LEARNING TO PROGRAM WITH PYTHON

Richard L. Halterman

Preface

Legal Notices and Information

Chapter 1

1.2. DEVELOPMENT TOOLS

1.4. WRITING A PYTHON PROGRAM

Figure 1.4: Launching the IDLE editor

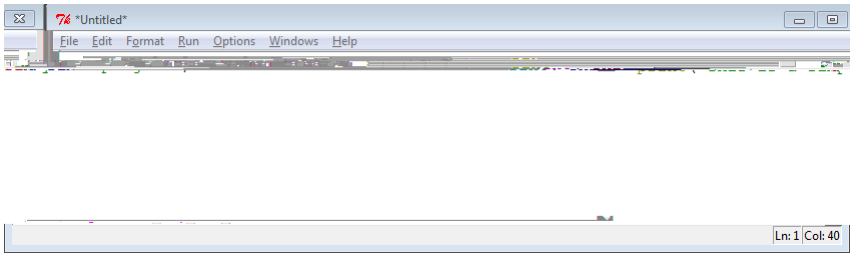


Figure 1.5: The simple Python program typed into the IDLE editor

Figure 1.6: Saving a file created with the IDLE editor

Listing 1.1 (`simple.py`) contains only one line of code:

```
print("This is a simple Python program
```

2. What is an interpreter?
3. How is a compiler similar to an interpreter? How are they different?
4. How is compiled or interpreted code different from source code?

The `type` function can determine the type of the most complicated expressions:

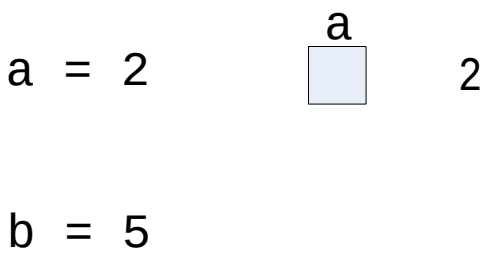


Figure 2.2: How variable bindings change as a program runs

program. Good variable names make programs more readable by humans. Since programs often contain many variables, well-chosen variable names can render an otherwise obscure collection of symbols more understandable.

Python has strict rules for variable names. A variable name is one example of an *identifier*. An identifier

and	del	from	None	try
as	elif	global	nonlocal	True
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield

2.6 User Input

The `print` function enables a Python program to display textual information to the user. Programs may use the `input` function to obtain information from the user. The simplest use of the `input` function assigns a string to a variable:

```
print('
```

-

Chapter 3

Expressions and Arithmetic

This chapter uses the Python numeric types introduced in Chapter 2 to build expressions and perform arithmetic. Some other important concepts are covered—user input, comments, and dealing with errors.

3.1 Expressions

A literal value like 34 and a variable like x are examples of a simple *expressions*. Values and variables can


Expression

```
x, y, z = 3, -4, 0
x = -x
y = -y
z = -z
print(x, y, z)
```

within a program would print

```
-3 4 0
```

The following statement

```
print(-(4 - 5))
```

within a program would print

```
1
```

The unary + operator is present only for completeness; when applied to a numeric value, variable, or

```
print(10/3, 3/10, 10//3, 3//10)
```

prints

```
3.3333333333333335 0.3 3 0
```

3.6 More Arithmetic Operators

3.8. SUMMARY

- A binary operator performs an operation using two operands.
-

```
x1 = 2
x2 = 2
x1 += 1
x2 -= 1
print(x1)
print(x2)
```

Why does the output appear as it does?

18. Consider the following program that attempts to compute the circumference of a circle given the radius entered by the user. Given a circle's radius, r, the circle's circumference, C is given by the formula:

$$C = 2$$

Chapter 4

Conditional Execution

All the programs in the preceding chapters execute exactly the same statements regardless of the input (if any) provided to them. They follow a linear sequence: *Statement* 1, *Statement* 2, etc. until the last statement is executed and the program terminates. Linear programs like these are very limited in the problems they can solve. This chapter introduces constructs that allow program statements to be optionally executed, depending on the context of the program's execution.

