1 Programs that work

1.1 Introduction

This first part of the text deals with "ordinary" programs, those comprising assignment statements, conditionals and loops, and simple data structures (mainly)... and how to check that they work. The language for the program examples is `Python`, but the techniques apply to other "conventional" languages (such as Java and *C*) that also have those features. Program text will be usually written in `monospaced font`.[1]

Part of the novelty (perhaps) of what we are doing will be that we use "conditions", which can be thought of as Boolean-valued expressions, to write down what we expect the programs to achieve. What we call "checking" a program is simply making sure that the program *does* achieve those conditions. The conditions' syntax will be in the `Python` style as well, written just as you would write a condition in an `if`-statement: for example we'll write `x==2` instead of $x{=}2$, and `x!=2` instead of $x{\neq}2$.

Although our aim is to write programs that work, we will begin our study of "everyday programs" with a famous case of one that *didn't* work. It's not hard, of course, to find programs that don't work — we all write them every day, and the point of this text is to help us to do that less often. More unusual, though, is to find a "doesn't work" program that was in the pockets of millions of people. So that will be our first example.

Before we do that, however, see Ex. 1.1 at the end of the chapter for a preview of what "checking a program" might mean.[2] Ex. 1.1

[1] Comments about `Python` specifically will be mainly in footnotes, from here on.
[2] Boxed notes such as "Ex.1.1" suggest exercises particularly relevant to that part of the text. Some have answers included in Appendix I; but other answers are reserved for class exercises.

1.2 Jack be nimble, Jack be quick...

The following program was part of the operating system for a portable music player, of which millions were sold around the world. The variable `d` (for "day") was kept up to date by the player's hardware, giving at all times the day-number of "today", where Day 1 begins the "epoch" of 1 January 1980 and counts from there. When the player was switched on, one of its first tasks was to convert that internal day-number `d` to a displayable date in the calendar style: "the `D`th day of month `M` in year `YYYY`".

Below is the program it used to calculate the `YYYY` part of that:[3]

```
# --- If 1/1/1980 is Day 1,
# --- and today is Day d counting from there,
# --- determine the Year y of today.
y= 1980
while d>365:
  if leapYear(y):
    if d>366:
      d= d-366
      y= y+1
  else:
    d= d-365
    y= y+1
# Today is in Year y.
```
(1.1)

Today's date could then be displayed on the player's screen in conventional form (after subsequent code dealt with month-in-year and day-in-month).

Yet on New Year's Eve –31 December– in the first *leap year* after the music player was released, every single one of those players in the whole world froze when it was turned on. Looking above at (1.1), can you see why?

There are many posts about that error on the internet, and many blog comments attached to them where the program is "fixed" by one reader — only to have the next reader find a new mistake introduced by the previous repair. But the "why" of that error is not the actual mistake in the program: the real cause of it is that programmers are often not given the chance to learn how to avoid simple mistakes like that in the first place. It's the process that *led* to that program that needs to be debugged.

That program should never have been installed. And even if the programmer was new, or inexperienced (but we do not know), more experienced colleagues should have been able to ask simple questions –as part of a code review, a checking process– that would have revealed the problem in spite of possible opinions that it was too obviously correct to be worth the attention.

> The problem is the culture that allowed that program to be written, installed and deployed.
>
> *Jack should have been encouraged to be less nimble, not so quick, and should have been shown how to be more careful.*

Can we do better than Jack? Yes. This book shows how.

[3] The original program was written in *C*; aside from that, this transliteration into `Python` is believed to be the exact code, except for the added comments.

1.3 When does a program "work"?

Suppose we have a sequence `A[0:N]` of numbers, indexed from `0` inclusive to `N` exclusive, and we want to calculate their sum. Does this program work?

```
s= 0 # Start from zero.
for n in range(N):
  s= s+A[n] # Add the current element.
```
(1.2)

To answer that question, we must check various things.

