Programming Interaction for Psychologists

A course in Python

Martin Schmettow and Lena Brandl

2018-11-29

Contents

Preface

1 Introduction

1.1 Installing Python

1. From https://www.anaconda.com/download/ download the 64-bit installer of Python 2.7
2. Run the downloaded file and
    a. Install for "just me"
    b. Accept the proposed destination folder
    c. Accept default advanced options
3. In the start menu find the program Anaconda Prompt
4. In the console, type: `conda install -c cogsci pygame` and hit Return. Confirm with y

For testing your Anaconda environment, follow these steps:

1. From the Start menu start Anaconda Navigator
2. From the Home screen launch Spyder
3. Download the Python file PIP_Stroop_interaction.py from the folder Week 1 on Google Drive https://goo.gl/UmBsil
4. Open this file in Spyder (File –> Open).
5. Run the program by clicking the green play button. You should see a window popping up, which greets you to the Stroop experiment.

And finally, it is a dear tradition to greet the world with your first program. Create a new file in Spyder and copy-paste the following lines into it. Then hit the play button once again.

```
x = 'hello, python world!'
print(x.split(' '))
```

```
## ['hello,', 'python', 'world!']
```

1.2 Computers are dumb

Programming is the process of giving instructions to a computer. The problem is ... computers are incredibly dumb. In order to get used to instructing dumb computers,

in the following exercise YOU will become the computer yourself. You will not just be the computer, you first have to program it, complying to some debilitating set of communication rules.

Your task is to create a theater play that resembles a computer to execute a program that Psychology researchers have used a thousand times in their labs. In the *Stroop task*, participants are shown a word that is written to the screen in one of three ink colors (red, blue, green). They are asked to name the ink color as quick as possible, usually by pressing one of four assigned keys (e.g., Y = red, X = blue, N = green). The task has a twist: the presented words are themselves color words. It can happen that the word "RED" appears in red ink, which we call the *congruent condition*, but it can also happen that the word "BLUE" appears in green letters, which is *incongruent*. Hundreds of experiments have shown, that in the incongruent condition people respond noticably slower. This observation is called the *Stroop effect* and dozens of theories have spawned from this observation. Discussing them is beyond the scope, but to not let you completely dissatisfied: people seem to always read the words, although they were not asked to do so and we can conclude:

1. people cannot *not* read
2. people do not (always) listen to what an experimenter wants from them

The following exercise is meant as a theater play and thus works best for a team of around 7-8 students. However, nothing keeps you from just being a drama poet and put down a script all by yourself.

Here are the rules for the theater play:

1. every actor (except the looper, see below) can only perform one simple action
2. actors can only talk to the looper, but not to each other
3. every actor can
    a) either do something on command
    b) or answer a question
4. actors who take commands
    a) only understand one command, like "pick a color"
    b) can only do one thing, like
        - picking an object
        - showing an object
        - making a particular change to an object
    c) in addition they can give and receive objects from the looper
5. actors who answer questions
    a) only understand one question type, for example: "What time is it now?"

b) can only give a one-word answer, for example:

- a word from a predefined set of words
- a number
- yes or no (as a special case of a pedefined set)

6. The *looper* is a special actor who

   a) is the only one can interact with other actors (speak, listen, exchange objects) and knows all the possible questions, answers and actions the other actors can perform.
   b) is slightly more intelligent than the other actors in that he/she can make decisions by combining information from various sources.
   c) can do simple calculations (like subtracting a number) and comparisons (like comparing two colors).
   d) unfortunately has no senses, but relies completely on what the actors tell
   e) speaks a continuous monologue to the audience, explaining all actions to the audience.

7. the *drawer* is a special actor whose task it is to present output to the user.
8. the *event handler* is a special actor whose task it is to take input from the user
9. the *user* is a special actor who only interacts with *drawer* and *event handler*.

For the play you will need the following requisites:

1. these instructions
2. three color word stencils
3. three ink colors (to put behind the stencil)
4. a bar with three buttons drawn on paper
5. a clock
6. paper and pencil

It is further recommended that one of the team is assigned the role of the *debugger*. This person does not take an active role, but watches over the compliance with the set of rules.

Have fun and do not let your intuition come between you and the task!

1.3 Dive in: a first example

In section 1.2 Computers are dumb, you have seen how debilitating it can be to program a computer to do the simplest things. Talking to a computer (as a programmer) requires to understand some fundamental ways of how computers work and learn their language. A good way to get started learning a language is just go there and expose yourself to

native speakers. Let's do that! The following text is a program the performs the Stroop task, just as in the theater play.

```
import pygame
import sys
from time import time
import random
from pygame.locals import *
from pygame.compat import unichr_, unicode_
##### CONFIG #####
# Colors abd screen
col_white = (250, 250, 250)
col_black = (0, 0, 0)
col_gray = (220, 220, 220)
col_red = (250, 0, 0)
col_green = (0, 200, 0)
col_blue = (0, 0, 250)
col_yellow = (250,250,0)
BACKGR_COL = col_gray
SCREEN_SIZE = (700, 500)
# Experiment
n_trials = 5
WORDS    = ("red", "green", "blue")
COLORS   = {"red": col_red,
            "green": col_green,
            "blue": col_blue}
KEYS     = {"red": K_b,
            "green": K_n,
            "blue": K_m}
### PYGAME STARTUP ###
pygame.init()
pygame.display.set_mode(SCREEN_SIZE)
pygame.display.set_caption("Stroop Test")
screen = pygame.display.get_surface()
# screen.fill(BACKGR_COL)
font = pygame.font.Font(None, 80)
font_small = pygame.font.Font(None, 40)
### MAIN PROGRAM ###
def main():

    STATE = "welcome"
    trial_number = 0

    while True:
```

```
    # refreshing the surface
    screen.fill(BACKGR_COL)
    # Event loop
    for event in pygame.event.get():

        # interactive transitionals
        if STATE == "welcome":
            if event.type == KEYDOWN and event.key == K_SPACE:
                STATE = "prepare_trial"
                print(STATE)
                continue

        if STATE == "trial":
            if event.type == KEYDOWN and event.key in KEYS.values():
                time_when_reacted = time()
                this_reaction_time = time_when_reacted - time_when_presented
                this_correctness = (event.key == KEYS[this_color])
                STATE = "feedback"
                print(STATE)
                continue

        if STATE == "feedback":
            if event.type == KEYDOWN and event.key == K_SPACE:
                if trial_number < n_trials:
                    STATE = "prepare_trial"
                else:
                    STATE = "goodbye"
                print(STATE)
                continue


        if event.type == QUIT:
            STATE = "quit"
            print(STATE)
            break

    # automatic transitionals
    if STATE == "prepare_trial":
            trial_number = trial_number + 1
            this_word  = pick_color()
            this_color = pick_color()
            time_when_presented = time()
            STATE = "show_trial"
```

```
                print(STATE)


        # Presentitionals
        if STATE == "welcome":
            draw_welcome()
            draw_button(SCREEN_SIZE[0]*1/5, 450, "Red: B", col_red)
            draw_button(SCREEN_SIZE[0]*3/5, 450, "Green: N", col_green)
            draw_button(SCREEN_SIZE[0]*4/5, 450, "Blue: M", col_blue)

        if STATE == "trial":
            draw_stimulus(this_color, this_word)
            draw_button(SCREEN_SIZE[0]*1/5, 450, "Red: B", col_red)
            draw_button(SCREEN_SIZE[0]*3/5, 450, "Green: N", col_green)
            draw_button(SCREEN_SIZE[0]*4/5, 450, "Blue: M", col_blue)

        if STATE == "feedback":
            draw_feedback(this_correctness, this_reaction_time)

        if STATE == "goodbye":
            draw_goodbye()

        if STATE == "quit":
            pygame.quit()
            sys.exit()

        # Updating the display
        pygame.display.update()

# Function definitions

def pick_color():
    """ Return a random word.
    """
    random_number = random.randint(0,2)
    return WORDS[random_number]
def draw_button(xpos, ypos, label, color):
    text = font_small.render(label, True, color, BACKGR_COL)
    text_rectangle = text.get_rect()
    text_rectangle.center = (xpos, ypos)
    screen.blit(text, text_rectangle)
def draw_welcome():
    text_surface = font.render("STROOP Experiment", True, col_black, BACKGR_COL)
```

```
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,150)
    screen.blit(text_surface, text_rectangle)
    text_surface = font_small.render("Press Spacebar to continue", True, col_black, BACKGR_COL
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,300)
    screen.blit(text_surface, text_rectangle)
def draw_stimulus(color, word):
    text_surface = font.render(word, True, COLORS[color], col_gray)
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,150)
    screen.blit(text_surface, text_rectangle)
def draw_feedback(correct, reaction_time):
    if correct:
        text_surface = font_small.render("correct", True, col_black, BACKGR_COL)
        text_rectangle = text_surface.get_rect()
        text_rectangle.center = (SCREEN_SIZE[0]/2.0,150)
        screen.blit(text_surface, text_rectangle)
        text_surface = font_small.render(str(int(reaction_time * 1000)) + "ms", True, col_blac
        text_rectangle = text_surface.get_rect()
        text_rectangle.center = (SCREEN_SIZE[0]/2.0,200)
        screen.blit(text_surface, text_rectangle)
    else:
        text_surface = font_small.render("Wrong key!", True, col_red, BACKGR_COL)
        text_rectangle = text_surface.get_rect()
        text_rectangle.center = (SCREEN_SIZE[0]/2.0,150)
        screen.blit(text_surface, text_rectangle)
    text_surface = font_small.render("Press Spacebar to continue", True, col_black, BACKGR_COL
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,300)
    screen.blit(text_surface, text_rectangle)
def draw_goodbye():
    text_surface = font_small.render("END OF THE EXPERIMENT", True, col_black, BACKGR_COL)
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,150)
    screen.blit(text_surface, text_rectangle)
    text_surface = font_small.render("Close the application.", True, col_black, BACKGR_COL)
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,200)
    screen.blit(text_surface, text_rectangle)

main()
```

At first sight, you'll probably see just gargle-bargle. But on closer examination, you may see one or the other thing you recognize. We start with the most visible feature of the Stroop task, colors. If you watch out for color words in the code, you'll find the following:

```
col_white = (250, 250, 250)
col_black = (0, 0, 0)
col_gray = (220, 220, 220)
col_red = (250, 0, 0)
col_green = (0, 200, 0)
col_blue = (0, 0, 250)
COLORS   = {"red": col_red,
            "green": col_green,
            "blue": col_blue}
WORDS    = ("red", "green", "blue")
```

The first paragraph defines each color used throughout the program as three numbers. This is because computer screens produce all colors by mixture of red, blue and green, which not coincidentily matches the three types of color vision sensors in our eyes. For example, black just means that all lights are out, hence the three zeroes. Red means that only the red channel is firing photons.

Whenever you see a single `=` character, that means that what is given to the right is stored in a *variable* to the left. The second paragraph just collects the three colors in one compound variable, which you will later learn to be a *dictionary*. Search for `COLOR` in the following code to see, how it is being used:

```
def draw_stimulus(color, word):
    text_surface = font.render(word, True, COLORS[color], col_gray)
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,150)
    screen.blit(text_surface, text_rectangle)
```

This code block is called a *function definition*. This function almost exactly represents what one of the actors did in the play: receiving a color word and a color, combine it into one Stroop trial and show it to the user. We can follow up how this function is used, by searching for "draw_stimulus" in the code:

```
if STATE == "present_trial":
            draw_stimulus(this_color, this_word)
            draw_button(SCREEN_SIZE[0]*1/5, 450, "Red: X", col_red)
            draw_button(SCREEN_SIZE[0]*3/5, 450, "Green: N", col_green)
            draw_button(SCREEN_SIZE[0]*4/5, 450, "Blue: M", col_blue)
```

What you see here is a so-called *conditional*. Only the most simple program are just a sequence of procedures. All "real" programs, and especially *interactive programs*, do certain things only when certain conditions are satisfied. Here the condition is that the current *state* of the program is to present a trial. If you look further around you will immediatly see that the variable `STATE` is used all over the place and it happens to always occur in one of two patterns in the code:

1. as a condition (`STATE == ...`)
2. as an assignment of new values (`STATE = ...`)

Note the difference between `=`, which stores a new value in a variable and `==` which means *equals*. Let's discover where the variable `STATE` actually gets such a value: `"present_trial"`:

```
if STATE == "feedback":
    if event.type == KEYDOWN and event.key == K_SPACE:
        if trial_number < 20:
            STATE = "present_trial"
        else:
            STATE = "goodbye"
        print(STATE)
```

Literally, this reads as: "If we are in state 'feedback' (where the participant is shown the results of the previous trial), and if the space key is pressed, and if we have not yet reached the maximum number of trials, then present the next trial (otherwise say good bye)"

These are precisely the rules the looper in our play had to have on her mind. If you look closely, you will find more of similar constructions, that change the state under certain conditions. These statements we later call *interaction conditionals*, and they control the flow of the program. This is why they are more complicated than the rest.

As a final endeavour, we try to find some clues how the measurement of *response time* is handled in the program. If you serach for "time", the first occurence is right at the top:

```
from time import time
```

Just as programmers can define functions, they can also make use of functions other programmers are already providing. Usually, they do so in larger packages of related functions, which are called *libraries*. This line of code means: from the library time, make available the function time. Where is this function used?

```
if STATE == "present_trial":
    trial_number = trial_number + 1
    this_word  = pick_color()
```

```
    this_color = pick_color()
    time_when_presented = time()
    STATE = "wait_for_response"
```

The only thing that happens here, is that in the moment where the trial is presented, the current absolute time is recorded in a variable `time_when_presented`. How is that used further down?

```
if STATE == "wait_for_response":
    if event.type == KEYDOWN and event.key in KEYS.values():
        time_when_reacted = time()
        this_reaction_time = time_when_reacted - time_when_presented
        this_correctness = (event.key == KEYS[this_color])
        STATE = "feedback"
```

When the user responds by pressing a button, that moment in time is recorded in another variable `time_when_reacted` and the current reaction time is computed as the time difference between reaction and presentation.

1.4 Exercises

1. Load the Stroop program into your Python editor and run it yourself.
2. Find further function definitions that resemble one of the actors in the play
3. If you have struggled with recording the response time in a compliant way, how would you do it, now that you have seen how the code works? How many actors are needed, and what precisely can they do?
4. Change the program such that it does 8 trials
5. Change the progran such that it uses orange instead of red (words and colors)
6. Change the program such that it uses Q, W, E as response keys
7. Change the program such that it performs the Stroop task with four words and colors

2 Variables, values and types

In chapter 1 Introduction, you took a closer look at the Stroop Task program. In doing so, you encountered components of a computer program called *variables*

```
col_white = (250, 250, 250)
col_black = (0, 0, 0)
col_gray = (220, 220, 220)
col_red = (250, 0, 0)
col_green = (0, 200, 0)
col_blue = (0, 0, 250)
COLORS   = {"red": col_red,
            "green": col_green,
            "blue": col_blue}
WORDS    = ("red", "green", "blue")
```

For instance, you learned that the variable `COLORS` collects combinations of numbers for red, green, and blue color in a compound variable, called a *dictionary*.

However, before we can talk about what compound variables such as dictionaries are, you will meet the most fundamental component of a computer program: *values*.

```
print(2 + 2)
```

```
## 4
```

```
print(20 / 4)
```

```
## 5
```

```
print("Hello, World!")
```

```
## Hello, World!
```

In the examples above, the *printed* outputs `4`, `5.0` and `Hello, World!` are *values*. Values can be numbers, but also words, or even whole sentences. In the interior of a

computer program, `4` and `Hello, World` serve the same purpose, they are both building blocks. Values are classified into different *classes*, or *data types*: `4` is a so-called *integer* and `"Hello, World"` is a *string*, so-called because it is a string of individual letters. The data type of a value determines which operations can be performed on a value, and how these operations are performed. You can think of operations such as arithmetic addition and subtraction, but also self-defined operations, called *functions*. You will learn about functions later in this book.