4.1 Boolean Expressions

Arithmetic expressions evaluate to numeric values; a *Boolean* expression, sometimes called a *predicate*,

The relational operators are binary operators and are all left associative. They all have a lower precedence than any of the arithmetic operators; therefore, the expression

```
x + 2 < y / 10
```

Python requires the block to be indented. If the block contains just one statement, some programmers will place it on the same line as the `if`; for example, the following `if` statement that optionally assigns `y`

```
if x < 10:
    y = x
```

could be written

```
if  x < 10: y = x
```

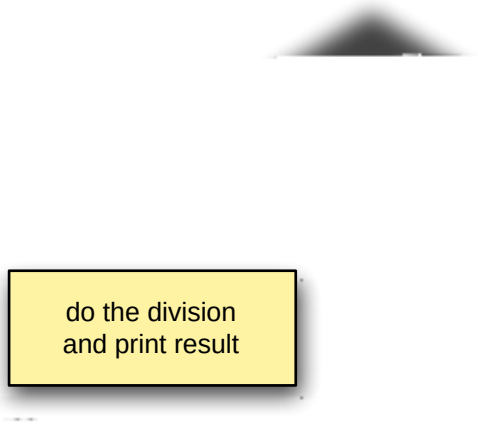
but may *not* be written as

```
if x < 10:
y = x
```

because the lack of indentation hides the fact that the assignment statement is optionally executed. Indentation is how Python determines which statements make up a block.

It is important not to mix spaces and tabs when indenting statements in a block. In many editors you cannot visually distinguish between a tab and a sequence of spaces. The number of spaces equivalent to the spacing of a tab differs from one editor to another. Most programming editors have a setting to substitute a specified number of spaces for each tab character. For Python development you should use this feature. It is best to eliminate all tabs within your Python source code.

```
if 1:
    print('one'
```


do the division
and print result

In mathematics, we expect the following equality to hold:

$$1.11 = 1.10 = 0.01 = 2.11 - 2.10$$

The output of the first `print` statement in Listing 4.6 (`samedifferent.py`) reminds us of the imprecision of floating-point numbers:

Convince yourself that the following expressions are equivalent:

```
x != y
```

The expression

The condition the `if` within Listing 4.8 (`newcheckrange.py`):

```
value >= 0 and value <= 10
```

and be expressed more compactly as

```
0 <= value <= 10
```

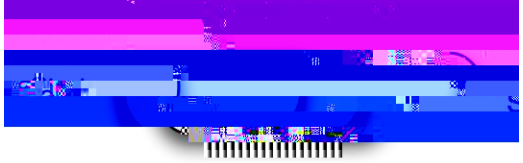
3.

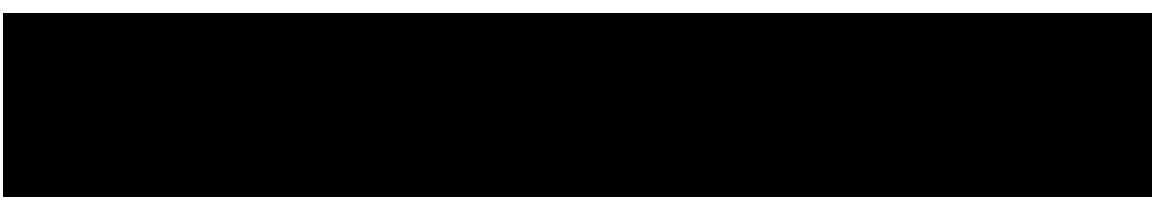
Some argue that the conditional expression is not as readable as a normal `if/else` statement. Regard-

-

evaluate the following Boolean expressions:

(b)





