Chapter 5
Conditionals and Recursion

A. Downey, *Think Python: How to Think Like a Computer Science,* 3rd ed., O'Reilly, 2024.

https://allendowney.github.io/ThinkPython/

Contents

Contents

1. Integer Division and Modulus

- Conventional division: /
  - Divides two numbers and returns a floating-point number

```
minutes = 105
minutes / 60
```

```
1.75
```

- Integer division: // (floor division)
  - Divides two numbers and rounds down to an integer

```
minutes = 105
hours = minutes // 60
hours
```

```
1
```

1. Integer Division and Modulus

- Modulus operator: %
  - Divides two numbers and returns the remainder

```
minutes = 105
remainder = minutes % 60
remainder
```

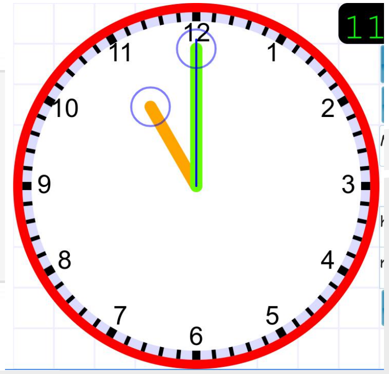
```
45
```

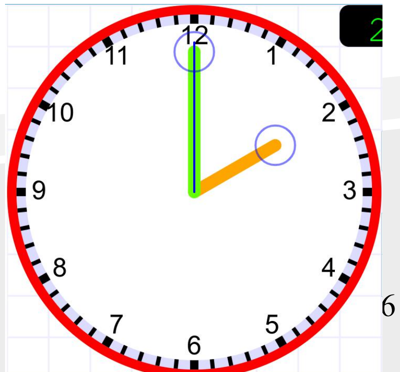
1. Integer Division and Modulus

- The modulus operator is more useful than it might seem.
  - It can check whether one number is divisible by another:
    - if x % y is zero, then x is divisible by y.
  - It can extract the rightmost digit or digits from a number.
    - For example, x % 10 yields the rightmost digit of x (in base 10). Similarly, x % 100 yields the last two digits.
  - Finally, the modulus operator can do “clock arithmetic.”
    - For example, if an event starts at 11 A.M. and lasts three hours, we can use the modulus operator to figure out what time it ends:

```
start = 11
duration = 3
end = (start + duration) % 12
end
```

```
2
```





2. Boolean Expressions

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

```
1 5 == 5
```

```
True
```

```
1 5 == 7
```

```
False
```

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

```
1 type(True)
```

```
bool
```

```
1 type(False)
```

```
bool
```

2. Boolean Expressions

- Relational Operators
  - Used to comare the operand values on either side

```
x = 5
y = 7
```

```
x == y          # x is equal to y
```

False

```
x != y          # x is not equal to y
```

True

```
x > y          # x is greater than y
```

False

```
x < y          # x is less than y
```

True

```
x >= y          # x is greater than or equal to y
```

False

```
x <= y          # x is lean than or equal to y
```

True

3. Logical Operators

- To combine boolean values into expressions, we can use **logical operators**.
- The most common are **and**, **or**, and **not**. The meaning of these operators is similar to their meaning in English.

```
x = 5
y = 7
```

```
x > 0 and x < 10
```

```
True
```

```
x % 2 == 0 or x % 3 == 0
```

```
False
```

```
not x > y
```

```
True
```

3. Logical Operators

- Strictly speaking, the operands of a logical operator should be boolean expressions, but Python is not very strict.
- Any nonzero number is interpreted as True:

```
42 and True
```

```
True
```

```
'Hello' and True
```

```
True
```

- This flexibility can be useful, but there are some subtleties to it that can be confusing. You might want to avoid it.

4. **if** Statements

- 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:

```python
x = 5
if x > 0:
    print('x is positive')
```

```
x is positive
```

  - **if** is a Python keyword. **if** statements have the structure: a **header** followed by an indented statement or sequence of statements called a **block**.
  - The boolean expression after **if** is called the **condition**. If it is true, the statements in the indented block run. If not, they don't.
- There is no limit to the number of statements that can appear in the block, but there has to be at least one.

4. **if** Statements

- Occasionally, it is useful to have a block that does nothing—usually as a place keeper for code you haven't written yet. In that case, you can use the **pass** statement, which does nothing:

```
if x < 0:
    pass    # TODO: need to handle negative values!
```

  - The word **TODO** in a comment is a conventional reminder that there's something you need to do later.

5. The **else** Clause

- An **if** statement can have a second part, called an **else** clause. The syntax looks like this:

```python
x = 5

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

```
x is odd
```

  - If the condition is true, the first indented statement runs; otherwise, the second indented statement runs.
  - Since the condition must be true or false, exactly one of the alternatives will run. The alternatives are called **branches**.

6. 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, which includes an **elif** clause:

```
x = 5
y = 7

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')
```

```
x is less than y
```

- **elif** is an abbreviation of "else if". There is no limit on the number of **elif** clauses. If there is an **else** clause, it has to be at the end, but there doesn't have to be one.
- 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 runs and the **if** statement ends. Even if more than one condition is true, only the first true branch runs.

7. Nested Conditionals

- One conditional can also be nested within another.