If you ever wonder what class a value belongs to, your Python interpreter can tell you.

```
print(type(4))
```

```
## <type 'int'>
```

```
print(type("Hello, World!"))
```

```
## <type 'str'>
```

You will learn more about the four basic data types in Section 2.4 Data types.

You will find the `print` function very useful; it makes some output visible by *printing* to the console of your programming environment whatever is written inside the two brackets `()`. Printing is a way to make things visible, such as the result of the simple calculation `2 + 2`. In the example above, if the `print` function were not included, your computer would perform the calculation `2 + 2` internally, but you would not see the result, `4`. Do not blame your computer, as we saw in chapter 1, computers are incredibly dumb. How is your computer supposed to know that in addition to making the calculation, it is also supposed to show you its result on the screen? Computers are unfortunately not gifted with common sense - you need to tell them precisely what to do.

2.1 Variables

Computer programs are usually required to store information in some way for later retrieval or manipulation. In the Stroop Task program we saw in chapter 1, for each trial, we need to store the color and the word that our computer picked. In addition, we want to record the reaction time of the subject. Otherwise, the Stroop Task program would be utterly useless for examining the relation between reaction times and (in)congruence between words and their colors. In other words, the Stroop task program would be useless if it could only manipulate values as we did when we calculated `2 + 2`. You see, when your computer performs a simple value manipulation you get a momentarily answer. At times, computers are too efficient for their own good. They quickly forget about the values that pass through their system. The very moment they finish a calculation,

computer programs forget the outcome of a calculation. They even forget that they ever performed the calculation (if you do not remind them that they indeed did). This is where *variables* come in handy. Variables are storage units in a computer's memory. A variable can store any kind of value. Unlike the result of simple value manipulations, such as `2 + 2`, variables have designated storage addresses in a computer's memory. You need a way to access variables in your computer's memory and hence, you give them a name. You can think of a variable's name as a unique identification address in the computer's memory. In fact, during a *value assignment*, a value is stored at a certain position in the computer's memory. This position as a precisely defined location, much like longitude and latitude coordinates describe a physical location in the Global Positioning System (GPS). The command by which a value is stored in a variable is called an *assignment statement*. It is common to say "a value is assigned to a variable".

```
a = 4
b = "This is a string"
print(a)
```

```
## 4
```

```
print(b)
```

```
## This is a string
```

The order in which the variable and a value are placed around the equal sign `=` is not arbitrary. Variables always have to be on the left side, values on the right side. Everything else will produce an error. As mentioned above, variables can store any kind of value. The right side can be a complex expression, which is not obviously a value at first sight.

```
a = 13*17/23
print(a)
```

```
## 9
```

The single equal sign, as in `a = 4` is exclusivey reserved for assignment statements. If you read a program out loud, practice saying

> a is assigned the value 4

A double equal expression, as in `a == 4` is used for comparing two values, as in "Does the value of 'a' equal 4?" That is completely different to an assignment and will be covered in Section 2.3 Data types and in more detail, in Chapter 3.

Programmers have much freedom in naming variables, but some strict rules exist, as well as some conventions. Some of them are good practice and others will throw syntax errors if not adhered to. In the beginning, you may think of good practice guidelines as superfluous, but you will learn to value them once you start making more complex programs.

Heeding them pays off!

1. Programmers give meaningful names to variables in order to make their code more readable for others and themselves. Without meaningful names, they easily get lost when reading through their code again. Even to them their own code may appear as nothing more than gargle-bargle.
2. The first character of a variable must be a letter of the alphabet, either uppercase or lowercase ASCII or Unicode, or an underscore.
3. The rest of a variable name may consist of letters (uppercase or lowercase ASCII or Unicode characters), underscores, or digits (0-9).
4. Variable names are case-sensitive. This means that the variables `myString` and `mystring` are **not** treated as the same variable by your computer.
5. You may still get sytnax errors despite adhering to all rules named above. Chances are high that you tried to use one of Python's *keywords* as a variable name. Python uses a number of keywords to recognize the structure of a program.`if` and `for` are keywords for example. Your Python interpreter can tell you the complete list of keywords known to your Python version:

```
import keyword
print(keyword.kwlist)
```

```
## ['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'els
```

2.2 Stroop Task variables

Now that you read about how variables are assigned their values, let's have a look at how variables are used in the interactive Stroop program.

```
# a list for gathering the response times
RT = []
# remember when stimulus was presented
time_when_presented = time()
# remember when the user has reacted
time_when_reacted = time()
# calculate the response time
```

```
this_reaction_time = time_when_reacted - time_when_presented
RT.append(this_reaction_time)
```

In the first line of the above code snippets from the Stroop Task program, an empty *list* is assigned to the variable `RT`. The purpose of this list is to store reaction times during the Stroop experiment. You will learn more about *lists* in Chapter 4.1. For now it suffices to know that lists are yet another data type, capable of containing values of multiple data types at the same time! You may reckon that they are the ultimate structure for storing information. The variables `time_when_presented` and `time_when_reacted` respectively store the point in time (using the *function* `time`) at a certain event. `time_when_presented` records the moment in time when a stimulus is presented and `time_when_reacted` saves the moment in time when the subject reacts to the stimulus. The reaction time of the subject is then calculated by *subtracting* one stored point in time from the other and saved in the variable `this_reaction_time`. Finally, the subject's reaction time is *appended* to `RT`, meaning that `this_reaction_time` is saved in `RT` before the program proceeds to the next trial. A disclaimer; the code snippets above obviously reduce the mechanics of the Stroop Task program to some highlighted variable assignments. This means that the code lacks the actual dynamics of the program and will not run as is. But it captures the basic idea of how the program records and manipulates a participant's reaction time with the help of variables.

2.3 How Variables Are Used

More formally, variables are an essential component of any program. As such, their applications are numerous. This section provides but a glimpse into the vast and creative application possiblities of variables. Do not worry, you will soon get a hang of it once you start doing some programming yourself. One common way of using variables is storing the output of *functions*:

```
import random
a = random.randint(0,10)
print(a)
```

```
## 7
```

You will learn in detail about functions in Chapter 6. For now it is good enough to know that the function `random.randint(0,10)` randomly picks an integer between 0 and 10, and *returns* the chosen integer. Whatever number is picked by the function is stored in the variable `number` and can easily be retrieved and used for other purposes elsewhere. If this piece of information does not satisfy you, simply type `help(random)` after importing `random` and your interpreter will give you a more detailed description of the *class* `random`.

Variables are mutable, meaning that their value can be changed after an initial assignment. This is very useful for structuring a program:

```
# determining the maximum number of trials
nTrials = 3
# setting a counting variable to 0
# at the beginning of the program
n = 0
# adding 1 to n as long as n does not exceed nTrials
while(n < nTrials):
    # do something
    n = n + 1
```

This comes in handy if you want to repeat a part of your program until a certain condition is fulfilled. For instance, imagine taking measurements for a fixed number of trials for a psychological experiment. In the example above, `nTrials` and `n` are used for creating a counting mechanism until the desired number of repeated measurements is reached. Note how the old value of `n` is used to mutate the value of `n` after one execution cycle of the program in `n = n + 1`. In particular, after the first execution of the `while` part of the code snippet,`n` will have the value `1` instead of `0`! If you are not yet fully getting the hang of it, do not worry. You will learn more about the use of *conditionals* and the *while loop* in Chapter 3 and Chapter 5, respectively. The basic idea behind the above code snippet is to show you how the mutability of variables can be used to create a counting mechanism in computer programs.

2.4 Data types

Earlier in this chapter, you learned that each value belongs to a certain *class* or *data type*. This section will introduce the *basic data types*: *integers*, *floats*, *strings*, and *booleans*. Apart from the basic data types, Python knows numerous specialized types. Earlier in this chapter, you already encountered two such specialized data types: *dictionaries* and *lists*. Literally any computer program contains values of at least the basic types and thus, it seems to be reasonable to say a word or two about them.

2.4.1 Numbers

There are two main types of numbers in Python, *integers* and *floating point numbers*, or *floats* for short. Integers are whole numbers, such as `6` and `1234`, while floats are numbers including a decimal point, such as `2.4` and `45.768`. But wait, what happens if we add a float to a variable of type integer? Let's try!

```
number = 7
print(type(number))
```

```
## <type 'int'>
```

```
number = number + 3.5
print(type(number))
```

```
## <type 'float'>
```

The variable changed from type *int* to *float*! This is an example of implicit *typecasting*. Typecasting means changing the type of a variable, and it can be done explicitly and implicitly. At times you will want to change the type of a variable to suit your needs. For this, you use explicit typecasting:

```
number = 10.5
int(number)
10
```

And here you discover one tricky thing about typecasting *floats* to *integers*. Python does not round off the value of floats when typecasting them into integers, it simply cuts off whatever there is after the decimal point. This can become especially tricky when typecasting implicitly and should be monitored with caution.

2.4.2 Strings

If numbers are *integers* or *floats*, what about values such as `"17"` and `"13.65"`? They look like numbers, but they are wrapped in quotation marks like *strings*.

```
print(type("17"))
print(type("13.65"))
```

They are strings! In Python, strings can be enclosed in either single quotes (`'`), or double quotes (`"`), or three of each (`'''` or `"""`). But what if your string includes a quote and you want to indicate this by using quotation marks *within* your string? Using the same quotation marks as used for defining the string will prematurely end the string.

```
quote = "Freud: "The mind is like an iceberg, it floats with one-seventh of its bulk above wat

SyntaxError: invalid syntax
```

Instead, when using single quotes, you can use double quotes or triple quotes inside them:

```
quote = 'Carl Rogers: "How can I provide a relationship which this person may use f
```

Double quoted strings can have single quotes and triple quotes inside them:

```
quote = "Ivan Pavlov's best known findings relate to the phenomenon of conditioning
```

Triple quoted strings can include either single or double quotes. Strings are basically a sequence of single characters tied together and unlike variable names, strings may include spaces! They are usually used to display text or to export information out of the program. You need to export data out of the program to save the data you collected during an experiment somewhere on your hard disk.

2.4.3 Booleans

A *boolean expression* is an expression that is either `True` or `False`.

```
print(5 == (3+2))
```

```
## True
```

```
print(5 == 6)
```

```
## False
```

```
j = "hel"
print("hello" == j + "lo")
```

```
## True
```

`True` and `False` are special values of type boolean; they are no strings!

```
print(type(True))
```

```
## <type 'bool'>
```

```
print(type(False))
```

```
## <type 'bool'>
```

A *boolean expression* consists of two operands, located left and right of a *relational* or *comparative* operator. There are six comparison operators in Python:

- `x == y # x equals y`
- `x != y # x is not equal to y`
- `x > y  # x is greater than y`
- `x < y  # x is smaller than y`
- `x >= y # x is greater or equal to y`
- `x <= y # x is smaller or equal to y`

These operators are probably familiar to you, but their usage in Python differs from the mathematical symbols you know from highschool. One very common confusion surrounds the `equals` (`==`) operator. Remember that `=` is reserved as assignment operator and `==` is used for comparisons. There is no such thing as `=>` or `=<`.
Apart from that, boolean values can be handled exactly as any other value, that is, they can be assigned to variables, printed and so on.

```
number_of_EC_required_for_BSA = 45
passed_BSA = number_of_EC_required_for_BSA <= 50
print(passed_BSA)
```

```
## True
```

2.5 Common errors

1. Confusing the *assignment operator* (`=`) with the comparative *equals* operator (`==`). The first is solely used for assigning values to variables as in

```
myString = "This is an assignment statement"
```

the latter for comparing two values

```
17+3 == 21
```

Usually, when you confuse the assignment operator with the equals operator, you will get a syntax error.

```
17+3 = 21
```

```
SyntaxError: can't assign to operator
```

2. Forgetting to typecast other data types when incorporating them into strings

```
myGrade = 8
print("I got an " + myGrade + " on the last exam!")
```

```
TypeError: cannot concatenate 'str' and 'int' objects
```

Often, you will want to export some arithmetic result (such as a participant's reaction time in the Stroop Task program) or other non-string data out of the program, or you simply want to print them to the screen. This is usually done by first converting the data into type string and then writing to some external file (such as good old plain text files) or using the *print command* to write to the console. Quite often people forget that *explicit typecasting* is required when incorporating non-string values into a sequence of strings.

```
myGrade = 8
print("I got an " + str(myGrade) + " on the last exam!")
```

```
## I got an 8 on the last exam!
```

3. Unintentionally cutting off decimals when typecasting to *integers*.

```
import numpy
RT = [3.749,2.998,3.0147]
mean_RT = numpy.mean(RT)
print("The subject had a mean reaction time of " + str(int(mean_RT)) + " seconds.")
```

```
## The subject had a mean reaction time of 3 seconds.
```

Obviously, a mean reaction time of exactly `3` is unrealistic with `3.749`, `2.998`, and `3.0147` as individual trial reaction times. You may feel inclined to typecast anyway to shorten the numerical string expression, but remember that typecasting from *float* to *integer* results in the loss of any information after the decimal point. Besides, usually there are better ways for rounding off floating points; consider the `round` method for instance.

```
import numpy
RT = [3.749,2.998,3.0147]
mean_RT = numpy.mean(RT)
print("The subject had a mean reaction time of " + str(round(mean_RT, 3)) + " seconds.")
```

```
## The subject had a mean reaction time of 3.254 seconds.
```

4. Forgetting to reassign the mutated value of a variable

```
count = 1
count + 1
print(count)
```

```
## 1
```

Remember to assign the new value of a mutated variable to either a new variable or the original variable name; your computer will otherwise ignore the variable mutation and keep the variable's old value in memory.

```
count = 1
count = count + 1
print(count)
```

```
## 2
```

2.6 Debugging

Despite all efforts and skill, programmers encounter programming errors on a daily (if not hourly) basis. In this, programming beginners and experts are no different. In fact, debugging is a very challenging and instructive task in itself.

Apart from some general good debugging practices introduced in chapter 9 Debugging, there are some tricks that are especially useful in tracing errors that originate from the variables in your program.

1. Checking all of your equals `==` and assignment `=` operators is a good start when tracking down bugs suspected to originate from variables. For this you can either scan your code from top to bottom manually or use the search function of your programming environment. Use `Cltr` + `F` to search for an expression in your code (in this case either `=` or `==`, or the name of the variable you expect to be the culprit). Then, for each instance of the equals `==` or `=` assignment operator, or the

variable itself, ask yourself whether you used the right operator for the intended job. Confusing the equals and assignment operator usually results in a syntax error, but a syntax error does not necessarily need to originate from confusing operators. In fact, syntax errors are undoubtly the most generic errors in programming.

2. Once you are positive that the bug is not caused by confusing the equals with the assignment operator, you should take a closer look at the operations you perform on your variables (such as comparing variables or performing arithmetic calculations). Operations are another potential source of bugs.

```
a = 1
b = "1"
print(a + b)
```

In the example above, it is evident what goes wrong. `Strings` and `integers` cannot be added as two numbers are summed. A `type error` is thrown. However, in more complex programs it is often less evident where the bug originates from. Therefore, it is useful to check the data type of your variables at multiple locations in your program. Who knows, you may have unknowingly converted a variable to another type somewhere in your code and at a later moment, this causes your program to throw an error. Use the functions `type()` and `print()` to display the data type of your variables at different places in your code and hit `run`!

```
a = 1
b = "1"
print(type(a))
```

```
## <type 'int'>
```

```
print(type(b))
```

```
## <type 'str'>
```

3. Performing operations on variables may still have unexpected results, even when no type errors are involved. Arithmetic (`+`, `-`, `*`, `/`) and *Boolean* operators follow rules of precedence. Arithmetic operators follow the same rules of precedence as you learned about in highschool mathematics. All arithmetic operators also precede any Boolean operator, meaning that in the code snippet below `1 + 1` is evaluated before being compared to `2`.

```
print(1 + 1 == 2)
```

```
## True
```

As a debugging strategy, it is useful to change the precedence of operators with the help of brackets `()`. The bug may originate in a false assumption about operator precedences. Using brackets, you can make sure that operations are executed in the exact order you intended.

