Chapter 1
Why should you learn to write programs?

Writing programs is a very creative and rewarding activity.
You can write programs for many reasons

- ranging from making your living to solving a difficult data analysis problem
- having fun to helping someone else solve a problem.

The hardware in our current-day computers is essentially built to continuously ask us the question, **"What would you like me to do next?"**

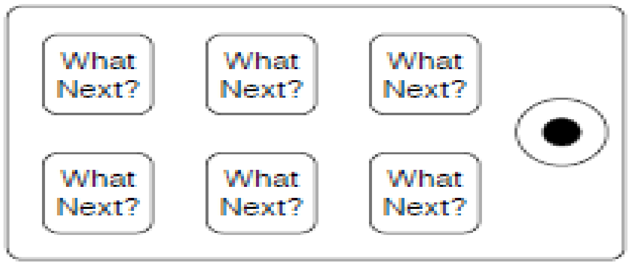


Figure 1.1: Personal Digital Assistant

Programmers add an operating system and a set of applications to the hardware and we end up with a Personal Digital Assistant that is quite helpful and capable of helping us do many different things.

Our computers are fast and have vast amounts of memory and could be very helpful to us if we only knew the language to speak to explain to the computer what we would like it to "do next".

1.1 Creativity and motivation

- Building useful, elegant, and clever programs for others to use is a very creative activity.

  Eg Your computer or Personal Digital Assistant (PDA) usually contains many different programs from many different groups of programmers, each competing for your attention and interest. They try their best to meet your needs and give you a great user experience in the process.

- Primary motivation is to be more productive in handling the data and information that we will encounter in our lives

1.2 Computer hardware architecture

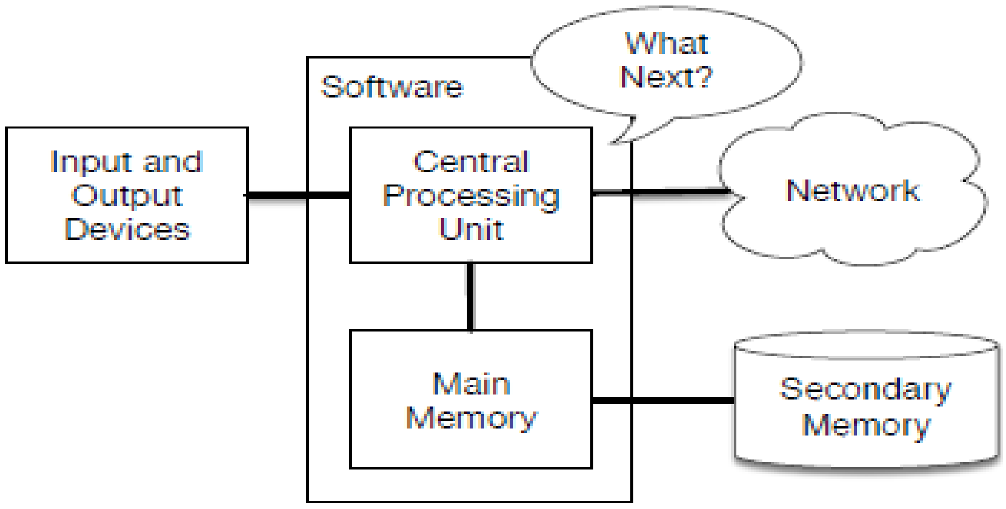


Figure 1.3: Hardware Architecture

• **The *Central Processing Unit* (or CPU)**
  - is the part of the computer that is built to be obsessed with “what is next?”
  - If your computer is rated at 3.0 Gigahertz, it means that the CPU will ask “What next?” three billion times per second.
  - You are going to have to learn how to talk fast to keep up with the CPU.

• **The *Main Memory***
  - is used to store information that the CPU needs in a hurry.
  - The main memory is nearly as fast as the CPU.
  - But the information stored in the main memory vanishes when the computer is turned off.

• **The *Secondary Memory***
  - is also used to store information, but it is much slower than the main memory.
  - The advantage of the secondary memory is that it can store information even when there is no power to the computer.
  - Examples of secondary memory are disk drives or flash memory (typically found in USB sticks and portable music players).

• **The *Input and Output Devices***
  - are simply our screen, keyboard, mouse, microphone, speaker, touchpad, etc.

- They are all of the ways we interact with the computer.

• These days, most computers also have a ***Network Connection***
- to retrieve information over a network.
- We can think of the network as a very slow place to store and retrieve data that might not always be “up”.
- The network is a slower and at times unreliable form of ***Secondary Memory***.

1.3 Understanding programming

You need two skills to be a programmer:

• First, you need to know the programming language (Python) -
- You need to know the vocabulary and the grammar.
- You need to be able to spell the words in this new language properly and know how to construct well-formed “sentences” in this new language.

• Second, you need to “tell a story”.
- In writing a story, you combine words and sentences to convey an idea to the reader.
- There is a skill and art in constructing the story, and skill in story writing is improved by doing some writing and getting some feedback.
- In programming, our program is the “story” and the problem you are trying to solve is the “idea”.

Once you learn one programming language such as Python, you will find it much easier to learn a second programming language such as JavaScript or C++.

1.4 Words and sentences
The reserved words in the language where humans talk to Python include the following:

```
and       del      global    not      with
as        elif     if        or       yield
assert    else     import    pass
break     except   in        raise
class     finally  is        return
continue  for      lambda    try
def       from     nonlocal  while
```

Eg for a sentence in python

```
print('Hello world!')
```

Sentence starts with the function *print* followed by a string of text of our choosing enclosed in single quotes.