```
x = 5
y = 7

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')

```

```
x is less than y
```

  - The outer **if** statement 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.
  - Although the indentation of the statements makes the structure apparent, nested conditionals can be difficult to read. I suggest you avoid them when you can.

7. Nested Conditionals

- Logical operators often provide a way to simplify nested conditional statements.

```
x = 5
y = 7

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

```
x is a positive single-digit number.
```

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

```
x is a positive single-digit number.
```

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

```
x is a positive single-digit number.
```

8. Recursion

- It is legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. Here's an example:

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

```
countdown(3)
```

```
3
2
1
Blastoff!
```

- If n is 0 or negative, countdown outputs the word, “Blastoff!”. Otherwise, it outputs n and then calls itself, passing n - 1 as an argument.
- A function that calls itself is **recursive**.

8. Recursion

- The execution of **countdown** begins with n = 3, and since n is greater than 0, it displays 3, and then calls itself...
  - The execution of **countdown** begins with n = 2, and since n is greater than 0, it displays 2, and then calls itself...
    - The execution of **countdown** begins with n = 1, and since n is greater than 0, it displays 1, and then calls itself...
      - The execution of **countdown** begins with n = 0, and since n is not greater than 0, it displays "Blastoff!" and returns.
    - The **countdown** that got n = 1 returns.
  - The **countdown** that got n = 2 returns.
- The **countdown** that got n = 3 returns.

8. Recursion

- We can write a function that prints a string n times:

```
def print_n_times(string, n):
    if n > 0:
        print(string)
        print_n_times(string, n - 1)
```

```
print_n_times('Spam ', 4)
```

```
Spam
Spam
Spam
Spam
```

- If n is positive, print_n_times displays the value of string and then calls itself, passing along string and n-1 as arguments.
- If n is 0 or negative, the condition is false and print_n_times does nothing.

- For simple examples like this, it is probably easier to use a for loop. But we will see examples later that are hard to write with a for loop and easy to write with recursion, so it is good to start early.

9. Stack Diagram

- To keep track of which variables can be used where, it is sometimes useful to draw a **stack diagram**. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to.
- Each function is represented by a **frame**. A frame is a box with the name of a function on the outside and the parameters and local variables of the function on the inside.

9. Stack Diagram

- Here's a stack diagram that shows the frames created when we called countdown with n = 3:

countdown	**n → 3**
countdown	**n → 2**
countdown	**n → 1**
countdown	**n → 0**

  – The four countdown frames have different values for the parameter n. The bottom of the stack, where n = 0, is called the **base case**. It does not make a recursive call, so there are no more frames.

10. Infinite Recursion

- If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as **infinite recursion**, and it is generally not a good idea.
- Every time recurse is called, it calls itself, which creates another frame. In Python, there is a limit to the number of frames that can be on the stack at the same time.
- If you encounter an infinite recursion by accident, review your function to confirm that there is a base case that does not make a recursive call. And if there is a base case, check whether you are guaranteed to reach it.

```
[1]: def recurse():
         recurse()
```

```
[2]: recurse()
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Cell In[2], line 1
----> 1 recurse()

Cell In[1], line 2, in recurse()
      1 def recurse():
----> 2     recurse()

Cell In[1], line 2, in recurse()
      1 def recurse():
----> 2     recurse()

    [... skipping similar frames: recurse at line 2 (2975 times)]

Cell In[1], line 2, in recurse()
      1 def recurse():
----> 2     recurse()

RecursionError: maximum recursion depth exceeded
```

11. Keyboard Input

- The programs we have written so far accept no input from the user.
- Python provides a built-in function called **input** that stops the program 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.
  - Before getting input from the user, you might want to display a prompt telling the user what to type. input can take a prompt as an argument.

```
name = input('What...is your name?\n')
name
```

```
What...is your name?
↑↓ for history. Search history with c-↑/c-↓
```

- The sequence **\n** at the end of the prompt represents a newline, which is a special character that causes a line break—that way the user's input appears below the prompt.

11. Keyboard Input

- If you expect the user to type an integer, you can use the **int** function to convert the return value to **int**:

```
prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
userInput = input(prompt)
speed = int(userInput)
```

```
What...is the airspeed velocity of an unladen swallow?
 32
```

  - But if they type something that's not an integer, you'll get a runtime error.

```
prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
userInput = input(prompt)
speed = int(userInput)
```

```
What...is the airspeed velocity of an unladen swallow?
 32.4
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[5], line 3
      1 prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
      2 userInput = input(prompt)
----> 3 speed = int(userInput)

ValueError: invalid literal for int() with base 10: '32.4'
```