4. When you neither confused the equals with the assignment operator, nor made a type error, nor had false assumptions about operator precedences, the last resort is to mentally follow the flow of your program and to check whether your variables are assigned the expected value at all times. For instance, in the code snippet below, you may expect the result to be `1` since you added `2` to the initial value of `a` and `2 / 2` is `1`.

```
a = 0
b = 2
a + 2
print(a / b)
```

```
## 0
```

However, `a` was never assigned the new value of `2` and `0 / 2` is obviously `0`. It is easy to overlook such a mistake when scanning through your code. Therefore, checking which values are assigned to your variables as you read through your program is very helpful. You could for instance adjust the code above if you are puzzled why the result is not `2`, like so

```
a = 0
b = 2
a + 2
print(a, b, a / b)
```

```
## (0, 2, 0)
```

which prints first the value of `a`, then `b` and then the result of `a / b` to the console.

2.7 Exercises

2.7.1 Exercise 1. Operators

What is the output of the following code snippets?

```
x = 2
y = 1.0
print(x + y)
```

```
x = 4.0
y = x + 1
print(x + y)
```

```
x = 1
y = "1.0"
print(x + y)
```

```
x = 4
y = 4.0
print(x == y)
```

```
x = 12
y = x / 2
print(y >= 5)
```

2.7.2 Exercise 2. Values

Which statements about values are true?

- Values are storage units in a computer's memory and thus, they are persistent throughout a computer program
- Each value exclusively belongs to one data type
- There are four basic data types that occur in virtually any computer program: strings, integers, floats and dictionaries
- Variables can be assigned any type of value
- Any operation (e.g. subtraction) can be performed on any two pairs of values

2.7.3 Exercise 3. Mini programs

What is the output of the following mini programs?

```
x = "1.0"
y = "1"
print(x + y)
```

```
x = 5
y = 2
print((x / y) == 2.5)
```

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

2.7.4 Exercise 4. Debugging

What goes wrong in the following code snippets?

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

- The letter `x` may not repeat in multiple assignment statements
- Variable declarations must be ordered in descending order, so the value `2` should be assigned before `1`
- An illegal variable name is used

```
x = 3
y = "3"
print(x = y)
```

- A wrong arithmetic equation is printed, `3` does not equal `"3"`
- Strings and integers are not comparable using boolean operators
- The assignment operator is used instead of a comparative operator

```
name = "Colin"
print(This is + name)
```

- Variables cannot be called in `print` statements
- The `name` variable needs to be typecasted into a String
- The content inside the brackets of the `print` statement lacks quotation marks

2.7.5 Exercise 5. String concatenation

Consider the three blocks of code below.

1. What is the output of the respective *print statement* at the end of each block?

2. Describe in your own words what *string concatenation* is.
3. There is an error hidden in the code. Explain what is going wrong and adjust the code so that it runs without throwing any errors.

```
myString = "The statement 'Veni, vidi, vici.'"
s1 = " was coined by Gaius Julius Caesar."
myString = myString + s1
print(myString)
myString = "The statement 'Veni, vidi, vici.'"
s1 = " was coined by Gaius Julius Caesar."
s2 = ", 'I came, I saw, I conquered.',"
myString = myString + s2 + s1
print(myString)
myString = "The statement 'Veni, vidi, vici.'"
s1 = " was coined by Gaius Julius Caesar,"
s2 = " supposedly around "
year = 47
s3 = " BC."
myString = myString + s1 + s2 + year + s3
print(myString)
```

Want to know more about *string concatenation*? The following website provides a gentle introduction to the topic: Python for Beginners: String concatenation and formatting in Python

2.7.6 Exercise 6. Variable names

For the following variable names indicate whether the name is legal or not. For illegal variable names, point out what makes the name violate rules and/or conventions.

```
Reg       = "What have the Romans ever given us?"
man1      = "The aqueduct?"
2ndMan    = "The sanitation"
m@n3      = "And the roads"
man_4     = "Irrigation"
man 5     = "Medicine"
SixthMan  = "Education"
man_no._7 = "And the wine!"
8thMan    = "Public votes"
no9atReg  = "Public order; it's finally safe to walk in the streets at night."
reg       = """All right. But apart from sanitation,
            medicine, education, wine, public order,
            irrigation, roads, a fresh water system,
```

```
and public health;
what have the Romans ever done for us?"""
```

2.7.7 Exercise 7. Stroop task welcome message

In this exercise you will add a welcome message to the starting screen of the Stroop Task program.

1. Open the file `stroop_modifiedCh1Ex3.py`. Run it.
2. Make two new variables, `msgColor` and `msgText`. Choose any color and welcome message you want.
3. Read and try to understand the code written in lines 134-137. Try to formulate in your own words what each line of the code does.
4. Uncomment lines 134-137. *Hint*: In Python, a commented line starts with a `#`. Commented lines will not be executed.
5. Hit `Run file` again and see your first modification of the Stroop task algorithm in action.

2.7.8 Exercise 8. Printing

The following code prints part of a Methods section. Copy it and run it.

```
participants = 52.0
trials = "200"
experimental_sessions = 3
trials_pp = trials * experimental_sessions
conditions = 4
"""
to be implemented by you
condition1 =
condition2 =
condition3 =
condition4 =
"""
print("In total,", participants, "participants participated in the study.")
print("A 2x2 factorial between-subjects design was employed.")
print("The study examined the interaction of two independent variables: ")
print("task difficulty (easy, difficult) and time (limited, unlimited).")
print(conditions, "conditions were devised, plus a control condition.")
print("The conditions were:",condition1,condition2,condition3,condition4)
print("Participants were tested in", experimental_sessions, "experimental sessions.")
print("Each session consisted of",trials,"trials.")
print("In total, each partcipant thus completed", trials_pp, "trials.")
```

1. The first time you run the script, you get an error. Initialize the variables `condition1, condition2, condition3, condition4` to solve the error. Fill in semantically and syntactically correct values for the four condition variables. For example, in the context of the given Method excerpt, it would not make sense to say that one of the four conditions was `easy-locationA-limited`, right?
2. Once you have filled in the blank variable declarations, you get a a curious output for the number of trials per person. Adjust the code so that the correct number of trails per person is printed. Explain what happened in your own words.
3. `52.0` participants looks a bit ugly when printed as part of the text. Adjust the code in such a way that only full integers are printed to the console.

2.7.9 Exercise 9. Using Python as a calculator

Make a python script `Ch1Ex5.py` and implement the following pseudo code:

- assign the value 37 to the variable `n1`
- assign the value 456 to `n2`
- calculate 1027 modulo n1 and assign the result to `n3`
- divide `n2` by `n3`
- add 4 to `n2`
- calculate `n2` modulo 5 and assign the result to `n4`
- subtract 17 from `n4`
- calculate 65 modulo `n4` and divide the result by 2

What is the resulting value of `n4`?

2.7.10 Exercise 10. A Boolean puzzle

Fill in the blank spots (indicated by a question mark `?`) in the following code in such a way that `True` Boolean values are returned. You can use your Python interpreter for calculations.

```
n1 = 238
n2 = 17

print(n1 ? n2)
print(n1 / ? == 14)
print(n1 * ? == n2)
print(n1 + ? == n2 - ? and n1 + ? == 972%243 and n2 - ? == 0)
print(n2 * ? == n1*47)
print(n1 / ? == n2 / ? * ?)
```

2.8 Think Further

For more information and exercise on Variables and Data Types you may want to consult some of the following resources:

- Downey, A. (2012). Think Python. O'Reilly Media, Inc.. Chapter 2. Variables, expressions and statements
- Official Python Tutorial, parts 3.1.1 Numbers and 3.1.2 Strings
- Learnpython "Variables and Types" interactive tutorial
- Swaroop, C. H. (2003). A Byte of Python. Chapter 6 "Basics"

3 Conditionals

```
if 1 == 1:
  print('Hello World')
```

```
## Hello World
```

3.1 Boolean logic

In Chapter 2 Variables, you read about simple *Boolean expressions* that produce *Boolean values* (`True`, `False`) with the help of *comparative operators* (`==`,`!=`,`>`,`<`,`>=`,`<=`). In this chapter, we will first take a closer look at Boolean expressions as we work towards introducing the main actor in regulating the flow of a computer program, *conditionals*. In chapter 1 Introduction you took a closer look at the Stroop Task program. You saw that the program shifts between different stages of the Stroop experiment, among which preparing a trial and waiting for the participant's response. These experimental stages are mirrored in the different *states* of the Stroop Task program. A simple lamp, for example, can be regarded as having two possible states, namely being either switched *on* or *off*. The Stroop Task program knows more states than simply being switched on or off, but just as a simple lamp, it needs some kind of switch to transit from one state to another. This is were *conditionals* come in handy as you will see at the end of this chapter. But first things first, let's take a look at the following two examples:

```
print(10 > 5 and 5 > 2)
```

```
## True
```

```
print(True or False)
```

```
## True
```

```
print(True and not False)
```

```
## True
```

```
print(not True and False)
```

```
## False
```

`and`, `or`, and `not` are the three *logical operators* used in Python. They allow us to build more complex Boolean expressions from simpler ones. Sometimes these operators confuse people because they are used differently in informal language. Therefore, it is important to understand what they formally mean in Python (and math and all other programming languages). The following *truth table* displays what these operators exactly mean for two variables `a` and `b` for all combinations of values:

a	b	not a	a and b	a or b	not a or b
True	True	False	True	True	True
True	False	False	False	True	False
False	True	True	False	True	True
False	False	True	False	False	True

There exists a set of rules for working with Boolean values, much like highschool mathematical algebra. First of all, the logical operators `and`, `or`, and `not` differ in their priority. In highschool, you learned that given a formula `a + b * c`, you first multiply `b` and `c` before adding `a`. This is the same as saying that `*` has a higher *precedence* than `+`. From highest to lowest the precedence ranking for the three logical operators is: `not`, `and`, `or`. In addition, the six *comparative* operators (`==`,`!=`,`>`,`<`,`>=`,`<=`) precede all logical operators.

As in mathematics, brackets can be used to change the order of operations.

```
print(not True or 1 == 1)
```

```
## True
```

Also notice that

```
print(not True and False)
```

```
## False
```

is not the same as

```
print(not (True and False))
```

```
## True
```

Finally, in Python code you will frequently encounter numerical values as part of Boolean expressions, such as in the following:

```
trial = 3
max_trials = 6
correct = True
print((max_trials - trial) and correct)
```

```
## True
```

Here, the numerical expression is evaluated in the context of the Boolean operator `and` and is converted to Boolean. When that happens, zero becomes `False` and any other valuebecomes `True`, *including negative values* :

```
print(-2 and True)
```

```
## True
```

That may not be what you expected, and the whole situation with implicit typecasts to Boolean is even worse, as the typecast depends on the order.

```
print(True and 2)
```

```
## 2
```

Certainly, there are some reasons, why Python behaves that way, but it is far from being intuitive (breaking the law of commutivity for the `and` operator!). Therefore one should *always do explicitly Boolean expressions*, such as the following:

```
trial = 3
max_trials = 6
correct = True
print((max_trials - trial) > 0  and correct)
```

```
## True
```

3.2 if this is true, then do this

Boolean values are essential for regulating the flow of a computer program as they provide the conditions under which certain blocks of code are executed. The simplest form of a *conditional* is the *if statement*.

```
a = 0
if (a < 0):
    print("a is smaller than 0")
if (a >= 0):
    print("a is larger than, or equal to 0")
```

```
## a is larger than, or equal to 0
```

You can read *if statements* as

> If whatever is written within the curved brackets equals `True`,
> then, and only then, the intended (from intendation, not intention!) code within the statement is evaluated.

An `if statement` consists of a *header* and a *body*. The header begins with the *keyword* `if`, followed by a *Boolean expression* (the *condition*) and ends with a colon (`:`). The brackets around the condition are optional, but recommended to facilitate readability. Only if the given condition in the *head* is met, the program will read and execute the code written in the statement's *body*. Note that Python requires that the statements written in the body of an `if statement` are indended consistently (commonly by four spaces). The intended statements are called a *block* and the first unintended statement designates the end of the block. The usefulness of intendation may at first not be apparent to you, but trust me, it facilitates reading complex programs enormously.

3.3 else do that

Actually, you can simplify the code above by using an *else clause*:

```
a = 0
if (a < 0):
    print("a is smaller than 0")
else:
    print("a is larger than, or equal to 0")
```

```
## a is larger than, or equal to 0
```

But, take great care: the conditions `a < 0` and `a >= 0` are special in that they are:

- mutually exclusive; only one can be true at any time
- complete; there is no third option.

else clauses are optional, but they often simplify your code. If you are dealing with two complete, mutually exclusive conditions, then using an `if/else clause` is the way to go. If there are more than two choices, you can either use *nested* conditionals or `if ... elif ... else`:

```
a = 0
if (a < 0):
    print("a is smaller than 0")
else:
    if (a == 0):
        print("a equals 0")
    else:
        print("a is larger than 0")
```

```
## a equals 0
```

The indentation makes the structure of the program visible. However, most of the time, nested conditionals complicate your code unecessarily, making it more difficult to read. It is commonly regarded as good practice to keep the usage of nested conditionals to a minimum.

Then again, how do you implement more than two execution branches, without resorting to nested conditionals? Let's pretend that it is incredibly important to your program that it does make a difference between the case that `a` is larger than 0, compared to when `a` equals 0.

```
a = 0
if (a < 0):
    print("a is smaller than 0")
elif (a == 0):
    print("a equals 0")
else:
    print("a is larger than 0")
```

```
## a equals 0
```

The *elif statement* (short for `else if statement`) allows you to include more than two alternative conditions. In theory, you can build an endless chain of alternative

conditions using `elif`. Whether it is useful to do so, is, of course, a different question. Whenever `elif` is included in a train of conditions, the whole structure is called a *chained conditional*. Note that when using an *else clause* in a chain of conditionals, it always marks the end of the chain. Furthermore, each condition is checked in order. If the first is `False`, the next is checked, and so on. Note that only the first `True` branch is executed, even if more than one condition is `True`. There is nothing like "more true" in Boolean logic.

```
a = 1
if (a >= 0):
    print("a is greater than or equals 0")
elif (a == 1):
    print("a equals 1")
else:
    print("a is smaller than 0")
```

```
## a is greater than or equals 0
```

Alternatively, nesting can often be circumvented by combining simple Boolean expressions using *logical operators*:

```
a = 5
if (a > -5):
    if (a >= 0):
        print("a is a positive digit")
```

```
a = 5
if (a > -5 and a >= 0):
    print("a is a positive digit")
```

3.4 Bringing interaction to life: Conditional state transitions

In programming interactive software, such as the Stroop experiment, conditionals are essential to control the flow of the program. By controlling the *flow*, we mean how the program changes states in reaction to the input of the user. Without the programmer's intervention, a program is executed from top to bottom. You can imagine that the Stroop Task program needs to be more flexible than that, depending on the actions of the participant. Take a closer look at how the Stroop Task program transits from the welcome screen to the first trial:

```
STATE = "welcome"
if STATE == "welcome":
    if event.type == KEYDOWN and event.key == K_SPACE:
        STATE = "prepare_next_trial"
        print(STATE)
```

In plain English, this would be the same as telling the program to switch from state `"welcome"` to state `"prepare_next_trial"` upon pressing the `SPACE` key. To see this in action, run the Stroop Task program, and watch closely what is printed to the console. You will see that whenever the program transits into a new state, the state of the program is printed to the console.