1.5 Conversing with Python

- The >>> prompt is the Python interpreter's way of asking you, "What do you want me to do next?"
  Eg
  ```
  >>> print('Hello world!')
  Hello world!

  >>> print('You must be the legendary god that comes from the sky')
  You must be the legendary god that comes from the sky
  >>> print('We have been waiting for you for a long time')
  We have been waiting for you for a long time
  >>> print('Our legend says you will be very tasty with mustard')
  Our legend says you will be very tasty with mustard
  >>> print 'We will have a feast tonight unless you say
          File "<stdin>", line 1
          print 'We will have a feast tonight unless you say
                  ^
          SyntaxError: Missing parentheses in call to 'print'
  >>>
  ```
- Python is amazingly complex and powerful and very picky about the syntax you use to communicate with it
- Python is *not* intelligent. You are really just having a conversation with yourself, but using proper syntax.

The proper way to quit python

```
>>> quit()
```

The proper way to say "good-bye" to Python is to enter *quit()* at the interactive chevron >>> prompt.

1.6 Terminology: interpreter and compiler

- The CPU understands a language we call *machine language*.
- Machine language is very simple and frankly very tiresome to write because it is represented all in zeros and ones:

  001010001110100100101010000001111
  11100110000011101010010101101101

- Machine language seems quite simple on the surface, given that there are only zeros and ones, but its syntax is even more complex and far more intricate than Python.
- Instead we build various translators to allow programmers to write in high-level languages like Python or JavaScript and these translators convert the programs to machine language for actual execution by the CPU.
- Since machine language is tied to the computer hardware, machine language is not *portable* across different types of hardware.
- Programs written in high-level languages can be moved between different computers by using a different interpreter on the new machine or recompiling the code to create a machine language version of the program for the new machine.

These programming language translators fall into two general categories:

(1) interpreters

(2) compilers.

(1) Interpreters

- An *interpreter* reads the source code of the program as written by the programmer, parses the source code, and interprets the instructions on the fly.
- Python is an interpreter and when we are running Python interactively, we can type a line of Python (a sentence) and Python processes it immediately and is ready for us to type another line of Python.
- Some of the lines of Python tell Python that you want it to remember some value for later.
- We need to pick a name for that value to be remembered and we can use that symbolic name to retrieve the value later.
- We use the term *variable* to refer to the labels we use to refer to this stored data.

```
>>> x = 6
>>> print(x)
6
>>> y = x * 7
>>> print(y)
42
>>>
```

- In this example, we ask Python to remember the value six and use the label *x* so we can retrieve the value later.
- We verify that Python has actually remembered the value using *print*.
- Then we ask Python to retrieve *x* and multiply it by seven and put the newly computed value in *y*.
- Then we ask Python to print out the value currently in *y*.
- Even though we are typing these commands into Python one line at a time, Python is treating them as an ordered sequence of statements with later statements able to retrieve data created in earlier statements.
- The Python interpreter is written in a high-level language called "C".

(2) Compilers.

- A *compiler* needs to be handed the entire program in a file, and then it runs a process to translate the high-level source code into machine language and then the compiler puts the resulting machine language into a file for later execution.

1.7 Writing a program

- When we want to write a program, we use a text editor to write the Python instructions into a file, which is called a *script*.
- By convention, Python scripts have names that end with .py.
- To execute the script, you have to tell the Python interpreter the name of the file.
- In a Unix or Windows command window, you would type python hello.py as follows:

```
csev$ cat hello.py
print('Hello world!')
csev$ python hello.py
Hello world!
csev$
```

- The “csev$” is the operating system prompt, and the “cat hello.py” is showing us that the file “hello.py” has a one-line Python program to print a string.
- We call the Python interpreter and tell it to read its source code from the file “hello.py” instead of prompting us for lines of Python code interactively.

1.8 What is a program?

- The definition of a *program* at its most basic is a sequence of Python statements that have been crafted to do something. Even our simple *hello.py* script is a program.
- It is a one-line program and is not particularly useful, but in the strictest definition, it is a Python program.
- For example, look at the following text about a clown and a car.
- Look at the text and figure out the most common word and how many times it occurs.