Is the intended sum `s` correctly initialised? *Yes.* Are all the sequence elements `A[n]` of `A` considered, exactly once? *Yes.* And does the `n`-indexing remain within the bounds of the sequence? (We remind ourselves that the elements of `A[0:N]` are indexed from `0` up to `N-1`, and that `n in range(N)` iterates over `0,1,...,N-1`.) Again *Yes.* Is the correct operator `+` being used in the loop body? *Yes.*

So –overall– *Yes.* It does work. For a program of that (tiny) size, we can be sure we've checked everything.

Or does it work, really... *What does that mean?* It does run without crashing. But you can't say that something works (or doesn't) *unless* you know what it's supposed to do. And there's nothing "officially" associated with (1.2) that says what it's actually *for*. All we have is the text "...we have a sequence `A[0:N]` of numbers, and we want to calculate their sum." written in a paragraph nearby (in this case, just above the program code). What if there had been a page break in the source-text file, in between? When asking whether a program works we have to point not only to the program, but also to a description of what it's supposed to do. (See Sec. 18.2.3.)

So it's probably a good idea to include a (rough) description of the program's purpose as *part of the program itself*, say as a comment at its beginning. That would give us

```
# --- Sum the elements of sequence A[0:N].
s= 0
for n in range(N): s= s+A[n]    .
```
(1.3)

(We have removed the two comments that were there, possibly pointless, not really doing anything useful; and so the `for`-loop can now be all on one line.)

And so we can *now* ask simply "Does (1.3) work?" and it *does* make sense, provided we know we are following the convention that the comment at the front (for now, its *only* comment) is supposed to give the program's purpose. That is, the first line is "officially" where the purpose is to be stated, the *requirements*; and we have here the convention that the " `# ---` " in the comment identifies it as such. (Any other similar convention would do: we are not quibbling about syntax.)

More than that, though, we'll also write a comment at the *end* of the program that states, perhaps in more precise and technical terms, what the final state of the program is supposed to have achieved. That gives us the program

```
# --- Sum the elements of sequence A[0:N].
s= 0
for n in range(N): s= s+A[n]
# s == ∑A[0:N] # We call this a "postcondition".
```
(1.4)

in which $\sum$ means "the sum of". It's a more technical comment, at the end because it is telling us what we expect to be true once the program's execution has finished. It can be thought of as a more precise, more detailed version of the `# ---` comment at the beginning.

Later we will see many more reasons that putting what we call a "postcondition" there, at the end, is a good idea.

1.4 Requirements and specifications: pre- and postconditions

In careful programming, the comment at (1.4)

```
# --- Sum the elements of sequence A[0:N].
```
(1.5)

might be called the *requirements*, and the comment

```
# s == ∑A[0:N]
```
(1.6)

would probably be called a *postcondition*. The requirements say what the program is supposed to achieve, and the postcondition states something that is supposed to be true (a Boolean *condition*) once the program has finished (*post*). That is, the requirements (1.5) express what our customer (or boss) wants the program for: and "gathering" requirements is an important topic on its own. (See Sec. 19.2.1.) The postcondition (1.6) states what the program will have achieved (made true) when it has ended.

Obviously, requirements and postconditions are related; but they do not have exactly the same purpose. We will explain at the end of this section what the difference between those two things is.

For now, however, we look at a third feature — *preconditions*. They go at the front of the program, and interact with postconditions in the following way. Suppose we have the slightly different program shown here, for finding the largest element in a sequence:

```
# --- Find the largest element of sequence A[0:N].
m= A[0]
for n in range(1,N): m= m max A[n] # Note! A[N] is not included.
# m == MAX A[0:N]    ,
```
(1.7)

where in general we are writing $a \max b$ for the maximum of a and b and $\text{MAX}\, A$ for the overall maximum of a sequence A of values.[4]

If we check that program, as we checked (1.2) at the beginning of Sec. 1.3, we would look at similar things; but here we have deliberately introduced a slight complication. The requirements comment `# ---` doesn't make sense if the sequence is empty; that is, there can't be a *largest* element if there is no element at all. So –if we are to meet the requirements– we must somehow make sure that the program is never run when `A` is empty, that is when `N` is zero... at least not by anyone who expects it to work. *That's what the precondition is for.*

Just as a postcondition states what should be true at a program's end, a precondition states what should be true at its beginning. To make it clear which is which, we will (often, but not always) write PRE and POST within the comments:

```
# PRE: 0<N
m= A[0]
for n in range(1,N): m= m max A[n]
# POST: m == MAX A[0:N]    .
```
(1.8)

One could read the whole of (1.8) as

[4] In Python "max" is written as a function; thus in the program we would find `max(m,A[n])`.