3.4.1 Transition tables

In the previous section, you saw how conditionals regulate the flow of a computer program. Depending on specific user actions, the state of the program transits into another state. When you first encounter the code of an interactive program such as the Stroop Task program, writing out all possible state transitions in a *transition table* helps to understand the structure of the program. A transition table lists all states against each other in a table and documents how the program transits from one state to another. The empty transition table for the Stroop Task program is shown below.

		welcome	present_trial	wait_for_response	feedback	goodbye	quit
FROM	welcome						
	present_trial						
	wait_for_response						
	feedback						
	goodbye						
	quit						

By taking a close look at how the Stroop Task program transits from one state to another, the transition table can be filled in. The changing state part of the Stroop Task program is displayed below, together with the completed transition table for the Stroop Task program.

```
KEYS     = {"red": K_b,
            "green": K_n,
            "blue": K_m}

# Changing states
for event in pygame.event.get():
```

```
    if STATE == "welcome":
        if event.type == KEYDOWN and event.key == K_SPACE:
            STATE = "present_trial"
            print(STATE)

    if STATE == "present_trial":
        trial_number = trial_number + 1
        this_word  = pick_color()
        this_color = pick_color()
        time_when_presented = time()
        STATE = "wait_for_response"
        print(STATE)

    if STATE == "wait_for_response":
        if event.type == KEYDOWN and event.key in KEYS.values():
            time_when_reacted = time()
            this_reaction_time = time_when_reacted - time_when_presented
            this_correctness = (event.key == KEYS[this_color])
            STATE = "feedback"
            print(STATE)

    if STATE == "feedback":
        if event.type == KEYDOWN and event.key == K_SPACE:
            if trial_number < 20:
                STATE = "present_trial"
            else:
                STATE = "goodbye"
            print(STATE)

    if event.type == QUIT:
        STATE = "quit"
```

		welcome	present_trial	wait_for_response	feedback	gooc
FROM	welcome		SPACE			
	present_trial			auto		
	wait_for_response				b OR n OR m	
	feedback		SPACE			SPA
	goodbye					
	quit					

The transition from state `present_trial` to state `wait_for_response` occurs automat-

ically once all preparations for the trial are finished. Generally, we distinguish between two types of transitions, *interactive changes* (ITC) and *automatic changes* (ATC). While an input from the user is required in interactive state transitions, the program changes from one state to another on its own in automatic transitions. It is generally a good idea to separate ITCs from ATCs in your code to make the structure of your program more apparent.

Apart from understanding another's code, transition tables are also useful when you conceive your own program. They serve as an interaction skeleton to which you gradually add functionality as you develop your program.

3.4.2 State charts

State charts are another tool for representing the interaction of a program. They are especially useful for developing your own programs. State charts serve as a blueprint. They depict the basic structure of your program to which you gradually add functionality. It helps to keep the basic functionality in sight while developing. You can easily get lost in small, optional features while the basic structure of your program is still unfinished. Below you see a state chart of the interaction in the Stroop Task program. States are shown as squares and conditions as diamonds. Arrows indicate the flow of the program, their labels give more information about state transition changes. Note how the state transition from `present_trial` to `wait_for_response` has no label. When you look into the code of the Stroop Task program, you will notice that the transition between these states is automatic with no further condition to be fulfilled.

3.5 Common errors

1. Confusing the *assignment operator* (`=`) with the comparative *equals operator* (`==`).

```
a = 5
if (a = 5):
    print("Yes, a is indeed equal to 5!")

SytnaxError: invalid sytnax
```

Yes, confusing the assignment operator with the comparative equals operator will be a re-occuring issue. For human beings it is just too easy to type `=` when they actually mean `==` in Python. Being a psychologist, go figure.

2. The most common syntax error involving *conditionals* is probably forgetting the colon at the end of the *head* of the `if statement`.

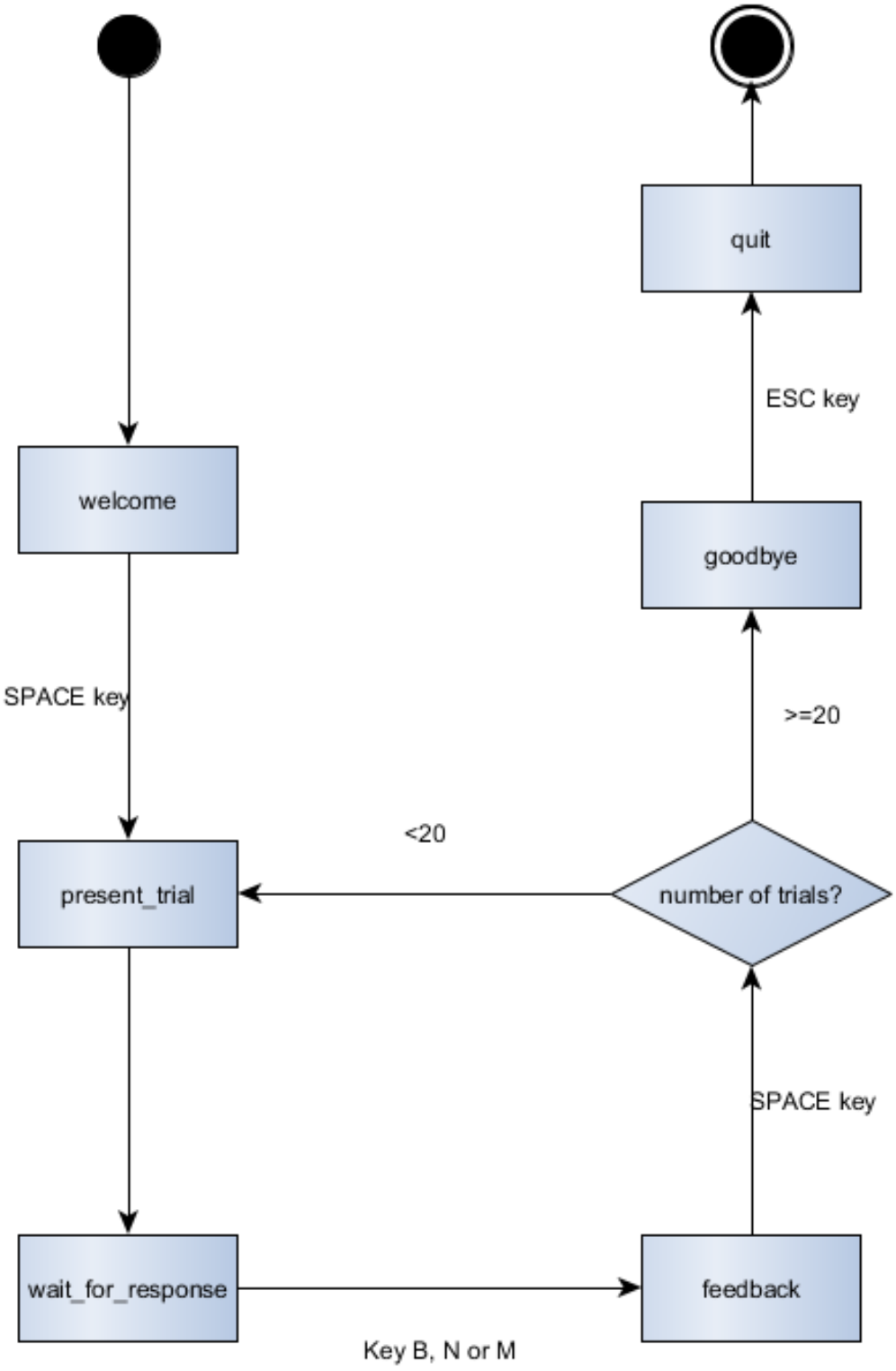

quit
ESC key
welcome
goodbye
SPACE key
>=20
<20
present_trial
number of trials?
SPACE key
wait_for_response
feedback
Key B, N or M

Figure 3.1:

```
if (1 == 1)
    print("Hello, World!")
```

Whenever you encounter a syntax error in your conditionals, you should first check whether you forgot a colon.

```
if (1 == 1):
    print("Hello, World!")
```

3. Exclusively using `if clauses` in a chain of conditionals. This is a tricky one because the consequences may not immediately be apparent. Plus, no error is necessarily thrown. Still, you program somehow does not do what you expect it to do. In truth, exclusively using `if clauses` is not wrong in itself; but chances are high that you intended to use a chain of `if/elif clauses` instead. Let's take a look at the following two code snippets to illustrate the difference between a chain of `if clauses` and a chain of `if/elif clauses`.

```
a = 6
if (a % 2 == 0):
    print("a is divisible by 2")
elif (a % 3 == 0):
    print("a is divisible by 3")
elif (a % 6 == 0):
    print("a is divisible by 6")
else:
    print("a is not divisible by 2, 3 or 6")
```

```
## a is divisible by 2
```

```
a = 6
if (a % 2 == 0):
    print("a is divisible by 2")
```

```
## a is divisible by 2
```

```
if (a % 3 == 0):
    print("a is divisible by 3")
```

```
## a is divisible by 3
```

```
if (a % 6 == 0):
    print("a is divisible by 6")
else:
    print("a is not divisible by 2, 3 or 6")
```

```
## a is divisible by 6
```

As you can see, in the first code snippet, only the first `True` conditional branch is executed, although `6` is obviously not only divisible by `2`, but also by `3` and `6`. In the second code snippet, however, all `True` conditions are executed consecutively. Why is that? Well, in the first code snippet we are dealing with one single `if statement` or conditional with several subclauses (the `elif` and `else` clauses), while the second code snippet strictly speaking consists of three individual `if statements`. These three conditionals are independent from each other and as such, are executed independently from each other. The first code snippet is read as "if `a` modulo 2 equals 0 (which is the same as saying 'if `a` is divisible by 2'), then print this or that; however, if (and only if) `a` is not divisible by 2, then look for a different `True` condition". In contrast, the second code snippet reads as "if `a` is divisible by 2, print this or that. Full stop. If `a` is divisible by 3, print this or that. Full stop. If `a` is divisible by 6, print this, else print that". In a chain of `if clauses` the individual clauses are disconnected, which is why they are read as a summation of individual clauses rather than one coherent statement.

You may ask yourself why this is included in the *Common errors* section. After all, using exclusively `if clauses` is not necessarily "wrong". The problem is that exclusively using `if clauses` may do the job at the beginning of your programming career, but it is a source for complex errors at a later stage. Remember to use chains of `if/elif/else clauses` whenever you want to need to make sure that only one `True` condition is executed at a time.

4. Another common error arises from using `tabs` and `spaces` interchangably to indent your code. To be honest, your Python compiler may not complain immediately when you mix `tabs` and `spaces`. However, there are good reasons why you do not want to do this anyway:

- Usually the human reader has even more difficulty than the computer discerning block structures when mixing `tabs` and `spaces`. This makes subsequent programming efforts a quite error-prone activity. For example, you are more likely to make indentation errors when you use `tabs` and `spaces` carelessly. And believe me, your Python compiler will throw an error at any of those!
- The semantics of tabs are not very well defined in the computer world. As a result, tabs may be displayed completely differently on different types of systems and editors. This means that you cannot safely transfer your code without risking to destroy your program's structure.
- More information on indentation in Python can be found here.

3.6 Debugging

In addition to some general good debugging practices summarized in the chapter on Debugging, there are some strategies that are especially useful when dealing with errors originating from conditionals in your program.

1. Again, checking your assignment `=` and equals `==` operators is a good starting point when you encounter an error that you suspect to originate from your conditionals. Using an assignment `=` operator where an equals `==` operator is needed will throw a syntax error and vice versa. While you are busy checking your assignment and equals operators, also check whether you forgot any colons `:` at the end of the first line of your if statements. This will also throw a syntax error.
2. Take a good look at the *conditions* in your if statements. You can check a variety of *conditions* interactively in your Python console outside of the program. Check whether the Boolean expressions in your conditions do indeed evaluate as you expect them to. Or would you immediately know what `True + 1 == False + 2` evaluates to? I still check what my conditions evaluate to when I am not a hundred percent sure. There a countless reasons why a condition may not evaluate as you intended. The variables you include in your condition may have unexpected values or subclauses of your condition may evaluate in a different order than you intended. You should take a look at how the variables you use in your conditions were assigned their respective values. If they are assigned unexpected values, you should look further upwards in the program's flow. Also check the precedence of your operators when you use several rather than a single one in your conditions. Use brackets `()` to enforce a certain operator order, if necessary.
3. You should also consider simplifying your conditionals. The more complex your conditionals, the more prone you are to making mistakes. Simplifying deeply nested conditionals can solve programming errors that are otherwise difficult to track down.

3.7 Exercises

3.7.1 Exercise 1. Boolean logic

What is the output of the following code snippets?

```
a = True
b = False
print(a and b)
```

```
a = True
b = False
print(not a and b)
```

```
a = True
b = False
print((not a) or b)
```

```
a = True
b = False
print(not (a and b))
```

```
a = True
b = False
print(not (a or b))
```

```
a = 1
b = 2
c = 4
print(a >= b and c > a + b)
```

```
a = 1
b = 2
c = 3
print(b % a >= c - (a + b))
```

3.7.2 Exercise 2. State charts

A state chart visualizes the interaction of a program. Which of the code snippets matches the state chart?

```
# Variables
STATE = "A"
number_of_trials = 0
while True:
  #ITC
  for event in pygame.event.get():
      if STATE == "A":
          number_of_trials = number_of_trials + 1
          if event.type==KEYDOWN and event.key == K_SPACE:
              STATE = "C"
```