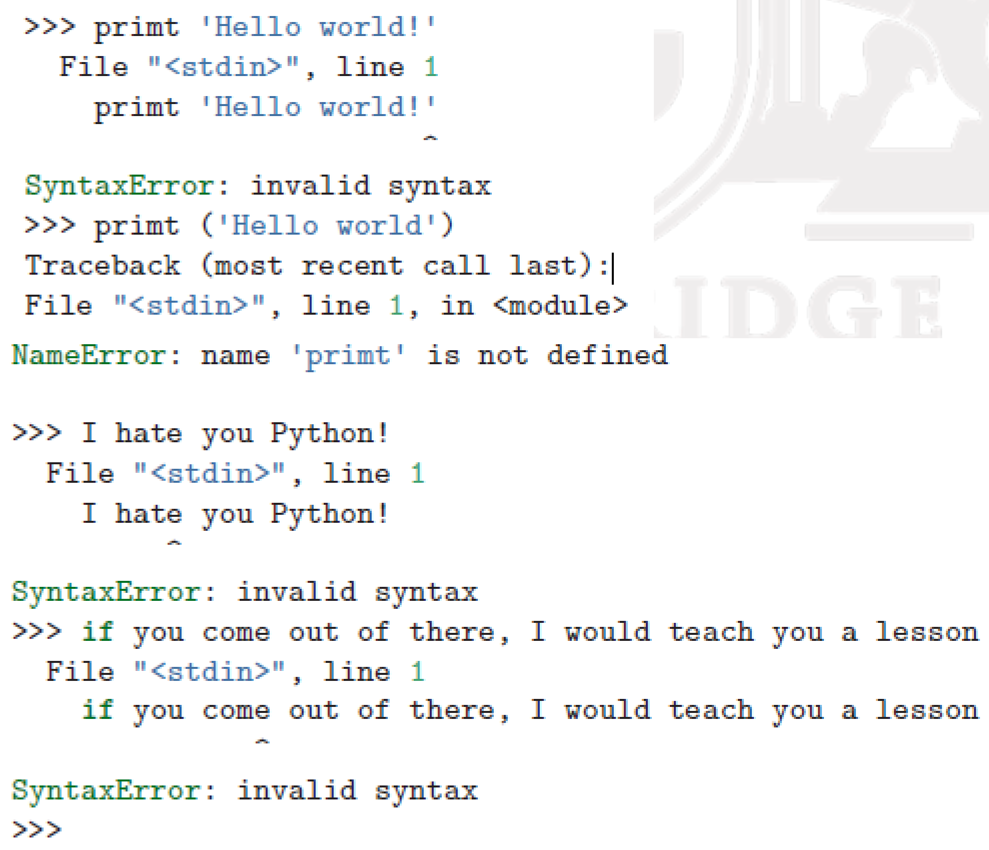
the clown ran after the car and the car ran into the tent
and the tent fell down on the clown and the car

```
name = input('Enter file:')
handle = open(name, 'r')
counts = dict()

for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

bigcount = None
bigword = None
for word, count in list(counts.items()):
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)
```

1.9 The building blocks of programs

- **input** Get data from the “outside world”. This might be reading data from a file, or even some kind of sensor like a microphone or GPS. In our initial programs, our input will come from the user typing data on the keyboard.
- **output** Display the results of the program on a screen or store them in a file or perhaps write them to a device like a speaker to play music or speak text.
- **sequential execution** Perform statements one after another in the order they are encountered in the script.
- **conditional execution** Check for certain conditions and then execute or skip a sequence of statements.
- **repeated execution** Perform some set of statements repeatedly, usually with some variation.
- **reuse** Write a set of instructions once and give them a name and then reuse those instructions as needed throughout your program.

1.10 What could possibly go wrong?



```
>>> primt 'Hello world!'
  File "<stdin>", line 1
    primt 'Hello world!'
                       ^
SyntaxError: invalid syntax
>>> primt ('Hello world')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'primt' is not defined

>>> I hate you Python!
  File "<stdin>", line 1
    I hate you Python!
         ^
SyntaxError: invalid syntax
>>> if you come out of there, I would teach you a lesson
  File "<stdin>", line 1
    if you come out of there, I would teach you a lesson
              ^
SyntaxError: invalid syntax
>>>
```

You will encounter three general types of errors:

Syntax errors

- These are the first errors you will make and the easiest to fix.
- A syntax error means that you have violated the "grammar" rules of Python.
- Python does its best to point right at the line and character where it noticed it was confused.
- The only tricky bit of syntax errors is that sometimes the mistake that needs fixing is actually earlier in the program than where Python *noticed* it was confused.
- So the line and character that Python indicates in a syntax error may just be a starting point for your investigation.

Logic errors

- A logic error is when your program has good syntax but there is a mistake in the order of the statements or perhaps a mistake in how the statements relate to one another.
- A good example of a logic error might be, "take a drink from your water bottle, put it in your backpack, walk to the library, and then put the top back on the bottle."

Semantic errors

- A semantic error is when your description of the steps to take is syntactically perfect and in the right order, but there is simply a mistake in the program.
- The program is perfectly correct but it does not do what you *intended* for it to do

Chapter 2
Variables,expressions, and statements

2.1 Values and types

- A *value* is one of the basic things a program works with, like a letter or a number.
- The values we have seen so far are 1, 2, and “Hello, World!”
- These values belong to different *types*: 2 is an integer, and “Hello, World!” is a *string*, so called because it contains a “string” of letters.
- The print statement also works for integers.
- We use the python command to start the interpreter.

```
python
>>> print(4)
4
```

- If you are not sure what type a value has, the interpreter can tell you.

```
>>> type('Hello, World!')
<class 'str'>
>>> type(17)
<class 'int'>
```

- Strings belong to the type str and integers belong to the type int.
- Less obviously, numbers with a decimal point belong to a type called float, because these numbers are represented in a format called *floating point*.

```
>>> type(3.2)
<class 'float'>

>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

- When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000.
- This is not a legal integer in Python, but it is legal:

```
>>> print(1,000,000)
1 0 0
```

- Python interprets 1,000,000 as a comma separated sequence of integers, which it prints with spaces between.
- This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

2.2 Variables

- A variable is a name that refers to a value.
- An *assignment statement* creates new variables and gives them values:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

- This example makes three assignments.
- The first assigns a string to a new variable named message;
- the second assigns the integer 17 to n;
- the third assigns the (approximate) value of _ to pi.

To display the value of a variable, you can use a print statement:

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

The type of a variable is the type of the value it refers to.

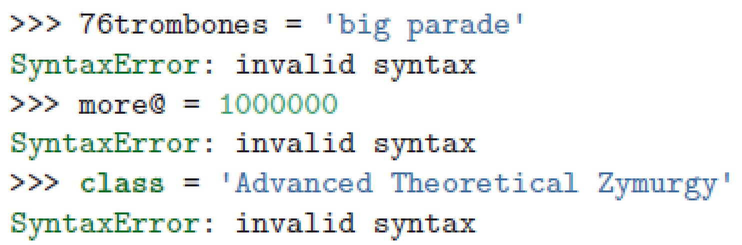
```
>>> type(message)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

2.3 Variable names and keywords

- Programmers generally choose names for their variables that are meaningful and document what the variable is used for.
- Variable names can be arbitrarily long.