"If `0<N` is true at the beginning (Pre) of the program

```
m= A[0]
for n in range(1,N): m= m max A[n]
```

then '`m ==` the largest element in `A[0:N]`' will be true at its end (Post)."

Reading it in that way, i.e. saying "if" this "then" that, makes it clear who has responsibility for making those conditions true: the person who is *using* the program must ensure that the precondition is met *if* she wants to be sure that the program will operate correctly — that is, after all, exactly what "if" means. And the person who wrote the program must make sure that *if* its precondition holds *then* its postcondition will hold once the program ends. In the case of (1.8), that means "If `0<N` then the program will set `m` to the largest element in `A[0:N]`."

But if (1.8) is run when its precondition does not hold, what then? In that case, the program (and its programmer) is absolved from any responsibility for what happens. That takes us back to the requirements: does the program meet them? They were there in (1.7), that is

```
# --- Find the largest element of sequence A[0:N].
```

and it's now clear that they are too strong: the requirements should read instead

```
# --- Find the largest element of non-empty sequence A[0:N].
```
(1.9)

and our overall program becomes

Ex. 1.2

```
# --- Find the largest element of non-empty sequence A[0:N].
# Pre: 0<N
m= A[0]
for n in range(1,N): m= m max A[n]
# Post: m == max A[0:N]
```
(1.10)

That leaves "specifications" as the last keyword in the heading of this section. We'll call the pre- and postcondition pair together the *specification*: you could think of it as

$$\textit{precondition} \implies (\text{program}) \implies \textit{postcondition} \quad ,$$

where if a program is sitting between the two conditions then it should "satisfy" the specification that the pre- and postcondition together express. The difference between the requirements and the specification $\textit{precondition} \Rightarrow \cdots \Rightarrow \textit{postcondition}$ is then that the requirements can be thought of as the purpose of the program, and the specification can be thought of as a contract — one that the programmer has checked the program is guaranteed to meet (as in Meyer's "Design by Contract" [39]).

It's crucial that the specification should imply that the requirements are met — but, of course, also that the requirements in turn capture accurately what the "customer" wants. (That's part of what is called "requirements analysis", sometimes "validation" — see Chapter 19.) And of course it's just as crucial that the program be guaranteed to satisfy its (validated) specification. (That's "program correctness", sometimes called "verification".) But those are two separate aspects. In the next section, we will take a closer look at the second of those: checking that the program meets, "satisfies", its specification.

That is the programmer's responsibility: *your* responsibility.

1.5 Satisfying a specification: assertions

We now return to the original (1.2), but suppose additionally that `N` is 3 (an arbitrary choice) so that we can "unroll" the program. Because `A[0:N]` is now `A[0:3]` –i.e. it is `[ A[0],A[1],A[2] ]`– we get

```
# Pre: True
s= 0
s= s+A[0]
s= s+A[1]
s= s+A[2]
# Post: s == ∑A[0:3]     ,
```
(1.11)

where we have put a precondition `True` to indicate unambiguously that the precondition hasn't simply been left off by accident.[5] Does that program (the middle four lines) satisfy its specification (the first- and last lines taken together)?

It's intuitively clear that it does (once we remember –again– that the last element of `A[0:3]` is `A[2]`). But there is now a sense in which we can check that (can"verify" it), even though for such a small program it seems hardly worth it. Yet we do it anyway, because we want to see exactly how it is done. Start by splitting it up into smaller pieces.