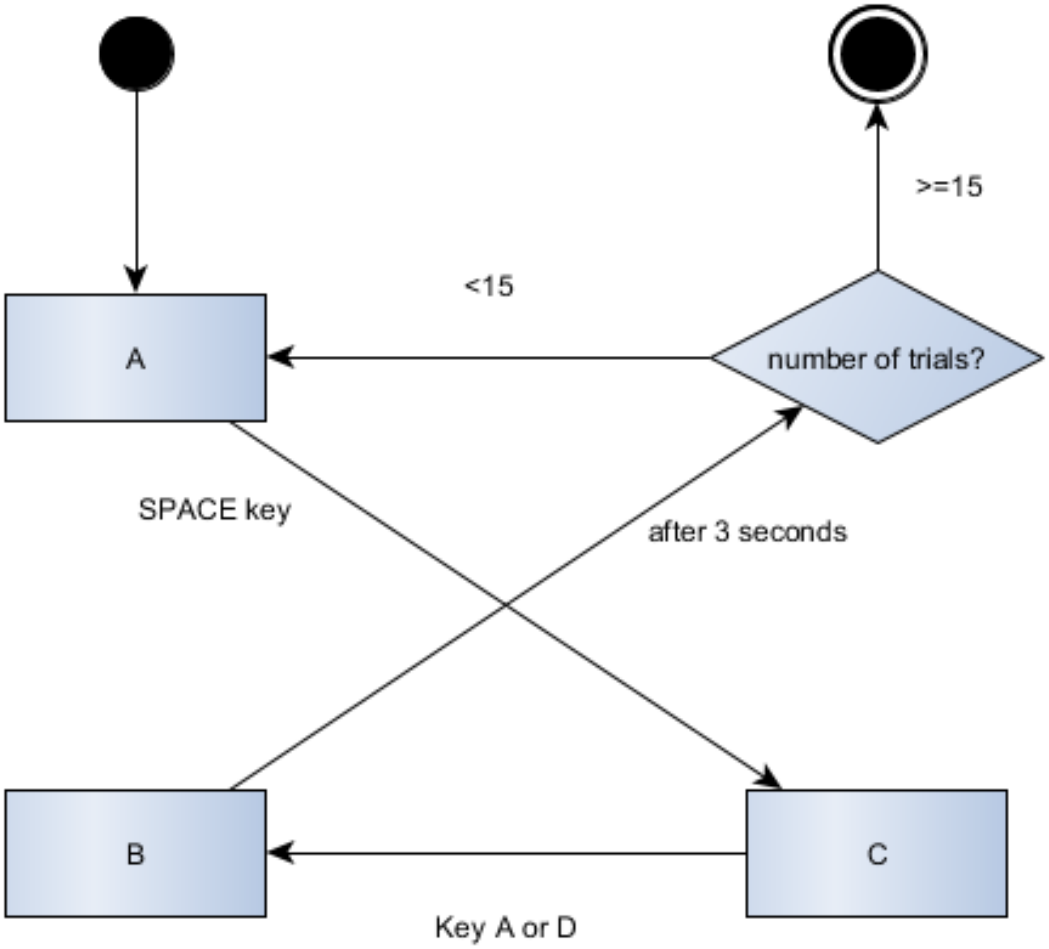


Figure 3.2:

```
        elif STATE == "C":
            if event.type==KEYDOWN and event.key == K_A or event.key == K_D:
                arrival_at_B = time()
                STATE = "B"

    #ATC
    if STATE == "B":
        if time() - arrival_at_B > 3:
            STATE = "number_of_trials"
    elif STATE == "number_of_trials":
        if number_of_trials < 15:
            STATE = "A"
        else:
            pygame.quit()
            sys.exit()
```

```
# Variables
STATE = "A"
number_of_trials = 0
while True:
    #ITC
    for event in pygame.event.get():
        if STATE == "A":
            number_of_trials = number_of_trials + 1
            if event.type==KEYDOWN and event.key == K_SPACE:
                STATE = "C"
        elif STATE == "C":
            if event.type==KEYDOWN and event.key == K_A and event.key == K_D:
                arrival_at_B = time()
                STATE = "B"
            else:
                arrival_at_B = time()
                STATE = "B"

    #ATC
    if STATE == "B":
        if time() - arrival_at_B > 3:
            if number_of_trials < 15:
                STATE = "A"
            else:
                pygame.quit()
                sys.exit()
```

```
# Variables
STATE = "A"
number_of_trials = 0
while True:
  #ITC
  for event in pygame.event.get():
      if STATE == "A":
          number_of_trials = number_of_trials + 1
          if event.type==KEYDOWN and event.key == K_SPACE:
              STATE = "C"
      elif STATE == "C":
          if event.type==KEYDOWN and event.key == K_A or event.key == K_D:
              arrival_at_B = time()
              STATE = "B"

  #ATC
  if STATE == "B":
      if time() - arrival_at_B > 3:
          if number_of_trials < 15:
              STATE = "A"
          else:
              pygame.quit()
              sys.exit()
```

3.7.3 Exercise 3. Transition tables

For the following code snippet, make a transition table.

```
STATE = "A"
while True:
    #ITC
    for event in pygame.event.get():
        if STATE == "A":
            count = 0
            if event.type == KEYDOWN and event.key == K_w:
                STATE = "B"
                print(STATE)
        elif STATE == "B":
            count = count + 1
            if event.type == KEYDOWN and event.key == K_a:
                timer = time()
                STATE = "C"
```

```
                print(STATE)
        elif STATE == "D":
            if event.type == KEYDOWN and event.key == K_SPACE:
                if count < 10:
                    STATE = "B"
                else:
                    STATE = "quit"
                print(STATE)
    #ATC
    if STATE == "C":
        present_picture()
        if time() - timer > 3:
            STATE = "D"
        print(STATE)
    elif STATE == "quit":
        pygame.quit()
        sys.exit()
```

3.7.4 Exercise 4. Mini programs

What is the output of the following mini programs?

```
x = 5
y = 0

if x >= 5 and y != False:
    print(y/x)
```

- 0
- Nothing
- A `TypeError`, numerical and Boolean values cannot be compared

```
x = 12

if x%3 == 0:
    print("x is divisible by 3")
elif x%4 == 0:
    print("x is divisible by 4")
```

- `x is divisible by 4`
- `x is divisible by 3`
  `x is divisible by 4`

- `x is divisible by 3`

```
x = 2
y = 0

if y + 1 == x or not x * 1 <= y:
   print(x + y)
```

- A `SyntaxError`, you need to use brackets if you mix different operator types in one condition
- Nothing
- 2

3.7.5 Exercise 5. Following the control flow

Read the following program code and try to follow its control flow. Write down the content of each `print statement`. Check your answers with the help of your python editor. *Hint*: spaces also add to the length of a string (as measured by the `len()` *method*)!

```
myString = "Hello, World!"
if len(myString) >= 13 or "ello" in myString:
    myString = "Hi, programming aspirant!"
elif len(myString) <= 12 and "ello" in myString:
    myString = "Hello, from the other side."
print myString
if "Hi" in myString:
    if len(myString) < 25:
        myString = "Wow, my computer seems to answer!"
    elif len(myString) > 25:
        myString = myString + " -- Your computer"
    else:
        myString = myString + " How are you?"
else:
    if len(myString) <= 29:
        myString = "How are you, my computer?"
    elif len(myString) == 27 and "Hello" in myString:
        myString = myString + " I must have called a thousand times."
    else:
        myString = myString + " -- Adele"
print myString
if "computer" in myString or len(myString) == 38 or "HI" in myString:
    myString = myString + " I myself am doing fine."
```

```
else:
    myString = myString + " I am doing just fine."
print myString
```

3.7.6 Exercise 6. Indentation

Copy the following piece of code and run it. Correct all indentation errors until the code runs error-free.

```
a = 4
if a < 20 and a > 0:
    if a > 0 and a < 15:
    a = a + 3
    else:
       a = a - 3
        print "This is a dead end statement."
elif a > 20:
print "This should only be printed if a exceeds 20!"
else:
    a = 17
```

3.7.7 Exercise 7. Pseudo code conditionals

Translate the following common language sentences into *pseudo code* conditionals as in

- If it rains tomorrow, I will take an umbrella with me.

```
if rain
    set umbrella to True
```

Pseudo code is a programming language-independent informal and simplified description of the essential operating principles of a computer program. It adheres to the structural conventions of a normal programming language, but it is rather meant for the human reader than machine reading. Pseudo code is commonly used for sketching out the structure of a computer program before the actual code writing takes place. You can find a more detailed description of what pseudo code is and some examples of written pseudo code here.

1. If there is no *rain* tomorrow, I will either go *swimming*, if Jan has time to join that is, or I will go *hiking*. Otherwise I will stay at home and *read*.
2. Under the condition that the participant is *18 years or older* and has *no history of alcohol abuse*, he or she *may participate* in our study.

3. When the streets are *wet*, the chance is high that it has *rained*.
4. If the participant turns out to be *eligible* to participate *brief* him or her about the experiment. Otherwise kindly *explain* that they do not meet the criteria to participate. If the eligible participant is assigned to *condition A*, lead them to *room 001*. If he or she is assigned to *condition B*, lead them to room *002*.

3.7.8 Exercise 8. Modify Stroop Task

Add a conditional to the interactive Stroop experiment so that in case the participant's reaction time exceeds 5 milliseconds, "Come on, you can be faster!" is printed to the screen

Hints:

- First think about where in the program you need to make changes to print the message
- Formulate the `if statement` or mutate existing `if statements` to fit your needs
- Use the existing program code as a template for drawing the message

3.7.9 Exercise 9. Simplify nested conditionals

For a research project, you want to examine to what extent gender affects the influence of coffee on a person's performance on a vigilance task. You perform matching during sampling. All females will be grouped together, and all males will form one condition.

The code below represents a decision tree that matches a candidate participant on the basis of their personal details with either condition A (for females) or condition B (for males). In addition, the decision tree raises a warning if the candidate does not meet at least one of the requirements for participation.

As is, the code is quite complex, including a lot of redundant lines and unneccessary nested conditionals. First, read the code carefully to figure out the requirements a candidate participant has to meet in order to participate in the study. Then, simplify the code while maintaining its functionality.

```
### participant details ###
age = 20
gender = "Male"
study = "Psychology"
speaks_Dutch = True
coffee = True
condition = "not eligible for the experiment"
if age >= 18:
```

```
    if gender == "Female":
        if study == "Communication Sciences":
            if speaks_Dutch == True:
                if coffee == True:
                    condition = "A"
                elif coffee == False:
                    print "Participant is not eligible to take part in the experime
            elif speaks_Dutch == False:
                print "Participant is not eligible to take part in the experiment."

        elif study == "Psychology":
            if speaks_Dutch == True:
                if coffee == True:
                    condition = "A"
                elif coffee == False:
                    print "Participant is not eligible to take part in the experime
            elif speaks_Dutch == False:
                print "Participant is not eligible to take part in the experiment."

        elif study != "Communication Sciences" and study != "Psychology":
             print "Participant is not eligible to take part in the experiment."

    elif gender == "Male":
        if study == "Communication Sciences":
            if speaks_Dutch == True:
                if coffee == True:
                    condition = "B"
                elif coffee == False:
                    print "Participant is not eligible to take part in the experime
            elif speaks_Dutch == False:
                print "Participant is not eligible to take part in the experiment."

        elif study == "Psychology":
            if speaks_Dutch == True:
                if coffee == True:
                    condition = "B"
                elif coffee == False:
                    print "Participant is not eligible to take part in the experime
            elif speaks_Dutch == False:
                print "Participant is not eligible to take part in the experiment."

        elif study != "Communication Sciences" and study != "Psychology":
            print "Participant is not eligible to take part in the experiment."
```

```
else:
    print "Participant is not eligible to take part in the experiment."
```

3.7.10 Exercise 10. Flow chart conditionals

Turn the following flow chart into a python script and run it. Given that the initial value of `n` is `-1`, what is the final value of `n`?

3.8 Think Further

- For a complete list of all operator precedences in Python feel free to consult the following summary: Official Python Documentation: 5.15 Operator precedence
- For an alternative explanation of the contents of this chapter, consult A Byte of Python: Chapter 6. Operators and Expressions and Chapter 7. Control Flow
- For more detail on Conditionals, Boolean algebra and related topics, you are invited to take a look at How to Think Like a Computer Scientist: Learning with Python 3: 5. Conditionals

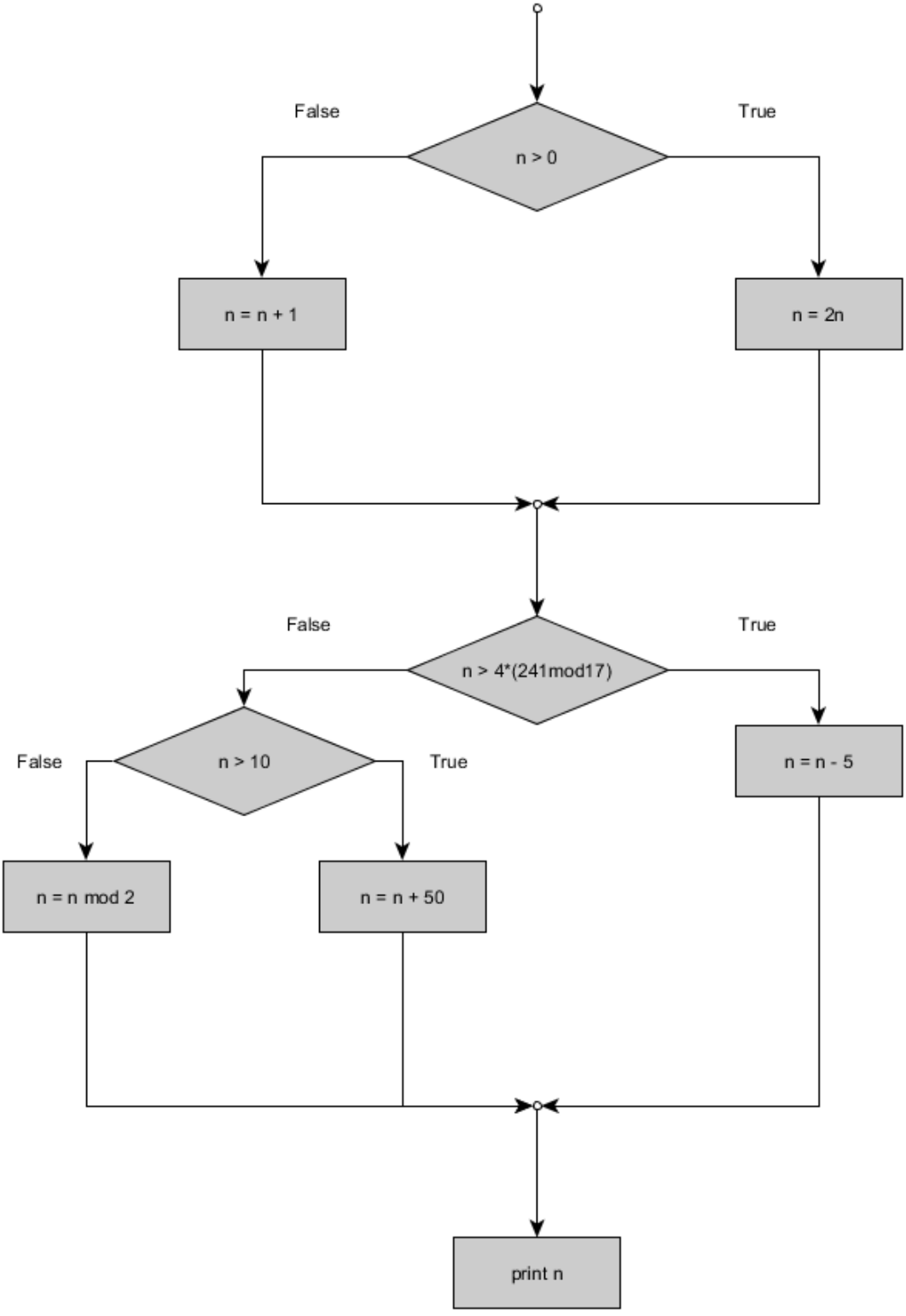

False
n > 0
True
n = n + 1
n = 2n
False
n > 4*(241mod17)
True
False
n > 10
True
n = n - 5
n = n mod 2
n = n + 50
print n

Figure 3.3:

4 Lists

```
message = ["Hello,"]
message.append("world")
print(" ".join(message))
```

```
## Hello, world
```

In chapter 2 Variables, you encountered the four basic data types of the Python programming language: *integers*, *floats*, *Strings* and *Booleans*. You also learned that apart from these four basic data types, Python knows numerous specialized types. Now it is time to encounter one group of these specialized data types: *collections*. You will learn about three types of collections, *lists*, *tuples*, and *dictionaries* in this chapter. Together, they make up a toolbox for storing information in a computer program.

4.1 Lists

```
a = [10, 5, 15, 25]
b = ["eggs", "bacon", "cheese"]
c = [4, "hello", 7.3, False]
```

Lists are one type of collection. The easiest way of creating a list is surrounding some values with cornered brackets `[]` and separating the values of the list with commas `,`. The values that make up a list are called *items* or *elements*. Lists can contain any type of value or variable. From this it follows that lists can also contain other lists! A list that contains another list is called a *list of lists*, or *nested* list (you may also think of this phenomenon as *listception*, if you ever watched Christopher Nolan's *Inception* (2010)).

```
a = 2
b = [a, 2*a]
c = [b, 6]
print(c)
```

```
## [[2, 4], 6]
```

A special type of list is the so-called *empty list*. An empty list is a list with no elements. Empty lists are denoted with `[]`.

```
a = []
```

In a sense, lists are similar to *Strings*. Strings are basically ordered collections of *characters*. For example, the word `Hello` is an ordered collection of five characters: `H`,`e`, `l`, `l`, and `o`. In contrast to Strings, however, the elements of lists can be of *any* type. Ordered collections such as strings and lists are also called *sequences*.