5.4 Nested Loops

Just like with `if` statements, `while`

```
5     #   The troubleshooting control logic
```

```
while condition_1 :
        statement_1
        statement_2
        ...
        statement_n
        if condition_2 :
                statement_n+1
                statement_n+2
                 ...
                statement_n+m
                continue
        statement_n+m+1
        statement_n+m+2
        ...
        statement_n+m+p
```

can be rewritten as

```
while condition_1 :
        statement_1
        statement_2
        ...
        statement_n
        if condition_2 :
                statement_n+1
                statement_n+2
                ...
                statement_n+m
        else:
                statement_n+m+1
                statement_n+m+2
                ...
                statement_n+m+p
```

The logic of the `else` version is no more complex than the `continue` version. Therefore, unlike the `break` statement above, there is no compelling reason to use the `continue` statement. Sometimes a

```
    while factor <= n:
        if n % factor == 0:
            print(factor, end='
```

Listing 5.19: `startree.py`

```
#  Get tree height from user
```

The two inner loops play distinct roles:

- The first inner loop prints spaces. The number of spaces printed is equal to the height of the tree the first time through the outer loop and decreases each iteration. This is the correct behavior since each

Some important questions can be asked.

1.

```
            print('*', end='')
        a += 1
    print()
```

9. How many asterisks does the following code fragment print?

```
a = 0
```

```
***
***
***
```

num is the information the function needs to do its work. We say num is the

```
print(sqrt("16"))  # Illegal,
```

Like mathematical functions that must produce a result, a Python function always produces a value to return to the client. Some functions are not designed to produce any useful results. Clients call such a function for the effects provided by the executing codr4(a)-24withi fa(a)-24unction for tny falue thea fhefcompute. Tprin

fwindo25(aw;-2939(it-2933ndos)-2834not)-3233ncompute-3233nand-2834neturnalue Sincfa-6306nalu1print]TJ/F209.96

The parameter passed by the client is known as the *actual* parameter. The parameter specified by the function is called the *formal* parameter. During a function call the first actual parameter is assigned to the first formal parameter, the second actual parameter is assigned to the second formal parameter, etc. Callers must be careful to put the arguments they pass in the proper order when calling a function. The call `pow(10,2)` computes $10^2 = 100$, but the call `pow(2,10)` computes $2^{10} = 1,024$.

A Python program that uses any of these mathematical functions must import the `math` module.

Figure 6.2 shows a problem that can be solved using functions found in the

Listing 6.3 (`orbitdist.py`) uses these mathematical results to compute the difference in the distances.

- When faced with the choice of using a standard library function or writing your own code to solve the same problem, choose the library function. The standard function will be tested thoroughly, well documented, and likely more efficient than the code you would write.

- The function is a standard unit of reuse in Python.

- Code that uses a function is known as *client* code.

-

6.7 Exercises

1. Suppose you need to compute the square root of a number in a Python program. Would it be a good idea to write the code to perform the square root calculation? Why or why not?
2. Which of the following values could be produced by the call `random.randrange(0, 100)` function (circle all that apply)?

Chapter 7

Writing Functions

```
value1 = prompt(1)
```

The parameter passed in, 1, is the actual parameter. An *actual* parameter is the parameter *actually*

```
12 #
```

given number is prime; `main` simply delegates the work to `is_prime` and makes use of the `is_prime` function's findings. For `is_prime`

```
7
```

```
2 #    Forces the user
```

```
i = 0
i = 0.1
i = 0.2
i = 0.3
i = 0.4
i = 0.5
i = 0.6
i = 0.7
i = 0.8
i = 0.9
i = 1
i = 1.1
i = 1.2
i = 1.3
i = 1.4
i = 1.5
i = 1.6
i = 1.7
i = 1.8
i = 1.9
i = 2
i = 2.1
```

We expect it stop when the loop variable `i` equals 1, but the program continues executing until the user types **Ctrl-C** or otherwise interrupts the program's execution. We are adding 0.1, just as in Listing 7.14 (`simplefloataddition.py`

```
def
```

```
def main():
    proc(5)

main()
```

9. The programmer was expecting the following program to print 200. What does it print instead? Why does it print what it does?

```
def proc(x):
    x = 2*x*x

def main():
    num = 10
```