The small program (1.11) is actually four tiny programs one after the other, and we can give each one a specification of its own. Instead of writing them as conventional comments, here we write them between braces `{···}` to help separate them textually from the program code in between. Their meaning is the same:

```
          {Pre: True}     s= 0     {Post: s == ∑A[0:0]}
{Pre: s == ∑A[0:0]}   s= s+A[0]   {Post: s == ∑A[0:1]}
{Pre: s == ∑A[0:1]}   s= s+A[1]   {Post: s == ∑A[0:2]}
{Pre: s == ∑A[0:2]}   s= s+A[2]   {Post: s == ∑A[0:3]}
```

The precondition of the first program is `True`, meaning that we are assuming nothing (because `True` is true no matter what the variables' values are). After that, though, the *post*-condition of each smaller program is the *pre*-condition of the next one, until –at the end– the final postcondition, i.e. of the final statement, is the postcondition of the whole original program. It's like passing a "baton of correctness".

Let's now look at the small program `s= s+A[1]` all on its own. If we write it vertically, we can include our conditions as comments again:

```
# Pre: s == ∑A[0:1]
s= s+A[1]
# Post: s == ∑A[0:2]     .
```
(1.12)

Its *pre* condition `s == ∑A[0:1]` will be true just before it is executed, because it is exactly the *post* condition of the previous program `s= s+A[0]` . And if that fragment (1.12) is itself correct, it will establish *its* postcondition `s == ∑A[0:2]`, which is then ready "to be passed" to the `s= s+A[2]` that follows. So each fragment "does its bit", and the next fragment assumes that "the previous bit" has been done.

But *does* the small program (1.12) work? (Yes.) How do we check that?

[5] We write `True`, i.e. in "`Python` font" as here, when it is a condition within a `Python` program that is identically true. See also App. D.1.

To find out whether (1.12) works, we take the postcondition `s == ∑A[0:2]` and simply make the textual substitution given by the assignment statement, that is replacing `s` by `s+A[1]`. It's as simple as that, just doing it with a text editor: *find* `s`, then *replace* `s` *with* `s+A[1]`. Literally: just find, then replace. The new text is then `s+A[1] == ∑A[0:2]` . Once that's done, we check that the new, edited, text is implied by (⇛) the precondition;[6] that is, we check that Dr. A.1 Dr. A.2

$$\underset{\text{—antecedent—}}{\texttt{s == ∑A[0:1]}} \quad \overset{\text{implies}}{\Rrightarrow} \quad \underset{\text{—consequent—}}{\texttt{s+A[1] == ∑A[0:2]}} \quad , \tag{1.13}$$

which is to say that "Whenever the *antecedent* (to the left of the ⇛ arrow symbol) is true, then so is the *consequent* (to the right of the ⇛ arrow symbol)." After some simplification, that boils down to just

$$\texttt{(∑A[0:1]) + A[1]} \quad \texttt{==} \quad \texttt{∑A[0:2]} \quad ,$$

and of course checking the other three small programs (i.e. including the initialisation `s= 0`) is just as simple, as you can see in App. B.1.1.

But now here is the magic: although we checked those program fragments separately,[7] we can be sure they will work when put all together — *without checking anything further* except that each postcondition implies the precondition immediately following it.

All of that is an example of decomposing one *fairly* simple problem into four *extremely* simple ones. Later however we will use the same technique to decompose not-so-simple problems into a number of simple-but-not-trivial problems. And later still, we will decompose complex problems into a number of less complicated ones, and those into less complicated ones still... until eventually we reach (at last) the extremely trivial again.

Overall, therefore, we are decomposing a large single complicated problem into a (large) number of very simple ones: it's like a tree with a large trunk (a complicated problem) whose branches split, and split again — getting smaller and smaller until we reach the leaves (many simple problems).