- They can contain both letters and numbers, but they cannot start with a number.
- It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter.
- The underscore character (_) can appear in a name. It is often used in names with multiple words, such as my_name or airspeed_of_unladen_swallow.
- Variable names can start with an underscore character, but we generally avoid doing this unless we are writing library code for others to use.

If you give a variable an illegal name, you get a syntax error:



```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

- 76trombones is illegal because it begins with a number. more@ is illegal because it contains an illegal character, @.
- The class is one of Python's *keywords*. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

2.4 Statements

- A *statement* is a unit of code that the Python interpreter can execute.
- We have seen two kinds of statements:
  - print being an expression statement and assignment.
- When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one.
  - A script usually contains a sequence of statements.
- If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script

```
print(1)
x = 2
print(x)
```

produces the output

```
1
2
```

The assignment statement produces no output.

2.5 Operators and operands

- *Operators* are special symbols that represent computations like addition and multiplication.
- The values the operator is applied to are called *operands*.
- The operators +, -, *, /, and ** perform addition, subtraction, multiplication, division, and exponentiation, as in the following examples:

```
20+32    hour-1    hour*60+minute    minute/60    5**2        (5+9)*(15-7)
```

- There has been a change in the division operator between Python 2.x and Python 3.x.
- In Python 3.x, the result of this division is a floating point result:
```
>>> minute = 59
>>> minute/60
0.9833333333333333
```
- The division operator in Python 2.0 would divide two integers and truncate the result to an integer:
```
>>> minute = 59
>>> minute/60
0
```
- To obtain the same answer in Python 3.0 use floored (// integer) division.

```
>>> minute = 59
>>> minute//60
0
```

2.6 Expressions

- An *expression* is a combination of values, variables, and operators.
- A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions

```
17
x
x + 17
```

- If you type an expression in interactive mode, the interpreter *evaluates* it and displays the result:

```
>>> 1 + 1
2
```

2.7 Order of operations

- When more than one operator appears in an expression, the order of evaluation depends on the *rules of precedence*.
- For mathematical operators, Python follows mathematical convention.
- The acronym *PEMDAS* is a useful way to remember the rules:
- ***Parentheses***
    - have the highest precedence
    - and can be used to force an expression to evaluate in the order you want.
    - Since expressions in parentheses are evaluated first, 2 * (3-1) is 4, and (1+1)**(5-2) is 8.
    - You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even if it doesn't change the result.
- ***E*xponentiation** has the next highest precedence, so 2**1+1 is 3, not 4, and 3*1**3 is 3, not 27.
- ***M*ultiplication and *D*ivision** have the same precedence, which is higher than ***A*ddition and *S*ubtraction**, which also have the same precedence.
- So 2*3-1 is 5, not 4, and 6+4/2 is 8.0, not 5.

- Operators with the same precedence are evaluated from left to right.
- So the expression 5-3-1 is 1, not 3, because the 5-3 happens first and then 1 is subtracted from 2.

2.8 Modulus operator

- The *modulus operator* works on integers and yields the remainder when the first operand is divided by the second.
- In Python, the modulus operator is a percent sign (%).
- The syntax is the same as for other operators:

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

- So 7 divided by 3 is 2 with 1 left over.
- The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if x % y is zero, then x is divisible by y.
- You can also extract the right-most digit or digits from a number.
- For example, x % 10 yields the right-most digit of x (in base 10).
- Similarly, x % 100 yields the last two digits.

2.9 String operations

- The + operator works with strings, but it is not addition in the mathematical sense.
- Instead it performs *concatenation*, which means joining the strings by linking them end to end.
- For example:

```
>>> first = 10
>>> second = 15
>>> print(first+second)
25
>>> first = '100'
>>> second = '150'
>>> print(first + second)
100150
```

- The output of this program is 100150.

2.10 Asking the user for input

- Sometimes we would like to take the value for a variable from the user via their keyboard.
- Python provides a built-in function called input that gets input from the keyboard.
- When this function is called, the program stops and waits for the user to type something.
- When the user presses Return or Enter, the program resumes and input returns what the user typed as a string.
- In Python 2.0, this function was named raw_input.