14. What happens if a client passes too few parameters to a function?

15. What are the rules for naming a function in Python?

16. Consider the following function definitions:

```
def fun1(n):
    result = 0
    while n:
        result += n
        n--
    return result

def fun2(stars):
    for i in range(stars + 1):
        print(end="*")
    print()
```

```
5 def help_screen():
6     print("Add:  Adds two numbers
```

8.2 Default Parameters

We have seen how clients may call some Python functions with differing numbers of parameters. Compare

```
a = input()
```

to

```
a = input("Enter your name: ")
```

We can define our own functions that accept a varying number of parameters by using a technique known as *default parameters*

1. The function must optionally call itself within its definition; this is the *recursive case*.
2. The function must optionally *not* call itself within its definition; this is the *base case*.
3. Some sort of conditional execution (such as an `if/else` statement) selects between the recursive case and the base case based on one or more parameters passed to the function.
4.

will use only local variables and parameters. Such a function is a truly independent function can be reused easily in multiple programs.

Listing 8.8: `docprime.py`

We will see in Section 10.2 that the ability to pass function objects around enables us to develop flexible

- the ability to dispense with creating individual variables to store all the individual values

These may seem like contradictory requirements, but Python provides a standard data structure that simultaneously provides both of these advantages—the list.

9.1 Using Lists

A list refers to a collection of objects; it represents an ordered sequence of data. In that sense, a list is similar to a string, except a string can hold only characters. We may access the elements contained in a list via their position within the list. A list need not be homogeneous; that is, the elements of a list do not all have to be of the same type.

Like any other variable, a list variable can be local or global, and it must be defined (assigned) before it is used. The following code fragment declares a list named `lst` that holds the integer values $2, -3, 0, 4, -1$:

```
lst = [2, -3, 0, 4, -1]
```

The right-hand side of the assignment statement is a literal list. The elements of the list appear within square brackets ([ ]

We clearly see that a single list can hold integers, floating-point numbers, strings, and even functions. A list

```
>>> a = [2, 4, 6, 8]
>>> a
[2, 4, 6, 8]
>>> a + [1, 3, 5]
[2, 4, 6, 8, 1, 3, 5]
>>> a
```

9.2. LIST ASSIGNMENT AND EQUIVALENCE

9.2. LIST ASSIGNMENT AND EQUIVALENCE

If a refers to a list, the statement

```
b = a
```

```
6
7 def main():
8     #  a and b are distinct lists that contain the same elements
```

Since the index may consist of an arbitrary integer expression whose value cannot be determined until run time, the interpreter checks every attempt to access a list. If the interpreter detects an out-of-bounds index, the interpreter raises an `IndexError` (list index out-of-bounds) exception. The programmer must ensure the provided index is in bounds to prevent such a run-time error.

The above unreliable code can be helped with conditional access:

```
#  Make a list containing
```

tents of the list. This is known as *slice assignment*. A slice assignment can modify a list by removing or adding a subrange of elements in an existing list. Listing 9.17 (`listslicemod.py`) demonstrates how slice assignment can be used to modify a list.