Lists are *ordered* collections of values. However, a list being *ordered* has nothing to do with how humans may be inclinded to order the elements of a list. For example, you may want to order a list `a` with three elements `"A"`, `"B"`, and `"C"` according to their alphabetic order: `a = ["A", "B", "C"]`. However, for lists, the value of an element is irrelevant for its position in the list. Lists are **not** *ordered* because they grasp some underlying semantics about their elements according to which they come up with an order. In fact, element values are arbitrary for a list. Instead, lists are called *ordered* because they allocate a certain position in memory to each element. Each list element occupies a unique position in memory. Elements of a list are accessed by querying the list at specified positions, so-called *indices*. Accessing a list at a specified index, the value located at that position in memory is returned. The *index operator* `[]` is used to access list elements at specified indices, like so

```
a = [1, 2, 3]
print(a[0])
```

```
## 1
```

```
print(a[1])
```

```
## 2
```

The *expression* inside the brackets of the index operator (in this case the integers `0` and `1`) specifies the *index* of the to-be selected element. Each element inside a list has a unique index, an *integer* describing the element's position in the list. One major cause of errors surrounding lists is that indices start at `0`, not at `1`, as you might expect. After all, to us it feels natural to start counting at `1`. As you can see below, this means that the first element of a list is accessed by indexing `0`.

```
a = ["Hello", "World", "!"]
print(a[0])
```

```
## Hello
```

```
print(a[2])
```

```
## !
```

Why do indices start at 0, instead of 1? As you might already expect, it has to do with how computers store information in memory. A list serves as a pointer, a reference to a memory location. The expression `list[i]` then refers to a memory position i-steps away from the starting element. This means that the index is used as an offset. The first element of a list is located at exactly the position the list refers to (which means that the offset is 0). Hence, the first element of a list is located at index 0.

4.1.1 Lists are mutable

In contrast to Strings, lists are *mutable* sequences. This means that even after a variable is assigned a list as value, specific elements of this list can be changed without reassigning a completely new list! You can use the *index operator* to mutate specific list items.

```
shoppinglist = ["appels", "bananas", "tomatoes"]
shoppinglist[2] = "peaches"
print("Updated shopping list: " + ", ".join(shoppinglist))
```

```
## Updated shopping list: appels, bananas, peaches
```

```
a = [1,2]
a[0] = 10
print(a)
```

```
## [10, 2]
```

Using the index operator, you can also access (and change!) several elements at a time using so-called *list slices*:

```
a = [1, "a", 2, "b", 3, "c"]
print(a[1:4])
```

```
## ['a', 2, 'b']
```

```
print(a[0:2])
```

```
## [1, 'a']
```

```
a[0:2] = "d"
print(a)
```

```
## ['d', 2, 'b', 3, 'c']
```

The indices around the `:` colon indicate the starting and ending index of the selected list slice. Note that a list slice will always include the element located at the starting index, but exclude the element located at the ending index. To include `2` in the second list slice you would thus need to access `a[0:3]`, or simplified `a[:3]`. The latter meaning "every element up until, but excluding index 3".

Note that the statement `a[0:2] = d` mutates both elements, `1` *and* `"a"` into a single new item `"d"`. This means that the statement did not only change some elements of `a`, but also `a`'s total number of elements!

When no starting or ending index is specified, the `:` expression selects all elements of a list and can be used for *cloning* lists.

```
a = [1, "a", 2, "b", 3, "c"]
aCopy = a[:]
print(aCopy)
```

```
## [1, 'a', 2, 'b', 3, 'c']
```

But watch out, you **cannot** use the index operator to add new elements to a list. To do so, you would need to access a new index, an index not yet claimed by an existing list element. Trying to access an index outside of the current range of indices will produce an index out of range error.

```
a = [1,2]
a[2] = 10
print(a)
```

```
IndexError: list assignment index out of range
```

Luckily, there are other ways of adding elements to a list.

```
a = []
a.append(1)
print(a)
```

```
## [1]
```

```
a.append(2)
print(a)
```

```
## [1, 2]
```

The *append* method adds the specified element at the end of a given list. You can use the *extend* method to add *multiple* values of another list (or other type of collection) to your list.

```
a = [1,2]
b = [3,4]
a.extend(b)
print(a)
```

```
## [1, 2, 3, 4]
```

Note that there is a difference between appending a list and extending your list with another list.

```
a = [1,2]
b = [3,4]
a.append(b)
print(a)
```

```
## [1, 2, [3, 4]]
```

In the above example, **extending** `a` with `b` causes the elements stored in `b` to be appended to `a` sequentially, each of them constituting a new element in `a`. In contrast, the whole list `b` is regarded as an element of `a` when `b` is **appended** to `a`. Because lists are mutable, you can sequentially fill them with elements, one by one. This is very handy for storing values you generate throughout your program.

4.1.2 A word about class specific functions

By now, you have seen a few examples of class-specific *functions*, such as `append()` for lists or `join()` for strings. Even though a detailed explanation of *functions* (also called *methods*) is postponed to Chapter 6, a quick introduction to the terminology of objects, classes, and class-specific functions will help you make the most of your lists and other collections. For now, you can see functions as pieces of reusable code that specify what to do with some input values.

When you create a variable `a` and assign a value to it, say `[4,5]`, you can think of it as creating an *object* (i.e. an instance) of the Python *class list*. The *list* class has been programmed by the developers of the Python language, same as any other built-in Python class, such as integers and strings. Classes are blueprints, a specification of how to build certain objects. Engineers use blueprints to build cars, buildings, bridges and more. Chefs use recipes to put together meals. Python classes are the same in that they specify how objects of a certain type are programmed internally when you create one of them. A class has built-in *methods* that can be used by *any* object that has been created using the class' blueprint. And this is what you basically do whenever you surround a number of values with cornered brackets. You create a list according to the blueprint of the Python class `list`. It is very useful that your list inherits *functions* from its blueprint. It means that without any effort on your part your lists have a variety of functionality at their disposal. They can immediately use a range of built-in methods such as the `append()` method.

```
a = [4,5]
a.append(6)
print(a)
```

```
## [4, 5, 6]
```

The `append()` method adds the value specified between the brackets to the end of the list. Typing `help(list)` in your console will give you a complete list of built-in functions of the class `list` and what they do.

What is the take-home message of this excursion to object-orientation? Well, make use of built-in class functions as much as you can! They provide a tremendous amount of functionality which you are likely going to need and want, written in neat code which makes them very efficient to use, too!

The interested reader can find more information on Python classes here.

4.2 Tuples

Much like lists, *tuples* are ordered collections of values, but with less exentensive functionality and their unique characteristic being that they are *immutable*. Once you created

a *tuple* in memory, you cannot change its elements.

```
thorndike = ("Edward", "Lee", "Thorndike", 1874, "Law of effect", "Law of exercise")
thorndike[4] = "Law of recency"
```

```
TypeError: 'tuple' object does not support item assignment
```

Tuples are usually used when you assume that a collection of values will not change in the course of your program. They are *records* of related information, chunked together so that we can use them as a single entity. In the Stroop Task program for instance, we saw that tuples were used to chunk together RGB combinations that make up a color. The Pygame module used these tuples to generate color in the Stroop Task.

Tuples are immutable. However, if you really need to change the contents of an already existing tuple, you can always assign a new value to the variable which holds the tuple as a value.

```
thorndike = ("Edward", "Lee", "Thorndike", 1874, "Law of effect", "Law of exercise")
thorndike = thorndike[:] + ("Law of recency",)
print(thorndike)
```

```
## ('Edward', 'Lee', 'Thorndike', 1874, 'Law of effect', 'Law of exercise', 'Law of recency')
```

Note the comma `,` in the second assignment statement `("Law of recency,")`. Without the comma at the end of the parentheses, Python treats the content of a one-element tuple such as `("Law of recency",)` as belonging to the data type of the element (in this case a `str`). As a result, a *type error* is thrown when you try to concatenate the `thorndike` tuple and `("Law of recency")`.

```
thorndike = thorndike[:] + ("Law of recency",)
```

```
TypeError: can only concatenate tuple (not "str") to tuple
```

Python has a very powerful *tuple assignment* feature that allows a tuple of variable names on the left of the assignment operator `=` to be assigned to a tuple of values, like so

```
thorndike = ("Edward", "Lee", "Thorndike", 1874)
(name, middle_name, surname, born) = thorndike
print(name)
```

```
## Edward
```

```
a, b = "Hello", "World"
print(a)
```

```
## Hello
```

The brackets `()` are optional in a tuple assignment. Using tuple assignment, you can easily perform several variable assignments in just one line!

4.3 Dictionaries

Dictonaries share similarities with real-life address books where you can find a person's contact details by looking up his or her name.

```
address_book = {"studyadviser-PSY": "studyadviser-psy[at]utwente.nl",
                "studentservices" : "studentservices[at]utwente.nl",
                "ICT-helpdesk"    : "servicedesk-ict[at]utwente.nl"
                }
print(address_book["studyadviser-PSY"])
```

```
## studyadviser-psy[at]utwente.nl
```

Dictionaries associate *keys* (e.g. a name) with *values* (e.g. contact details). The `key:value` pairs of a dictionary are separated by commas. Each pair contains a *key* and a *value* separated by a colon `:`. `key:value` pairs are always one-to-one mappings. This means that you cannot map the same key to several values, like so

```
address_book = {"studyadviser-PSY": "studyadviser-psy[at]utwente.nl",
                "studyadviser-PSY": "Cubicus, room C116"
                }

print(address_book["studyadviser-PSY"])
```

```
## Cubicus, room C116
```

```
print("Number of key:value pairs in address_book: %d" % len(address_book.keys()))
```

```
## Number of key:value pairs in address_book: 1
```

As you can see, the first mapping of the key `"studyadviser-PSY"` to `"studyadviser-psy[at]utwente.nl"` has been overwritten by the second mapping of the key to `"Cubicus, room C116"`. In total, `address_book` now only contains a single `key:value` pair. As with lists and tuples, values are arbitrary for dictionaries. This means that while you cannot map the same key to several values, you can map the same value to multiple keys.

```
address_book = {"studyadviser-PSY": "studyadviser-psy[at]utwente.nl",
                "studyadviser-PSY_email": "studyadviser-psy[at]utwente.nl"
                }

print("Number of key:value pairs in address_book: %d" % len(address_book.keys()))
```

```
## Number of key:value pairs in address_book: 2
```

Dictionaries require their keys to be *immutable* and *unique*. As you above, you cannot have two keys named `"studyadviser-PSY"` in the `address_book`. This also means that certain data types are unsuitable as dictionary keys, like lists.

```
d = {['a']: 'b'}
```

```
TypeError: unhashable type: 'list'
```

Whenever you try to use a mutable object (like lists) as a key, a `TypeError` will be thrown. Because lists are mutable (you can update their elements), they are *unhashable*. Without going into the details of hash functions here, a *hash* is another representation of an object. It is a transformation of the object's length sequence of bytes into a fixed length sequence. *Hash functions* are used in cryptographic algorithms, in digital signatures, manipulation detection, password storage and more. Hash functions are constructed so that calculating an object's hash representation is simple, but retrieving the original object is difficult without knowing the function used to hash the object in the first place. You can imagine that things start to get messy if the object itself can change after its hash has been calculated. A hash is a momentary snapshot of an object. It is impossible to retrieve the "latest version" of a changable object from its hash representation if, even if you know its hash function. Dictionaries themselves are also mutable (and thus, unhashable) and cannot be used as *keys*, by the way.

Dictionaries are *unordered* collections. Unlike sequences, such as lists and tuples, the elements of a dictionary cannot be accessed using indices. Dictionaries do not have a sequential representation of their elements. Values stored in a dictionary can be accessed and changed through their respective *keys*, like so

```
address_book = {"studyadviser-PSY": "studyadviser-psy[at]utwente.nl",
                "studentservices" : "studentservices[at]utwente.nl",
                "ICT-helpdesk"    : "servicedesk-ict[at]utwente.nl"
                }

address_book["studentservices"] = ("studentservices[at]utwente.nl", "0534892124")
print("The telephone number of the student services is", str(address_book["students
```

```
## ('The telephone number of the student services is', '0534892124')
```

Apart from updating existing dictionary entries, you can obviously also add and remove entries of a dictionary. Use the `del` statement to remove dictionary entries.

```
address_book = {"studyadviser-PSY": "studyadviser-psy@utwente.nl ",
                "studentservices" : "studentservices@utwente.nl",
                "ICT-helpdesk"    : "servicedesk-ict@utwente.nl"
                }

address_book["studentUnion"] = "studentunion@union.utwente.nl"
print(address_book)
```

```
## {'studentservices': 'studentservices@utwente.nl', 'studyadviser-PSY': 'studyadvi
```

```
del address_book["ICT-helpdesk"]
print(address_book)
```

```
## {'studentservices': 'studentservices@utwente.nl', 'studyadviser-PSY': 'studyadvi
```

4.4 Putting collections to use

Lists, tuples and dictionaries are readily used in interactive programs, such as the Stroop Task program. Collections help structure user input and the subsequent storage of user input in computer memory. Structuring user input makes later retrieval of information easier. Below you see a code snippet from the Stroop Task program.

```
KEYS     = {"red": K_b,
            "green": K_n,
            "blue": K_m}
RT = []
```

```
trial_number = 0

while True:
    pygame.display.get_surface().fill(BACKGR_COL)
    # Changing states
    for event in pygame.event.get():
        if STATE == "welcome":
            if event.type == KEYDOWN and event.key == K_SPACE:
                STATE = "present_trial"
                print(STATE)

        if STATE == "present_trial":
            trial_number = trial_number + 1
            this_word  = pick_color()
            this_color = pick_color()
            time_when_presented = time()
            STATE = "wait_for_response"
            print(STATE)

        if STATE == "wait_for_response":
            if event.type == KEYDOWN and event.key in KEYS.values():
                time_when_reacted = time()
                this_reaction_time = time_when_reacted - time_when_presented
                RT.append(this_reaction_time)
                this_correctness = (event.key == KEYS[this_color])
                STATE = "feedback"
                print(STATE)

        if STATE == "feedback":
            if event.type == KEYDOWN and event.key == K_SPACE:
                if trial_number < 20:
                    STATE = "present_trial"
                else:
                    STATE = "goodbye"
                print(STATE)

        if event.type == QUIT:
            STATE = "quit"
```

Note how the variable `RT` is used to keep track of the user's reaction times throughout the trials. The variable `this_reaction_time` is overwritten in each trial. This means, that at the end of the program, when all trials are completed, `this_reaction_time` only contains the last value assigned to it during the 20th trial. All other reaction times

were long erased from computer memory. But wait, what if we want to calculate a user's mean reaction time once the experiment is finished? In order to do so, we need some way of saving the reaction times from trial to trial. They need to persist until the last trial is completed. This is exactly where collections step into the picture. Collections are capable of storing multiple values at the same time and they can be mutated iteratively (one reaction time per trial). Hence, we `append` the calculated reaction time to a list in each trial.

In the end, you as a programmer still need to decide how to proceed with the collected reaction times. Do you want to print their mean to the screen, export them out of the program, or combine them with data gathered from other experimental subjects? It is up to you. Using collections, at least you have a user's reaction times at your disposal, even after they completed the experiment.