```
>>> input = input()
Some silly stuff
>>> print(input)
Some silly stuff
```

- Before getting input from the user, it is a good idea to print a prompt telling the user what to input.
- You can pass a string to input to be displayed to the user before pausing for input:

```
>>> name = input('What is your name?\n')
What is your name?
Chuck
>>> print(name)
Chuck
```

- The sequence \n at the end of the prompt represents a *newline*, which is a special character that causes a line break.
- That's why the user's input appears below the prompt.
- If you expect the user to type an integer, you can try to convert the return value to int using the int() function:

```
>>> prompt = 'What...is the airspeed velocity of an unladen
      swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
>>> int(speed) + 5
22
```

- But if the user types something other than a string of digits, you get an error:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
```

2.11 Comments

- As programs get bigger and more complicated, they get more difficult to read.
- Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.
- For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing.
- These notes are called *comments*, and in Python they start with the # symbol:

```
# compute the percentage of the hour that has elapsed
        percentage = (minute * 100) / 60
```

- In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
percentage = (minute * 100) / 60 # percentage of an hour
```

- Everything from the \# to the end of the line is ignored; it has no effect on the program.
- Comments are most useful when they document non-obvious features of the code.
- It is reasonable to assume that the reader can figure out *what* the code does; it is much more useful to explain *why*.
- This comment is redundant with the code and useless:

```
v = 5 # assign 5 to v
```

- This comment contains useful information that is not in the code:

```
v = 5 # velocity in meters/second.
```

- Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a trade-off.

2.12 Choosing mnemonic variable names

- As long as you follow the simple rules of variable naming, and avoid reserved words, you have a lot of choice when you name your variables.
- In the beginning, this choice can be confusing both when you read a program and when you write your own programs.

- For example, the following three programs are identical in terms of what they accomplish, but very different when you read them and try to understand them.

```
a = 35.0
b = 12.50
c = a * b
print(c)

hours = 35.0
rate = 12.50
pay = hours * rate
print(pay)

x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

- The Python interpreter sees all three of these programs as *exactly the same* but humans see and understand these programs quite differently.
- Humans will most quickly understand the *intent* of the second program because the programmer has chosen variable names that reflect their intent regarding what data will be stored in each variable.
- We call these wisely chosen variable names "mnemonic variable names".
- The word *mnemonic* means "memory aid".
- We choose mnemonic variable names to help us remember why we created the variable in the first place.

2.13 Debugging

- At this point, the syntax error you are most likely to make is an illegal variable name, like class and yield, which are keywords, or odd~job and US$, which contain illegal characters.
- If you put a space in a variable name, Python thinks it is two operands without an operator:

```
>>> bad name = 5
SyntaxError: invalid syntax
>>> month = 09
File "<stdin>", line 1
month = 09
```

```
^
SyntaxError: invalid token
```

- For syntax errors, the error messages don’t help much. The most common messages are SyntaxError: invalid syntax and SyntaxError: invalid token, neither of which is very informative.
- The runtime error you are most likely to make is a “use before def;” that is, trying to use a variable before you have assigned a value.
- This can happen if you spell a variable name wrong:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

- Variables names are case sensitive, so LaTeX is not the same as latex.
- At this point, the most likely cause of a semantic error is the order of operations.
- For example, to evaluate 1/2_, you might be tempted to write

```
>>> 1.0 / 2.0 * pi
```

- But the division happens first, so you would get _/2, which is not the same thing!
- There is no way for Python to know what you meant to write, so in this case you don’t get an error message; you just get the wrong answer.

Chapter 3
Conditional execution

3.1 Boolean expressions

- A *boolean expression* is an expression that is either true or false. The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise:

```
>>> 5 == 5
True
>>> 5 == 6
False
{}
```

- True and False are special values that belong to the class bool; they are not strings:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

- The == operator is one of the *comparison operators*; the others are:

```
x != y        # x is not equal to y
x > y         # x is greater than y
x < y         # x is less than y
x >= y        # x is greater than or equal to y
x <= y        # x is less than or equal to y
x is y        # x is the same as y
x is not y    # x is not the same as y
```

- Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols for the same operations.
- A common error is to use a single equal sign (=) instead of a double equal sign (==).
- Remember that = is an assignment operator and == is a comparison operator.
- There is no such thing as =< or =>.

3.2 Logical operators

- There are three *logical operators*: and, or, and not.
- The semantics (meaning) of these operators is similar to their meaning in English.

- For example, x > 0 and x < 10 is true only if x is greater than 0 *and* less than 10.
- n%2 == 0 or n%3 == 0 is true if *either* of the conditions is true, that is, if the number is divisible by 2 *or* 3.
- Finally, the not operator negates a boolean expression, so not (x > y) is true if x > y is false; that is, if x is less than or equal to y.
- Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict.
- Any nonzero number is interpreted as "true."

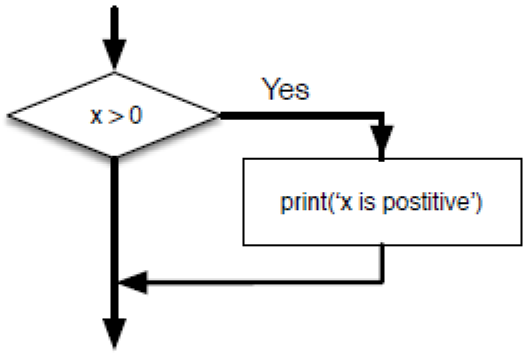
```
>>> 17 and True
True
```

3.3 Conditional execution

- In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly.
- *Conditional statements* give us this ability.
- The simplest form is the if statement:

```
if x > 0 :
print('x is positive')
```

- The boolean expression after the if statement is called the *condition*.
- We end the if statement with a colon character (:) and the line(s) after the if statement are indented.



- If the logical condition is true, then the indented statement gets executed. If the logical condition is false, the indented statement is skipped.
- if statements have the same structure as function definitions or for loops
- The statement consists of a header line that ends with the colon character (:) followed by an indented block.
- Statements like this are called *compound statements* because they stretch across more than one line.