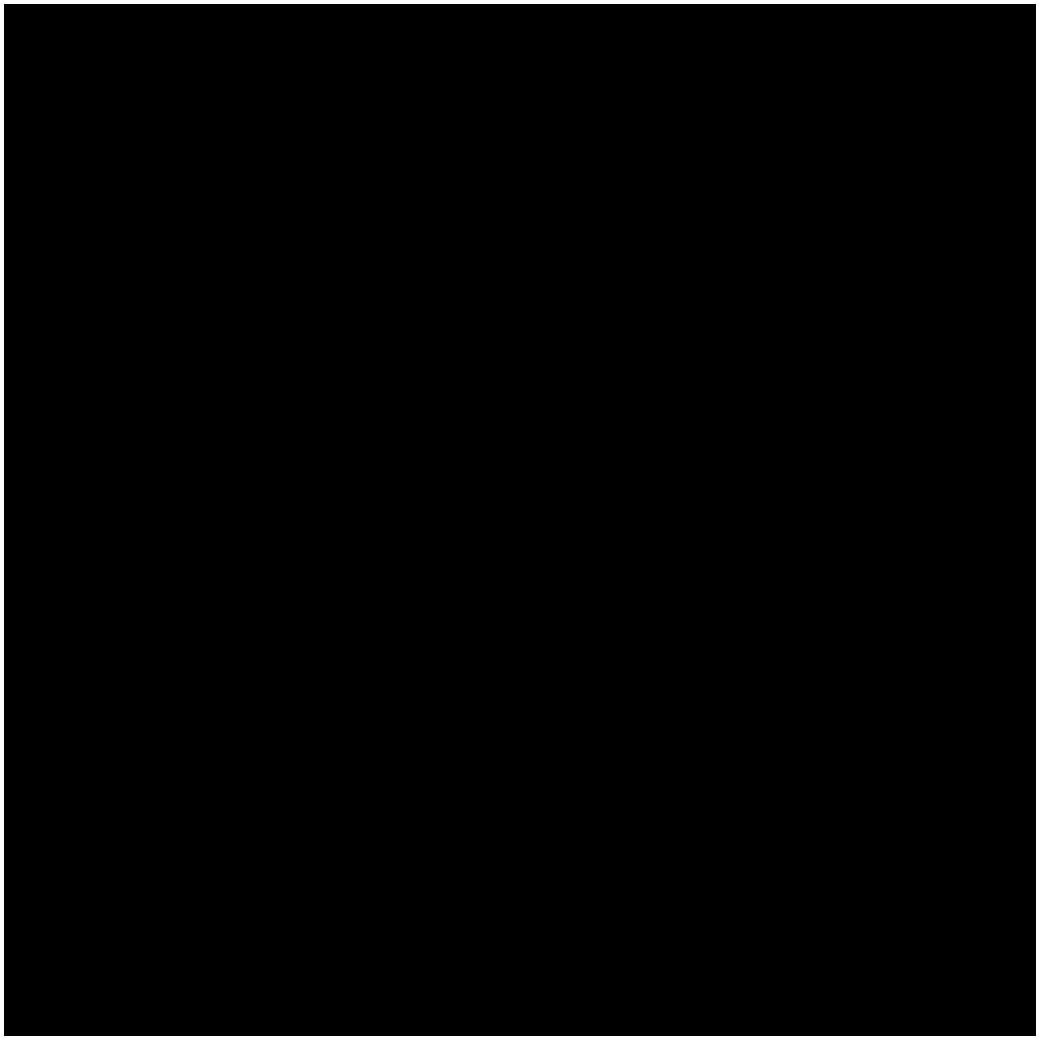
```
    lst is the
```

- A list subscript must evaluate to an integer. Integer literals, variables, and expressions can be used as

Chapter 10

```
    # Examine
```

```
    The contents of lst are physically rearranged.
    '''
    n = len(lst)
    for i in range(n - 1):
        #  Note: i, small,
```



simply change the line

```
            if lst[j] < lst[small]:
```

to be

```
            if lst[j] > lst[small]:
```

What if instead we want to change the sort so that it sorts the elements in ascending order except that all the even numbers in the list appear before all the odd numbers? This modification would be a little more complicated, but it could accomplished in that `if`

10.3.1 Linear Search

Listing 10.3 (`linearsearch.py`) uses a function named `locate` that returns the position of the first occurrence of a given element in a list; if the element is not present, the function returns `None`.

Listing 10.3: `linearsearch.py`

```
1 def locate(lst, seek):
2     '''
3     Returns the index
```

The client code, in this example the `display` function, must ensure that `locate`'s result is not `None` before attempting to use the result as an index into a list.

The kind of search performed by `locate` is known as *linear search*, since a straight line path is taken from the beginning of the list to the end of the list considering each element in order. Figure 10.1 illustrates linear search.

