor model builds on this approach to provide a very clean model for multiple interacting concurrent systems.
9. **Partial evaluation**. Since functions do not have side effects, it also becomes practical to bind one or more parameters to a function at compile time and to reuse these functions with bound values as new functions that take fewer parameters.

15.4 Disadvantages of Functional Programming

If Functional Programming has all the advantages previously described, why isn't it the mainstream force that imperative programming languages are? The reality is that Functional Programming is not without its disadvantages, including:

- **Input–Output is harder in a purely functional language**. Input–Output flows naturally align with stream style processing, which does not neatly fit into the data in, results out, nature of functional systems.
- **Interactive applications are harder to develop**. Interactive applications are constructed via request response cycles initiated by a user action. Again, these do not naturally sit within the purely functional paradigm.
- **Continuously running programs** such as services or controllers may be more difficult to develop, as they are naturally based upon the idea of a continuous loop.
- **Functional Programming languages have tended to be less efficient on current hardware platforms**. This is partly because current hardware platforms are not designed with Functional Programming in mind and also because many of the systems previously available were focused on the academic community where out and out performance was not the primary focus. However, this has changed to a large extent with modern functional languages such as Scala and Heskell.
- **Not data oriented**. A pure functional language does not really align with the needs of the primarily data-oriented nature of many of today's systems. Many (most) commercial systems are oriented around the need to retrieve data from a database, manipulate it in some way and store that data back into a database. Such data can be naturally represented via objects in an object-oriented language.
- **Programmers are less familiar** with Functional Programming concepts and thus find it harder to pick up function-oriented languages.
- **Functional Programming idioms are often less intuitive** to (traditional) procedural programmers than imperative idioms which can make debugging and maintenance harder. Although with the use of a functional approach in many other languages now becoming more popular (including in Python) this trend is changing.
- Many Functional Programming languages have been viewed as **Ivory tower languages** that are only used by academics. This has been true of some older functional languages but is increasingly changing with the advent of languages such as Scala and with the facilities provided in more mainstream programming languages such as Python.

15.5 Referential Transparency

An important concept within the world of Functional Programming is that of Referential Transparency.

An operation is said to be *Referentially Transparent* if it can be replaced with its corresponding value, without changing the program's behaviour, for a given set of parameters.

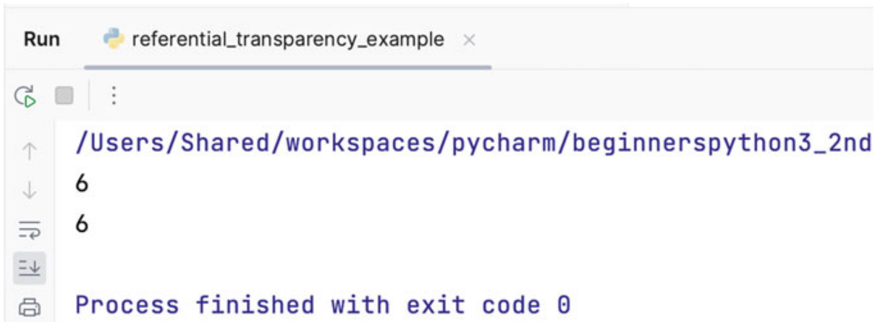
For example, let us assume that we have defined the function increment as shown below.

```
def increment(num):
    return num + 1
```

If we use this simple example in an application to increment the value 5:

```
print(increment(5))
print(increment(5))
```

We can say that the function is Referentially Transparent (or RT) if it always returns the same result for the same value (i.e., that `increment(5)` always returns 6):



Any function that references a value which has been captured from its surrounding context and which can be modified cannot be guaranteed to be RT. This can have significant consequences for the maintainability of the resulting code. This can happen if for example the increment function did not add 1 to the parameter but added a global value. If this global value is changed then the function would suddenly start to return different values for the previously entered parameters. For example, the following code is no longer Referentially Transparent:

```
amount = 1
def increment(num):
    return num + amount

print(increment(5))
amount = 2
print(increment(5))
```

The output from this code is not 6 and 7—as the value of amount has changed between calls to the `increment()` function.

A closely related idea is that of *No Side Effects*. That is, a function should not have any side effects, it should base its operation purely on the values it receives, and its only impact should be the result returned. Any hidden side effects again makes software harder to maintain.

Of course, within most applications there is a significant need for side effects; for example, any logging of the actions performed by a program has a side effect of updating some logged information somewhere (typically in a file), and any database updates will have some side effect (i.e., that of updating the database). In addition some behaviour is inherently non-RT; for example, a function which returns the *current* time can never be Referentially Transparent.

However, for pure functions it is a useful consideration to follow.

15.6 Further Reading

There is a large amount of material on the web that can help you learn more about Functional Programming including:

- https://codeburst.io/a-beginner-friendly-intro-to-functional-programming-4f69aa109569 intended as a friendly introduction to Functional Programming.
- https://medium.freecodecamp.org/an-introduction-to-the-basic-principles-of-functional-programming-a2c2a15c84 which provides an introduction to the basic principles of Functional Programming.
- https://www.tutorialspoint.com/functional_programming which provides a good grounding in the basic concepts of Functional Programming.
- https://docs.python.org/3/howto/functional.html which is the Python standard library tutorial on Functional Programming.