Because we have become used to preconditions and postconditions, and to writing them either as comments `# ···` or between braces `{···}`, just as we like, and to suppressing the Pre and Post annotations — we'll now write just

```
{ True }
s= 0
{ s == ∑A[0:0] }
s= s+A[0]
{ s == ∑A[0:1] }      line 5
s= s+A[1]             line 6
{ s == ∑A[0:2] }      line 7
s= s+A[2]
{ s == ∑A[0:3] }
```

(1.14) → Recall program (1.12) above.

for the example above. The items `{···}` are called *assertions*, and each one is at the same time the postcondition `{Post: ···}` of the program statement before it

[6] The implication "⇛" is discussed more thoroughly in App. D.10.1. For now, read it as "if ... then".
[7] They could even have been checked by different people, at different times and in different places.

and the precondition `{PRE: ···}` of the one after: an example is lines 5–7 of the above, which together give our earlier (1.12) once we re-insert the PRE and the POST annotations and use the vertical format. If we wrote the whole thing out in that style, it would become

```
# PRE: True
s= 0
# POST: s == ∑A[0:0]

# PRE: s == ∑A[0:0]
s= s+A[0]
# POST: s == ∑A[0:1]
                                  } → These are the same.        (1.15)
line 5  # PRE: s == ∑A[0:1]
line 6  s= s+A[1]
line 7  # POST: s == ∑A[0:2]

# PRE: s == ∑A[0:2]
s= s+A[2]
# POST: s == ∑A[0:3]      ,
```

and that is why the whole, single program is correct provided each of its four smaller components is correct.

1.6 Turn the handle, and watch the program flow

An easy way to visualise (1.14) is to imagine its execution flowing from one statement to the next: we just follow the program's progress, the *program counter's* position, as it moves from the program's beginning to its end.[8]

When the program counter "flows through a *statement*", the statement is executed; and when it flows through an *assertion*, that assertion must be true. The second of those –that the assertion must be true– is the key to rigorous reasoning about programs of this kind.[9]

If each statement does its job, then the truth of its precondition guarantees the truth of its postcondition, which becomes in turn the precondition of the next statement... and so on. Taken all together, the truth initially of the very first precondition, i.e. of the first statement in the program and thus the precondition of the *whole* program, leads inevitably via this "baton passing" of what's-true-here assertions to the truth finally of the postcondition of the very last statement — which is the postcondition of the whole program.

It's like lining up dominoes, checking the adjacent pairs separately to be sure that as each one falls it is certain to strike the next. If *all* adjacent pairs have been checked, even if by different people at different times then pushing the very first "precondition domino" –no matter how many dominoes there are– will eventually cause the postcondition domino to fall.

[8] The *program counter* points to the program statement about to be executed.
[9] That second, complementary point of view was proposed by Robert W. Floyd — and it changed *everything*.

1.7 Assertions for loops: invariants

Our original (1.2) from Sec. 1.3 was a loop, and to discuss its correctness we added pre- and postconditions (1.15) and then "unrolled" the loop. We now repeat that, but this time we include the variable `n` that is initialised and then incremented "behind the scenes" by the iteration `for n in range(N)`. The initial (multiple) assignment `s,n= 0,0` sets both `s` and `n` to `0` and establishes the postcondition (of the initialisation), i.e. that `s == ∑A[0:n]` which, because `n==0` at that point, is the same as our earlier `s == ∑A[0:0]`. And all the subsequent assertions `s == ∑A[0:1]` and `s == ∑A[0:2]` and `s == ∑A[0:3]` are also replaced by the same `s == ∑A[0:n]`, and for the same reason:

```
{Pre: True}
s,n= 0,0                          ↓ Now these are all the same.
{Post: n==0 and s==∑A[0:n]} [10]

{Pre: s==∑A[0:n]}
s,n= s+A[0],n+1
{Post: n==1 and s==∑A[0:n]}                                        (1.16)

{Pre: s==∑A[0:n]}
s,n= s+A[1],n+1
{Post: n==2 and s==∑A[0:n]}

{Pre: s==∑A[0:n]}
s,n= s+A[2],n+1
{Post: n==3 and s==∑A[0:n]}        .
```