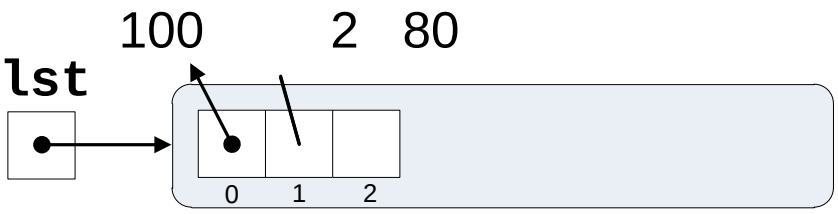


Figure 10.1: Linear search

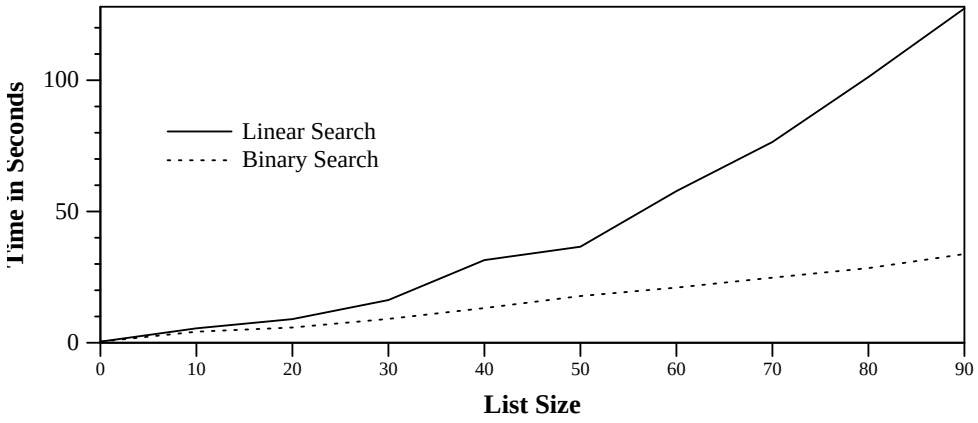
10.3.2 Binary Search

Linear search is acceptable for relatively small lists, but the process of examining each element in a large list is time consuming. An alternative to linear search is *binary search*. In order to perform binary search, a list must be in sorted order. Binary search exploits the sorted structure of the list using a clever but simple strategy that quickly zeros in on the element to find:

1. If the list is empty, return `None`.

```
53      else:
54          print("(", value, " not in list)", sep='')
55      print()
56
57
58  def main():
59      a = [2, 5, 11, 13, 44, 80, 100, 110]
```

5230Tdl(print())]TJE592.1-5355.0112603.3699cm0.970.970.97rg0.970.970.97RG10010-3.2877cm002.988810.9589ref10012.98883.2877cm0g0G1001-3.38730cm0.970.970.97rg0.970.970.


Time in Seconds
100
50
0
Linear Search
Binary Search
0
10
20
30
40
50
60
70
80
90
List Size

```
    suffix_size = len(suffix)
    if suffix_size == 0:
```

```
20      print('After :', a)
21
22  main()
```

```
74      report(permutation_tally)   #  Report results
75
76
77
```

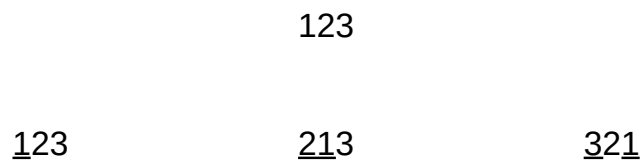


Figure 10.5: A tree mapping out the ways in which `faulty_permute` can transform the list [1, 2, 3] at each iteration of its `for` loop

As Figure 10.5 shows, the lists `[1, 2, 3]`, `[2, 1, 3`, and `[2, 3, 1]` each appear twice in the last row, while `[1, 3, 2]`, `[3, 1, 2]`, and `[3, 2, 1]` each appear only once. This means, for example, that the function is twice as likely to produce `[1, 2, 3]` as `[1, 3, 2]`.