4.5 Common errors

- The by far most occurring errors involving lists are `index out of range` errors. I think it is fair to say that they make up a large amount of all programming errors. An `index out of range error` is thrown when you try to access an index that does not exist in a sequence.

```
shoppinglist = ["milk", "eggs", "bread", "appels"]
print shoppinglist[4]

IndexError: list index out of range
```

You will usually encounter this error as a result of confusing the starting index `0` with `1`. Against all intuition, in a list of four elements, the last element is stored at index `3`, not at index `4`. It helps to keep in mind that the indices of your sequences always have a range one shorter than the number of elements in your list. For a list with four elements, the positive index range is `0` to 4 minus 1, thus `3`.

```
shoppinglist = ["milk", "eggs", "bread", "appels"]
print(shoppinglist[3])

## appels
```

- Another common mistake surrounding collections concerns the immutability of *tuples*. In particular, trying to change the content of a *tuple* will result in a `TypeError`. Tuples are immutable, meaning that you cannot change their content after their initialization. You can, however, re-assign the variable name of an existing tuple to a new tuple containing different values.

```
a = (1,2,4)
b[2] = 3

TypeError: 'tuple' object does not support item assignment
```

```
a = (1,2,4)
print(a)


## (1, 2, 4)
```

```
b = (1,2,3)
print(b)


## (1, 2, 3)
```

- Concerning dictionaries, you will most probably not escape one or another `KeyError`. A `KeyError` is thrown when you try to access a `key` that does not exist in the dictionary.

```
d = {}
print(d['a'])

KeyError: 'a'
```

The same is likely to occur when you try to use indices to access the entries of a dictionary

```
d = {'a': 1}
print(d[0])

KeyError: 0
```

Dictionaries are the only *unordered* type of collection you encountered in this chapter. It occurs now and then that one assumes that a dictionary can be treated as other common types of collections, such as lists or tuples.

4.6 Debugging

Here are some tips and tricks on how to debug errors that potentially originate from your collections.

1. Regularly checking the `length` of your collections can be very helpful to confirm that your code does indeed what you expect it to do. Take the example of `RT`, the list we made to keep track of a participant's reaction times throughout the Stroop Task program. At the end of the experiment, you would expect the list to contain 20 elements, given that `RT` was initialized as an empty list and that the experiment had 20 trials. By checking the `length` of `RT`, you can immediately see whether your code indeed stored one reaction time per trial. `RT` should have a `length` of 20 after the experiment. You can check the length of a collection using the `len()` method, like so

```
a = [3, 4]
print(len(a))
```

```
## 2
```

2. Dictionaries have a very useful built-in method that represents their `keys` as a list.

```
d = {'a': 1,
     'b': 2}

print(d.keys())
```

```
## ['a', 'b']
```

This comes in handy when you keep getting a `KeyError` while you are convinced that your dictionary should contain a certain key.

```
d = {'a ': 1,
     'b': 2}

print(d['a'])
KeyError: 'a'
```

You can use `in` to check whether a candidate `key` is among the keys of the dictionary

```
d = {'a ': 1,
     'b': 2}

print('a' in d.keys())
```

```
## False
```

In this case, a typing error is the culprit. Instead of `'a'`, `'a '` with a whitespace was set as the first key of `d`.

Alternatively, you can use a simple conditional constructions and `not in` to make sure that you do not overwrite any existing keys

```
d = {'a': 1,
     'b': 2}
if 'a' not in d.keys():
    d['a'] = 0
print(d)
```

```
## {'a': 1, 'b': 2}
```

4.7 Exercises

4.7.1 Exercise 1. Indexing

What is the output of the following code snippets?

```
a = [6, 7]
b = [4, 5]
b.append(a)
print(b)
```

```
d = {1: 'a',
    'a': 1}

print(d[1])
```

```
a = [1]
b = ('a', 2)
a.extend(b)
print(a[1])
```

```
a, b = (['a', 'd'], ['b', 'c'])
print(b[0])
```

```
a = [1, 2, 3]
print(a[-1])
```

```
a = (1, 2, [3, 4], 5)
print(a[2][1])
```

4.7.2 Exercise 2. Debugging

Will the following code snippets throw an error? If anything goes wrong, why does it go wrong?

```
a, b = 'Sigmund', 'Freud', (1856, 1939)
```

- No error is thrown, `a` and `b`are assigned the values `'Sigmund'` and `'Freud'` respectively, and `(1856, 1939)` will not persist in computer memory.
- A `ValueError` is thrown. There are too many values on the right side of the equals sign `=` to unpack.
- A `NameError` is thrown. The variables `a` and `b` have not been assigned yet and can therefore not be compared with other values.

```
a = [1, 2]
b = (3, 4)

a.append(b)
print(a[3])
```

- Nothing goes wrong, the output is `4` because `4` is located at index `3` after `b` is appended to `a`.
- A `TypeError` is thrown. `b` cannot be appended to `a` because lists cannot have tuples as elements.
- An `IndexError` is thrown. After appending `b`, `a` has three elements. `3` is not a valid index.

```
a = ('a', [1, 2])

a[1].append(3)
```

- A `TypeError` is thrown because tuples are immutable and do not support item assignment.
- No error is thrown, `3` is appended to `[1, 2]`.
- An `AttributeError` is thrown, tuples do not have an `append` method.

```
a = {('a', 'b'): 3}

print(a.keys())
```

- No error is thrown, the list of keys contains one element, `('a', 'b')`.
- A `TypeError` is thrown. Tuples cannot be *hashed* because they are changable and therefore unsuitable as dictionary keys.
- No error is thrown. By means of *tuple assignment*, the dictionary has two keys, `a` and `b`.

4.7.3 Exercise 3. Mini programs

What is the output of the following mini programs?

```
a = {}

print(a.keys())
```

- `[]`
- `0`
- A `KeyError` because `a` is empty and has no keys.

```
a = []
b = ([1, 2], [3, 4], [5, 6])

a.extend(b)
print(a[1])
```

- `[1, 2]`
- `2`
- `[3, 4]`

```
a = ('a', ('b', 'c'))

print('b' in a)
```

- `True`
- `False`
- A `TypeError` is thrown because the `in` operator cannot be used with tuples.

```
a = [{'a': 1 }, {'a': 2}]

if 'a' in a[0]:
    print(a[0]['a'])
elif 'a' in a[1]:
    print(a[1]['a'])
```

- 1
- 2
- 1 and 2

4.7.4 Exercise 4. Shopping list

```
shoppinglist = ["bread","cheese","jam","milk","oatmeal","blueberries","oranges","ap
shoppinglist[0] = "buns"
shoppinglist = shoppinglist + ["chocolate"]
del shoppinglist[2]
shoppinglist_mum = ["eggs","yoghurt","potatoes","salmon"]
shoppinglist += shoppinglist_mum
del shoppinglist[10]
ingredients_muffins = ["icing sugar","whipping cream","lemons","flower","vanilla su
shoppinglist.append(ingredients_muffins)
del shoppinglist[11][6]
shoppinglist[8:11] = [shoppinglist[8:11]]
del ingredients_muffins[7]
```

The above shopping list has been subject to several changes since it has been written. Standing in the supermarket, you want to know the contents of the final shopping list. Which groceries do you have to buy?

You can check your answer afterwards by running the code.

4.7.5 Exercise 5. Dictionaries

```
famous_psychologists = {"Freud":("Sigmund", "Freud", (1856,1939), "Psychoanalysis")
                        "Piaget":("Jean", "Piaget", (1896,1980), "Theory of Cogniti
                        "Skinner":(("Burrhus", "Frederic"), "Skinner", (1904,1990),
                        }

erikson = ("Erik", "Erikson", (1902,1994), "Theory of Psychological Development")
famous_psychologists["Erikson"] = erikson
more_on_Skinner = {"Schedules of Reinforcement":("positive reinforcement", "negativ
                   "Skinner Box":("pigeon","lever","food"),
                  }
famous_psychologists["Skinner"] = famous_psychologists["Skinner"][:3] + ("Behaviori
```

Using the given dictionary on famous psychologists and the above changes made to the dictionary:

- Try to understand the changes made to the original dictionary by writing down the content of the following print statements

```
print(famous_psychologists["Erikson"][:3])
print(famous_psychologists["Piaget"][3:])
print(famous_psychologists["Freud"][2][0])
print(famous_psychologists["Skinner"][0][1])
print(famous_psychologists["Skinner"][5].keys()[1])
word = famous_psychologists["Skinner"][5].keys()[0]
print(famous_psychologists["Skinner"][5][word][0])
```

- Afterwards, you may check your answers by running the code. Pay attention that your own answers match exactly.

4.7.6 Exercise 6. Element selection

For the following data structures, give the code by which you can access the specified element.

```
dinner = ["minced beef", "tomatoes", "pasta", "onions", "pasta sauce", "courgette"]
shoppingList = [dinner, "bread", "milk", "cornflakes", "cheese"]
```

```
grades = {"p1": ("Judith",21,"female","B-PSY",(5.6,7.7,7.3)),
          "p2": ("Menno",19,"male","B-CREATE",(8.6,6.9,7.4)),
          "p3": ("Jan",19,"male","B-CREATE",(3.9,4.5,8.9)),
          "p4": ("Eva",20,"female","B-MATH",(8.0,7.0,9.1)),
          "p5": ("Sophia",22,"female","M-PSY",(4.9,6.7,7.6)),
          "p6": ("Jeroen",25,"male","M-BME",(8.5,8.5,8.5)),
          "p7": ("Max",18,"male","B-IBA",(7.9,2.0,6.5))
         }
```

1. `milk` in `shoppingList`
2. `minced beef` in `shoppingList`
3. The study of `Jan` in `grades`
4. The age of `Sophia` in `grades`
5. The name of participant 6 in `grades`
6. Judith's grade on the third partial exam in `grades`
7. The grade of participant 5 on the first partial exam in `grades`
8. The maximum grade achieved on the second partial exam

4.7.7 Exercise 7. Adjust a dictionary data set

The data set below originates from a (fictive) language course given at the University of Twente. During the course, the following information was stored about the participants:

age, gender, nationality, study programme, for each lesson, whether the participant was present or not (`True` indicates presence during the lesson and `False` absence), and the paricipant's score on the final exam.

```
dataset = {'p1':[21,"Female","Dutch","B-Psychology",[True,False,True,False,False],5
           'p2':[20,"Female","Dutch","B-Psychology",[True,True,True,False,True],8.4
           'p3':[21,"Female","Dutch","B-Applied_Mathematics",[False,True,True,False
           'p4':[23,"Male", "German","B-Communication_Science",[True,True,"n/a","n/
           'p5':["n/a","n/a","Dutch","M-Business_Administration",[False,False,True,
           'p6':[19,"Male","Swedish","B-Computer_Science",[True,False,False,True,Fa
           'p7':[19,"Male","German","B-Communication_Science",[True,True,False,True
           }
```

The lecturer has to perform some last minute changes to the data set before handing it in for statistical examination. Perform the following changes to the data set by accessing the relevant elements.

1. After taking a resit, participant 1 and participant 7 did pass the course. Participant 1 scored a 6.1 on the resit and participant 7 a 7.4.
2. Participant 5 forgot to put her age and gender on the registration form. Participant 5 is female and 23 years old.
3. The docent forgot to sign the presence of participant 4 during two lessons. Participant 4 was present during all lessons.
4. During the course, participant 2 changed her study programme from B-Psychology to B-Health Sciences.
5. After the final examination, it was revealed that participant 4 cheated during the exam. The examination board decided to void the student's achievements on the exam. In such a case, the additional note `expelled` is added to the student's record, indicating the student's misbehaviour.

4.7.8 Exercise 8. A dictionary data set

Fill in the participant information given in the table below in a Python dictionary with the following structure:

```
participants = {participant_no:(gender, age, nationality, profession)}
```

Participant No.	Gender	Age	Nationality	Profession
1	Male	19	Dutch	Student
2	Male	47	Dutch	Pharmacist
3	Male	31	Italian	PhD Student
4	Female	22	German	Student

Participant No.	Gender	Age	Nationality	Profession
5	Female	46	Dutch	Florist
6	Male	27	Dutch	Student
7	Female	22	Dutch	Police trainee
8	Female	26	Indian	Architect
9	Male	18	American	Student
10	Male	20	Chinese	Student

Using Python and by accessing the relevant elements in the dataset

1. Calculate the *average age* of the participants
2. Calculate the *standard deviation* of the age variable
3. Calculate the *maximum* and *minimum* of the age variable

Hint: Libraries provide built-in methods beyond the repertoire of the standard Python language. For instance, `NumPy` provides all kinds of methods for statistical operations in Python. You can import a library by typing `import (library name)` and make use of library-specific methods like this `(library name).(method)`. Take a look at the documentation of the `NumPy` library.

4.7.9 Exercise 9. Stroop extension

In this exercise you will implement a conditional execution of the interactive Stroop program. Imagine that you want to examine whether the number of colors influences the reaction time of participants in the Stroop experiment. In particular, you have gathered ten participants whom you randomly assigned to either a three or a five word version of the Stroop task. The table below summarizes which participant is assigned to which condition.

Participant No.	Condition
1	Stroop 3
2	Stroop 3
3	Stroop 5
4	Stroop 3
5	Stroop 5
6	Stroop 5
7	Stroop 5
8	Stroop 3
9	Stroop 5
10	Stroop 3

For this exercise, a modified version of the Stroop task is given. In `ch4ex6Stroop.py`, a rudimentary implementation of user console input is provided. In two additional `STATES`, the user enters his participant number and the entered number is stored for the rest of the program run. The modified script has a function `pickColor5()`, which implements the random selection of words and colors for the five-color version.

In order to implement the comparative Stroop version program (three vs. five colors) program, consider the following steps:

- Open `ch4ex6Stroop.py` and scan the changes to the original code. Try to understand the changes performed to the code.
- Incorporate the information contained in the table above on the condition assignment of the ten participants. Which data structure will you use for storing this information?
- Prepare the code for both Stroop task versions, a version with three words, three colors, and three buttons and the same for the five-word version. Choose any two additional colors.
- Think about which places in the program need to be adapted and which of them need adaptation depending on the condition.

4.8 Think further

- For more detailed information on *lists* beyond the scope of this chapter, you are advised to take a look at Wentworth, P., Elkner, J., Downey, A. B., & Meyer, C. (2015). How to think like a computer scientist: Learning with Python 3. Chapter 11. Lists
- For more information on *dictionaries*, including explanation on useful built-in functions of the dictionary class, you are invited to take a look at Wentworth, P., Elkner, J., Downey, A. B., & Meyer, C. (2015). How to think like a computer scientist: Learning with Python 3. Chapter 20. dictionaries
- More information on *tuples* can be found in Wentworth, P., Elkner, J., Downey, A. B., & Meyer, C. (2015). How to think like a computer scientist: Learning with Python 3. Chapter 9. Tuples

5 Loops

```
for i in range(2):
    print("Hello, World!")
```

```
## Hello, World!
## Hello, World!
```

Very much unlike human beings, computers are good at executing repetitive tasks without making errors. It just so happens that boring, repetitive tasks are an inescapable part of many exciting things. Take psychological research as an example. In conducting psychological research, you come up with a theory about human beings, about how things work between humans, or maybe between a human operator and a machine. Or, alternatively, you may have a research question, a question about how things work, how human beings function, why they make the mistakes they make. Now, what do psychologists do when they try to find an answer to their questions? Right, psychologists conduct empirical research.

Let's say, you have a research question and you try to answer your question by setting up an empirical experiment. You invite others to participate in your research. You repeat your experimental procedure with each and every participant. Depending on the kind of research you conducted, you repeat the qualitative inspection of the generated data for each participant. Depending on the statistical tool you use, you may end up filling in the data of each and every participant by hand into your statistical software. Now, here is the good news. Programming can help you unbore (yes, that is a word) the boring and repetitive bits of your exciting research.