But we can go even further: we can replace the `A[0]` and `A[1]` etc. in the assignments by `A[n]` in all cases, and we can remove the `n==0` and `n==1` etc. from the assertions. That gives

```
{Pre: True}
s,n= 0,0                          } → This is the initialisation, but...
{Post: s==∑A[0:n]}

{Pre: s==∑A[0:n]}
s,n= s+A[n],n+1                   } → ...this statement, and
{Post: s==∑A[0:n]}

{Pre: s==∑A[0:n]}
s,n= s+A[n],n+1                   } → ...this statement and
{Post: s==∑A[0:n]}

{Pre: s==∑A[0:n]}
s,n= s+A[n],n+1                   } → ...this statement are all the same.
{Post: s==∑A[0:n]}

# But how do we know that n==3 here?
```

And so we find that all the component programs –and their assertions– are now the same (except the initialisation). If we could somehow add the postcondition `{n==3}` at the very end, we would be able to assert `{s==∑A[0:3]}` there as well.

[10] For example, in `Python` "`and`" is used rather than "`&&`" for *conjunction* (App. E.2.1): it makes a single condition that holds just when both of its components (conjuncts) do.

As a final step, to recover our final assertion `{n==3}`, we'll "roll (1.16) up" again, but use a `while`-loop rather than a `for`-loop: the advantage of the `while`-loop is that it makes the incrementing and testing of the loop index `n` explicit. With coloured program statements, making them stand out from the assertions, that gives us

Ex. 1.3 Ex. 1.4 Ex. 1.5

```
{Pre: 0<=N}
s,n= 0,0
{Post: 0<=n<=N and s == ∑A[0:n]}

{Pre: 0<=n<=N and s == ∑A[0:n]}    ← so-called loop invariant
while n!=N:
  {n!=N and 0<=n<=N and s == ∑A[0:n]}
                                                              (1.17)
  {Pre: 0<=n<N and s == ∑A[0:n]}
  s,n= s+A[n],n+1
  {Post: 0<=n<=N and s == ∑A[0:n]}    ← Post of loop body, invariant again

{Post: n==N and 0<=n<=N and s == ∑A[0:n]}    ← Post of whole loop
# Assertion just above ↑ must imply (⇒) the one just below ↓.[11]
{Post: s == ∑A[0:N]}    ← Post of whole program
```

Ex. 1.6 The assertion `{0<=n<=N and s == ∑A[0:n]}` just before the `while`-loop and at the end of its body (but still within the loop) is the *loop invariant*. Just after the `while`-loop however, at the beginning of the body, the invariant is joined (with an `and`) to the *loop guard* `n!=N`, the condition which determines whether the loop is entered; and after the *whole* loop it is joined with the *negation* of the loop guard `n==N`. (The first `Post` reminds us that the assertion there is at the end of the loop body; the second `Post` reminds us that the assertion is at the end of the whole loop; and the third `Post` is at the end of the whole program. It's `Post` in all cases, but of different program fragments.)

The `while` statement –on its own, without its body– appears to have *two* postconditions, because it can send the program execution in either of two directions: in one of them it enters the loop, where the loop guard `n!=N` is added; and in the other it exits the loop, where the negation of the guard, i.e. `n==N`, is added.

Again, the program has been split up into simpler pieces; but this time the number of pieces does not depend on `N`, as it did before (where for example there were `N+1`, i.e. four pieces when `N` was 3). Instead of having to write out an instance of the loop body for every time it's executed, we can write it just once — because the pre- and postconditions are the same every time:

– Initialisation

$$\{\text{Pre: } 0\le N\}\ s,n=0,0\ \{\text{Post: } 0\le n\le N \text{ and } s = \textstyle\sum A[0:n]\}$$

If we treat the assignment statement just as we did at (1.13), and make the substitution, we need only check that

$$0\le N \quad\Rightarrow\quad 0\le 0 \text{ and } 0\le N \text{ and } 0 = \textstyle\sum A[0:0] \quad ,$$

which is indeed easy.

– Test for loop entry

$$\{\text{Pre: } \cdots\}\ \text{while } n\ne N\ \{n\ne N \text{ and } \cdots\ n\le N\ \cdots\}\ \text{(loop body)}$$

[11] When assertions are vertical, each must imply (⇒) the one below.