Figure 10.6: A tree mapping out the ways in which `permute` can transform the list [1, 2, 3] at each iteration of its `for` loop

Compare Figure 10.5 to Figure 10.6. The second row of the tree for `permute` is identical to the second row of the tree for `faulty_permute`

10.6 Reversing a List

Listing 10.10 (`listreverse.py`) contains a recursive function named `rev` that accepts a list as a parameter and returns a new list with all the elements of the original list in reverse order.

10. How many different orderings are there for the list

Chapter 11

Objects

In the hardware arena, a personal computer is built by assembling

- a motherboard (a circuit board containing sockets for a microprocessor and assorted support chips),
- a processor and its various support chips,
- memory boards,

Except for the object prefix, a method works just like a function. The

`str` Methods
`upper` Returns a copy of the original string with all the characters converted to uppercase
`lower` Returns a copy of the original string with all the characters converted to lower case
`rjust` Returns a string right justified within a field padded with a specified character which defaults to a space

```
[     ABCDEFGHBCDIJKLMNOPQRSBCDTUVWXYZ      ]
[ABCDEFGHBCDIJKLMNOPQRSBCDTUVWXYZ]
3
```

```
s = "
```

maps to

```
list.__setitem__(lst, 2, x)
```

11.5 Exercises

1. Add exercises

Chapter 12

Custom Types

```
point = [2.5, 6]
print("In", point, "the x coordinate is", point[0])
```

or as a tuple:

```
point = 2.5, 6
print("In", point, "the x coordinate is", point[0])
```

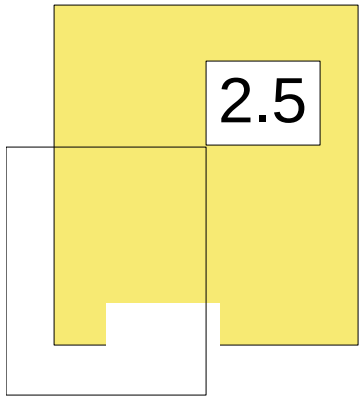


Figure 12.1: A Point object

A component data element of an object is called a *field*. Our `Point` objects have two fields, `x` and `y`. The terms *instance variable* or *attribute* sometimes are used in place of field. As with methods, Python uses the dot (`.`) notation to access a field of an object; thus,

```
pt.x = 0
```

12.2. METHODS

Clients may appear to change a protected field as

```
fract = Rational(1, 2)
fract.__denominator = 0         #  Legal, but what does it do?
```

```
        self.__account_number = number  #  Account number
        self.__ssn = ssn                #  Social security number
        self.__name = name              #  Customer name
        self.__balance = balance        #  Funds
```

12.3. CUSTOM TYPE EXAMPLES

```
3  class Stopwatch:
4      def __init__(self):
5          self.reset()
6
7      def start(self):  #  Start the timer
8          if not self.__running:
9              self.__start_time = clock()
10             self.__running = True  #  Clock now running
11         else:
12             print("Stopwatch already running"
```

```
        if lst[mid] == seek:
```

Notice that the `sort`

defines a new class named `CountingStopwatch`, but this new class is based on the existing class `Stopwatch`. This single line means that the `CountingStopwatch` class *inherits* everything from the `Stopwatch` class. `CountingStopwatch` objects automatically will have `start`, `stop`, `reset`, and `elapsed` methods.

We say `stopwatch` is the *superclass* of `CountingStopwatch`. Another term for superclass is *base class*. `CountingStopwatch` is the *subclass* of `Stopwatch`, or, said another way, `CountingStopwatch` is a *derived class* of `Stopwatch`.

Even though a subclass inherits all the fields and methods of its superclass, a subclass may add new fields, methods, and also provide new code for an inherited method. The statement

5570T7n the e

Chapter 13

Handling Exceptions

In our programming experience so far we have encountered several kinds of run-time errors, such as integer division by zero, accessing a list with an out-of-range index, using an object reference set to `None`

```
16          print("
```

13.4. CUSTOM EXCEPTIONS