- If you enter an if statement in the Python interpreter, the prompt will change from three chevrons to three dots to indicate you are in the middle of a block of statements, as shown below:

```
>>> x = 3
>>> if x < 10:
... print('Small')
...
Small
>>>
```

3.4 Alternative execution

- A second form of the if statement is *alternative execution*, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
if x%2 == 0 :
        print('x is even')
else :
        print('x is odd')
```

- If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect.
- If the condition is false, the second set of statements is executed.
- Since the condition must either be true or false, exactly one of the alternatives will be executed.
- The alternatives are called *branches*, because they are branches in the flow of execution.

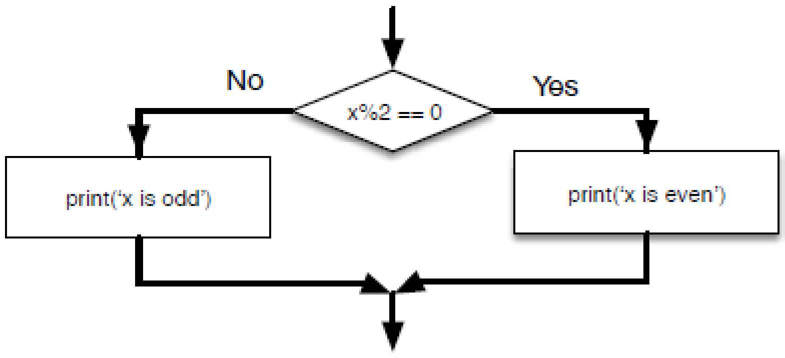


Figure 3.2: If-Then-Else Logic

3.5 Chained conditionals

- Sometimes there are more than two possibilities and we need more than two branches.
- One way to express a computation like that is a *chained conditional*:

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

- elif is an abbreviation of "else if." Again, exactly one branch will be executed.

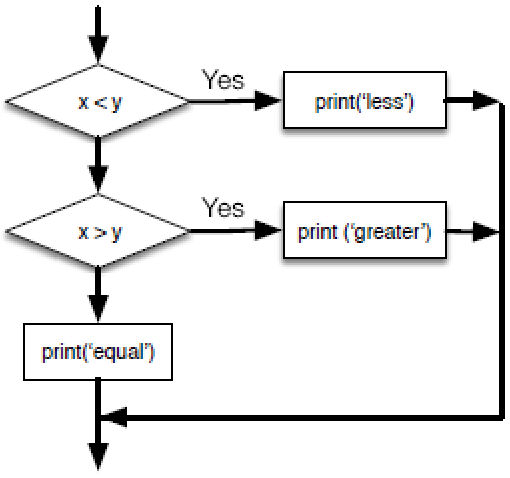


Figure 3.3: If-Then-ElseIf Logic

- There is no limit on the number of elif statements. If there is an else clause, it has to be at the end

```
if choice == 'a':
    print('Bad guess')
elif choice == 'b':
    print('Good guess')
elif choice == 'c':
    print('Close, but not correct')
```

- Each condition is checked in order. If the first is false, the next is checked, and so on.
- If one of them is true, the corresponding branch executes, and the statement ends.
- Even if more than one condition is true, only the first true branch executes.

3.6 Nested conditionals

- One conditional can also be nested within another. We could have written the three-branch example like this:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

- The outer conditional contains two branches. The first branch contains a simple statement.
- The second branch contains another if statement, which has two branches of its own.
- Those two branches are both simple statements, although they could have been conditional statements as well.

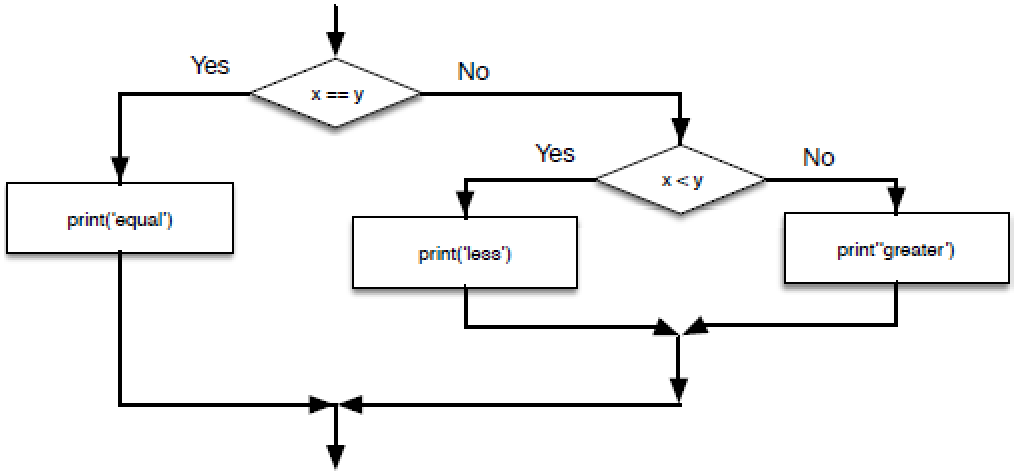


Figure 3.4: Nested If Statements

- Logical operators often provide a way to simplify nested conditional statements.
- For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
        if x < 10:
                print('x is a positive single-digit number.')
```

- The print statement is executed only if we make it past both conditionals, so we can get the same effect with the and operator:

```
if 0 < x and x < 10:
        print('x is a positive single-digit number.')
```

- The print statement is executed only if we make it past both conditionals, so we
can get the same effect with the and operator:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

3.7 Catching exceptions using try and except

- Here is a sample program to convert a Fahrenheit temperature to a Celsius temperature:

```
inp = input('Enter Fahrenheit Temperature: ')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)
```

- If we execute this code and give it invalid input, it simply fails with an unfriendly error message:

```
python fahren.py
Enter Fahrenheit Temperature:72
22.22222222222222

python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
File "fahren.py", line 2, in <module>
fahr = float(inp)
ValueError: could not convert string to float: 'fred'
```

- There is a conditional execution structure built into Python to handle these types of expected and unexpected errors called “try / except”.
- The idea of try and except is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs.
- These extra statements (the except block) are ignored if there is no error.

- We can rewrite our temperature converter as follows:

```
inp = input('Enter Fahrenheit Temperature:')
try:
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
```

```
print(cel)
except:
print('Please enter a number')
```

- Python starts by executing the sequence of statements in the try block. If all goes well, it skips the except block and proceeds.
- If an exception occurs in the try block, Python jumps out of the try block and executes the sequence of statements in the except block.

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.22222222222222

python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

- Handling an exception with a try statement is called *catching* an exception.
- In this example, the except clause prints an error message.
- In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

3.8 Short-circuit evaluation of logical expressions

- When Python is processing a logical expression such as x >= 2 and (x/y) > 2, it evaluates the expression from left to right.
- Because of the definition of and, if x is less than 2, the expression x >= 2 is False and so the whole expression is False regardless of whether (x/y) > 2 evaluates to True or False.
- When Python detects that there is nothing to be gained by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression.
- When the evaluation of a logical expression stops because the overall value is already known, it is called *short-circuiting* the evaluation.
- The short-circuit behavior leads to a clever technique called the *guardian pattern*.
- Consider the following code sequence in the Python interpreter:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

- ➢ The third calculation failed because Python was evaluating (x/y) and y was zero, which causes a runtime error.
- ➢ But the second example did *not* fail because the first part of the expression x >= 2 evaluated to False so the (x/y) was not ever executed due to the *short-circuit* rule and there was no error.
- ➢ We can construct the logical expression to strategically place a *guard* evaluation just before the evaluation that might cause an error as follows:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

- ➢ In the first logical expression, x >= 2 is False so the evaluation stops at the and.
- ➢ In the second logical expression, x >= 2 is True but y != 0 is False so we never reach (x/y).
- ➢ In the third logical expression, the y != 0 is *after* the (x/y) calculation so the expression fails with an error.
- ➢ In the second expression, we say that y != 0 acts as a *guard* to insure that we only execute (x/y) if y is non-zero.

Chapter 4
Functions

4.1 Function calls

- In the context of programming, a *function* is a named sequence of statements that performs a computation.
- When you define a function, you specify the name and the sequence of statements.
- Later, you can "call" the function by name.

```
>>> type(32)
<class 'int'>
```

- The name of the function is type.
- The expression in parentheses is called the *argument* of the function.
- The argument is a value or variable that we are passing into the function as input to the function.
- The result, for the type function, is the type of the argument.
- It is common to say that a function "takes" an argument and "returns" a result.
- The result is called the *return value*.

4.2 Built-in functions

- Python provides a number of important built-in functions that we can use without needing to provide the function definition.
- The creators of Python wrote a set of functions to solve common problems and included them in Python for us to use.
- The max and min functions give us the largest and smallest values in a list, respectively:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

- The max function tells us the "largest character" in the string (which turns out to be the letter "w") and the min function shows us the smallest character (which turns out to be a space).

- Another very common built-in function is the len function which tells us how many items are in its argument.
- If the argument to len is a string, it returns the number of characters in the string.

```
>>> len('Hello world')
11
>>>
```

- These functions are not limited to looking at strings. They can operate on any set of values
- You should treat the names of built-in functions as reserved words (i.e., avoid using “max” as a variable name).

4.3 Type conversion functions

- Python also provides built-in functions that convert values from one type to another.
- The int function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

- int can convert floating-point values to integers, but it doesn’t round off; it chops off the fraction part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

- float converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

- Finally, str converts its argument to a string:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

4.4 Random numbers

- Given the same inputs, most computer programs generate the same outputs every time, so they are said to be *deterministic*.
- For some applications, though, we want the computer to be unpredictable.
- Making a program truly nondeterministic turns out to be not so easy, but there are ways to make it at least seem nondeterministic.
- One of them is to use *al- gorithms* that generate *pseudorandom* numbers.
- Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.
- The random module provides functions that generate pseudorandom numbers
- The function random returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0).
- Each time you call random, you get the next number in a long series.

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

This program produces the following list of 10 random numbers between 0.0 and up to but not including 1.0.

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

- The random function is only one of many functions that handle random numbers.
- The function randint takes the parameters low and high, and returns an integer between low and high (including both).

```
>>> random.randint(5, 10)
5
```

```
>>> random.randint(5, 10)
9
```

- To choose an element from a sequence at random, you can use choice:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

- The random module also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.

4.5 Math functions

- Python has a math module that provides most of the familiar mathematical functions.
- Before we can use the module, we have to import it:

```
>>> print(math)
<module 'math' (built-in)>
```

- The module object contains the functions and variables defined in the module.
- To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period).
- This format is called *dot notation*.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)
>>> radians = 0.7
>>> height = math.sin(radians)
```

- The first example computes the logarithm base 10 of the signal-to-noise ratio.
- The math module also provides a function called log that computes logarithms base e.
- The second example finds the sine of radians.
- The name of the variable is a hint that sin and the other trigonometric functions (cos, tan, etc.) take arguments in radians.
- To convert from degrees to radians, divide by 360 and multiply by 2_:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865476
```

- The expression math.pi gets the variable pi from the math module. The value of this variable is an approximation of _, accurate to about 15 digits.
- you can check the previous result by comparing it to the square root of two divided by two:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

4.6 Adding new functions

- So far, we have only been using the functions that come with Python, but it is also possible to add new functions.
- A *function definition* specifies the name of a new function and the sequence of statements that execute when the function is called.
- Once we define a function, we can reuse the function over and over throughout our program.
  Here is an example:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')
```

- def is a keyword that indicates that this is a function definition.
- The name of the function is print_lyrics.
- The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but the first character can't be a number.
- You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.
- The empty parentheses after the name indicate that this function doesn't take any arguments.
- Later we will build functions that take arguments as their inputs.
- The first line of the function definition is called the *header*; the rest is called the *body*.
- The header has to end with a colon and the body has to be indented.
- By convention, the indentation is always four spaces.
- The body can contain any number of statements.
- The strings in the print statements are enclosed in quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.
- If you type a function definition in interactive mode, the interpreter prints ellipses (. . .) to let you know that the definition isn't complete:

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay.")
...     print('I sleep all night and I work all day.')
...
```

- To end the function, you have to enter an empty line
- Defining a function creates a variable with the same name.

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> print(type(print_lyrics))
<class 'function'>
```

- The value of print_lyrics is a *function object*, which has type "function".
- The syntax for calling the new function is the same as for built-in functions:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

- Once you have defined a function, you can use it inside another function.
- For example, to repeat the previous refrain, we could write a function called repeat_lyrics:

```
def repeat_lyrics():
print_lyrics()
print_lyrics()
```

And then call repeat_lyrics:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

But that's not really how the song goes.

4.7 Definitions and uses

Pulling together the code fragments from the previous section, the whole program looks like this:

```
def print_lyrics():
print("I'm a lumberjack, and I'm okay.")
print('I sleep all night and I work all day.')
def repeat_lyrics():
print_lyrics()
print_lyrics()
```

4.8. FLOW OF EXECUTION 49

repeat_lyrics()
# Code: http://www.py4e.com/code3/lyrics.py
This program contains two function definitions: print_lyrics and repeat_lyrics.
Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not get executed until the function is called, and the function definition generates no output.
As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.
Exercise 2: Move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.
Exercise 3: Move the function call back to the bottom and move the definition of print_lyrics after the definition of repeat_lyrics. What happens when you run this program?

4.8 Flow of execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the *flow of execution*.
Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.
Function *definitions* do not alter the flow of execution of the program, but remember
that statements inside the function are not executed until the function is called.
A function call is like a detour in the flow of execution. Instead of going to the next
statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.
That sounds simple enough, until you remember that one function can call another.
While in the middle of one function, the program might have to execute the statements
in another function. But while executing that new function, the program might have to execute yet another function!
Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it.
When it gets to the end of the program, it terminates.
What's the moral of this sordid tale? When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

4.9 Parameters and arguments

Some of the built-in functions we have seen require arguments. For example, when you call math.sin you pass a number as an argument. Some functions take more than one argument: math.pow takes two, the base and the exponent.
Inside the function, the arguments are assigned to variables called *parameters*.
Here is an example of a user-defined function that takes an argument:

```
def print_twice(bruce):
print(bruce)
print(bruce)
```

This function assigns the argument to a parameter named bruce. When the function is called, it prints the value of the parameter (whatever it is) twice.
This function works with any value that can be printed.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> import math
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
```

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for print_twice:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

The argument is evaluated before the function is called, so in the examples the expressions “Spam ’*4andmath.cos(math.pi)‘ are only evaluated once.
You can also use a variable as an argument:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

The name of the variable we pass as an argument (michael) has nothing to do with the name of the parameter (bruce). It doesn’t matter what the value was called back home (in the caller); here in print_twice, we call everybody bruce.

4.10 Fruitful functions and void functions

Some of the functions we are using, such as the math functions, yield results; for lack of a better name, I call them *fruitful functions*. Other functions, like print_twice, perform an action but don't return a value. They are called *void functions*.

When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

When you call a function in interactive mode, Python displays the result:

```
>>> math.sqrt(5)
2.23606797749979
```

But in a script, if you call a fruitful function and do not store the result of the function in a variable, the return value vanishes into the mist!

```
math.sqrt(5)
```

This script computes the square root of 5, but since it doesn't store the result in a variable or display the result, it is not very useful.

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result to a variable, you get a special value called None.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

The value None is not the same as the string "None". It is a special value that has its own type:

```
>>> print(type(None))
<class 'NoneType'>
```

To return a result from a function, we use the return statement in our function. For example, we could make a very simple function called addtwo that adds two numbers together and returns a result.

```
def addtwo(a, b):
added = a + b
return added
x = addtwo(3, 5)
```

```
print(x)
# Code: http://www.py4e.com/code3/addtwo.py
```

When this script executes, the print statement will print out “8” because the addtwo function was called with 3 and 5 as arguments. Within the function, the parameters a and b were 3 and 5 respectively. The function computed the sum of the two numbers and placed it in the local function variable named added. Then it used the return statement to send the computed value back to the calling code as the function result, which was assigned to the variable x and printed out.

4.11 Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

• Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.

• Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.

• Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.

• Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Throughout the rest of the book, often we will use a function definition to explain a concept. Part of the skill of creating and using functions is to have a function properly capture an idea such as “find the smallest value in a list of values”. Later we will show you code that finds the smallest in a list of values and we will present it to you as a function named min which takes a list of values as its argument and returns the smallest value in the list.