At the root of boring tasks often lies repetition. You repeat an experimental procedure, you repeat the same quality checks for each participant's data, you perform the same data cleaning on each single row in a data frame, and so on. Human beings tend to make mistakes when they repeat the same task over and over again. At the same time, this is exactly where the strength of computers lies. Computers do not make mistakes when they repeat a task. When a computer makes a mistake, then it is because it was falsely instructed in the first place.

Chapter 5 introduces you to the mechanism that lies at the heart of a computer's paramount strength, repetition; or as it is called in computer jargon: *loops*. The Python language has a number of features that make it easier for you as a programmer to make your computer execute those boring, repetitive tasks for you.

5.1 The `for` loop

```
for letter in "Hello":
    print(letter)
```

```
## H
## e
## l
## l
## o
```

The `for` loop is one possibility to create repetition in a computer program. In computer science, the repeated execution of a set of statements is called *iteration*. The `for` loop *iterates* over a *sequence* of objects. Remember from chapter 4 Lists, that *lists* and *strings* are both sequences. They are *ordered* collections of elements, where strings are sequences of characters and lists can contain any kind of object. So what does it mean for the `for` loop to iterate over a sequence? It means that it visits each element in the sequence one by one, in an ordered fashion.

Regarding the syntax of `for` loops; a `for` loop begins with the `for` keyword, followed by the so-called *loop variable*. In the example above, the variable `letter` serves as the loop variable. The `in` keyword, which you encountered in chapter 4 follows on the loop variable. Then a *sequence* which can (amongst other things) be a list or a string is included. Finally, a colon `:` demarcates the end of the `for` statement. Semantically, the `for` statement in the above example says nothing else but

> For each letter that exists in the word "Hello"

On each iteration of the `for` loop, the *loop variable* takes on the value of the next element in the sequence. This means that on the first iteration, the loop variable `letter` has the value `H`, on the second iteration, it has the value `e`, on the third iteration, its value is `l`, and so on. The loop variable ceases to take on a new value once it is assigned the last element of the designated sequence, in this case the`o` of `"Hello"`.

Syntactically, after the `for` statement, the so-called *loop body* follows. The body of a loop specifies the statements that are to be executed on each iteration. In the example above, printing the current value of the loop variable `letter` is executed on each iteration of the loop.

The body of a loop needs to be *indented*. The end of a `for` loop is demarcated by indentation. Once the value of the last element in the sequence has been assigned to the loop variable, and the body of the loop has been executed, the loop terminates. The execution of the computer program is continued on the first line after the loop body, and indentation (or rather the lack thereof) demarcates this line, like so

```
for letter in "Hello":
    print(letter)
print("World")
```

Here, `print("Hello")` constitutes the first line after the `for` loop.

Instead of including a manually defined sequence in the `for` statement, such as the string `"Hello"`, you can use the `range` method to create a list of integers. For example, you may want to repeat a certain block of statements twice. However, the block of statements may not involve a suitable sequence over which you can make a `for` loop iterate. In such cases, you can use the `range` method to create a looping mechanism, like so

```
for i in range(2):
    print("Hello, World!")
```

```
## Hello, World!
## Hello, World!
```

Unless specified otherwise, the `range` method initiates a list that contains each integer starting at `0` until, but excluding the specified integer between the brackets `()`.

```
print(range(2))
```

```
## [0, 1]
```

Read the documentation of the `range` method for further information on how to customize the sequence of integers it produces.

```
help(range)
```

In summary, from the above it follows that there are at least two ways for iterating over *ordered* sequences of objects using `for` loops. First, you assign each element of the sequence iteratively to the loop variable, like so

```
a = ['a', 'b']
for i in a:
    print(i)
```

```
## a
## b
```

Second, you access the elements of the sequence through their indices, like so

```
for i in range(len(a)):
    print(a[i])
```

```
## a
## b
```

Which way of iterating over *ordered* sequences you prefer is up to you. Sometimes, the problem you try to solve with your code will require one way, sometimes the other. However, remember that you **cannot** access the elements of *unordered* collections like dictionaries using the index operator.

```
d = {'a': 1,
     'b': 2}
for i in range(len(d)):
    print(d[i])
```

```
KeyError: 0
```

5.2 The `while` loop

```
sum = 0
value = 1
while value < 5:
    sum += value
    value += value
    print "current sum is:", sum, "current value is:", value
```

```
## current sum is: 1 current value is: 2
## current sum is: 3 current value is: 4
## current sum is: 7 current value is: 8
```

```
print "The loop is over, the sum is:", sum
```

```
## The loop is over, the sum is: 7
```

A `while` loop allows you to repeatedly execute a number of statements as long as the condition in the `while` statement remains `True`. Note that the body of a while loop is never executed if the loop condition is `False` at its first execution.

The body of a `while` loop should change one or more (loop) variables so that the loop condition eventually becomes `False`. Otherwise, the loop will continue forever and we speak of an *infinte loop*. If unintended, infinite loops can manoeuver your program into a dead end.

At times it may be confusing whether either a `for` or a `while` loop will suit your needs best, ask yourself the following questions:

- Do I know in advance how many times I need the loop body to be executed? Any problems like "search a list of words" or "execute until the desired number of trials is reached" suggest that a `for` loop will suit your needs best.
- Do I require the computations to stop when some condition is met, but I cannot calculate in advance when (of if) this will happen? Chances are high that a `while` loop will fit your needs best in this case.

5.3 The `break` statement

```
myNumber = 17
while True:
    guess = input("Enter a number:")
    if guess > 17:
        print "My number is smaller than", guess
    elif guess < 17:
        print "My number is larger than", guess
    else:
        break
print "Ding! Ding! You guessed correctly, my number is", myNumber
```

The `break` statement allows you to immediately exit any loop. The next statement that is executed is the first after the loop body. Interactive programs that process user input often require to exit their loops prematurely. Depending on the input given, the program may need to transition to the next state or terminate altogether.

5.4 Control flow

By now you will have noticed that by default, Python faithfully executes any program in a top-down order. But what if you want to change the order in which specific parts of your program are being executed, possibly depending on external conditions such as the time of the day, or user input?

Yes, you may have guessed it already, this is achieved by using *control flow statements*. Python knows three control flow statements: : `if`,`for`, and `while`statements. We already

encountered `if` statements in Chapter 3 Conditionals. Control flow statements are readily used in interactive software such as the Stroop experiment

```
STATE = "welcome"
while True:
    # Changing states by user input
    for event in pygame.event.get():
        if STATE == "welcome":
            if event.type == KEYDOWN and event.key == K_SPACE:
                STATE = "prepare_next_trial"
                print(STATE)

        if event.type == QUIT:
                STATE = "quit"
```

In principle, the above code can run infinitely, unless the user triggers the `QUIT` `event.type`. Note that in the original Stroop experiment, there is another condition that prevents the program from running infinitely, namely a maximum number of trials.

Within the main `while` loop, a `for`loop manages the changing of program states as a consequence of user input. To this end, the loop variable `event` iterates over *event objects* stored in an *event queue*, which is essentially a special sequence devised by the `Pygame library`. The `pygame.event` *module* knowns a number of event types, among which the `KEYDOWN` event, referring to the event that the user presses a key on the keyboard.

In the example above, several conditions have to be fulfilled so that the program state changes to `"prepare next trial"`: The current program state has to be `"welcome"`, the event has to be a `KEYDOWN`and the key pressed has to be `SPACE`. Note that in the program above, program termination can be achieved at any time through the `"quit"` state.

5.5 Common errors

- Trying to make a `for` loop iterate over an *integer* instead of a *sequence*. Very often, you will want your program to modify all elements of a given sequence according to a set of statements. Logically, you will need as many iterations as elements in the sequence. This often leads to `for`statement definitions such as:

```
myList = [1,2,3,4,5]
for i in len(myList):
    myList[i] += 1
```

```
TypeError: 'int' object is not iterable
```

Probably, your intention was to make the loop repeat as many times as there are elements in `myList`. However, `len(myList)` returns an *integer*, not a *sequence* object. Instead, you will have to convert the *length* of `myList` into a *sequence*, for instance by using `range()`.

```
myList = [1,2,3,4,5]
for i in range(len(myList)):
    myList[i] += 1

print myList
```

```
## [2, 3, 4, 5, 6]
```

- `index out of range` errors tend to slip in when working with loops

```
myList = [range(10),[1,2,3]]
for i in range(len(myList[0])):
     myList[0][i] = 0
     myList[1][i] = 0
```

```
IndexError: list assignment index out of range
```

Always make sure that sequences you intend to modify in a `for` loop body actually have as many elements as the sequence over which you are iterating in the `for` statement.

5.6 Debugging

5.7 Exercises

5.7.1 Exercise 1. Following the control flow

Read the script below carefully. Indicate the values of the following expressions after the script has finished executing:

```
dataset.keys()
dataset[dataset.keys()[0]]
condA[0]
condA[0][0]
len(condA)
len(condB)
meanA
meanB
```

You can check your answers afterwards by running the script.

```
import numpy as np
dataset = {'p1':(21,"Female","Dutch","B-Psychology"),
           'p2':(20,"Female","Dutch","B-Psychology"),
           'p3':(21,"Female","Dutch","B-Applied_Mathematics"),
           'p4':(23,"Male", "German","B-Communication_Science"),
           'p5':(20,"n/a","Dutch","M-Business_Administration"),
           'p6':(19,"Male","Swedish","B-Computer_Science"),
           'p7':(19,"Male","German","B-Communication_Science"),
           'p8':(24,"Female","Italian","M-Psychology"),
           'p9':(23,"Female","Italian","M-Communication_Science"),
           'p10':(18,"Male","Dutch","B-Computer_Science")
           }
condA = []
condB = []
ageA = []
ageB = []
for participant in dataset.keys():
    if dataset[participant][1] == "Female":
        condA.append(dataset[participant])
    elif dataset[participant][1] == "Male":
        condB.append(dataset[participant])
for participant in condA:
    ageA.append(participant[0])
count = 0
while count < len(condB):
    ageB.append(condB[count][0])
    count += 1

meanA = np.mean(ageA)
meanB = np.mean(ageB)
```

5.7.2 Exercise 2. Debugging

The script below calculates some descriptives from `data`. The code is still somewhat buggy. Debug the script and indicate the values of `fastest`, `slowest`, and `all_mean`.

```
data = {'p1':(21,"Female","Condition B",[0.675,0.777,0.778,0.62,0.869]),
        'p2':(20,"Female","Condition A",[0.599,0.674,0.698,0.569,0.7]),
        'p3':(21,"Female","Condition A",[0.655,0.645,0.633,0.788,0.866]),
        'p4':(23,"Male", "Condition A",[0.721,0.701,0.743,0.682,0.654]),
        'p5':(20,"Male","Condition B",[0.721,0.701,0.743,0.682,0.654]),
```

```
        'p6':(19,"Male","Condition B",[0.711,0.534,0.637,0.702,0.633]),
        'p7':(19,"Male","Condition B",[0.687,0.657,0.766,0.788,0.621]),
        'p8':(24,"Female","Condition A",[0.666,0.591,0.607,0.704,0.59]),
        'p9':(23,"Female","Condition B",[0.728,0.544,0.671,0.689,0.644]),
        'p10':(18,"Male","Condition A",[0.788,0.599,0.621,0.599,0.623])
        }
fastest = ("initialization",100)
for participant in data:
    RTsum = 0
    for i in len(data[participant][3]):
        RTsum += data[participant][3][i]
    RTmean = RTsum/len(data[participant][3])
    if RTmean < fastest[1]:
        fastest = (participant,RTmean)

slowest = ("initialization",0)
for participant in data:
    RTsum = 0
    for RT in participant[3]:
        RTsum += RT
    RTmean = RTsum/len(data[participant][3])
    if RTmean > slowest[1]:
        slowest = (participant,RTmean)

all_mean = 0
all_sum = 0
number_of_trials = 0
for participant in data:
    counter = 0
    while counter < len(data[participant][3]):
        all_sum += data[participant][3][counter]
all_mean = all_sum/number_of_trials*len(data)
```

5.7.3 Exercise 3. Nested loops

Nested data often requires *nested loops* for answering certain questions about the data. Let's have a look at the `list` data set below. each participant has a participant number paired with a list of additional experiment information.

```
participants = [
    ("p1", ["Condition 1","Location A","withdrew","no audio"]),
    ("p2", ["Condition 1","Location A","no audio","no video"]),
```

```
    ("p3", ["Condition 2","Location B"]),
    ("p4", ["Condition 1","Location A","withdrew"]),
    ("p5", ["Condition 2","Location A"]),
    ("p6", ["Condition 2","Location B","withdrew"]),
    ("p7", ["Condition 1","Location A","no video"]),
    ("p8", ["Condition 1","Location B"]),
    ("p9", ["Condition 2","Location B","withdrew"]),
    ("p10",["Condition 2","Location A","withdrew"])]
```

If we want to know how many participants withdrew their participation we need a counter, and for each participant we need a second loop that tests each of the experiment information in turn on the `"withdrew"` keyword:

```
counter = 0
for (participant,expInfo) in participants:
  pass
        # to be implemented by you

print "The number of participants who withdrew their participation is", counter
```

Implement the second `for` loop that iterates over the experiment information and count the number of participants who withdrew their participation.

5.7.4 Exercise 4. Data transformation using loops

- Write a script that transforms the data set `participants` from Exercise 3 into a dictonary data set.
- Extend the script with a `counter` mechanism that counts the number of participants who were tested at location A and location B respectively.

5.7.5 Exercise 5. Calculating a mean

For the sequence `seq` calculate the arithmetic mean using a `while` loop. Check your implementation using the `mean()` method of the `NumPy` library.

```
import numpy as np
seq = range(1000)
```

5.7.6 Exercise 6. A guessing game

Make a Python script `guessing.py` that implements the following behaviour:

- At the beginning of the script, a random number between 0 and 1000 is generated.
- The program continuously asks for user input prompting the user with the request to enter a number.
- If the user guesses the number correctly, the program gives feedback that the number was guessed correctly. The program also shows the number of guesses needed. Finally, the program terminates.
- If the guessed number is too large, the program gives feedback that the number is smaller.
- If the guessed number is too small, the program gives feedback that the number is larger.
- If no input is given, the program should terminate. In this case, the program should terminate if the given input is *empty*. *Hint:* Use the internet to figure out how to handle empty input.

Hint: Prompting and feedback via the console is sufficient.

6 Functions

```
def hello(repetitions):
    for i in range(repetitions):
        print("Hello, World!")
hello(2)
```

```
## Hello, World!
## Hello, World!
```

Functions are reusable pieces of code. By giving a block of statements a name, functions allow you to make use of that block anywhere in your program and as many times as you want. You can use the block of statements without having to write out the actual code contained in the block. This makes your code a lot easier to read and it eliminates a great deal of repetitive code. If you want to make a change later, you only have to do so in one place. We have already seen and used many built-in functions of Python, such as `len()` and `range()`. In this chapter, you will learn how to *define* and *call* your own functions.

6.1 Defining and calling functions

Before you can use a function, it has to be defined in your program.

```
def echo(text):
    print(text)
```

The syntax for defining a function includes the `def` keyword, an identifier name for the function (`echo`), followed by parentheses which may optionally enclose one or multiple function *parameters* (`text` in the above example). Finally, a colon `:` demarcates the end of the definition statement. The body of a function may include any number of statements and is indented from the `def` statement.

Functions enable us to organize our programs into chunks; code chunks as well as mental chunks. Programs have goals, and usually propose a solution to a given problem. Just as with real-life goals and problem solving, subdividing a task at hand into smaller subgoals helps us organize our approach.

Defining a function does not make the function run. To run a function, you need to *call* it.

```
def echo(text):
    print(text)
echo("Hello there, programming aspirant!")
```

```
## Hello there, programming aspirant!
```

Function calls consist of the name of the function, and if the function takes one or more arguments, some values are included in the parentheses following the function name.

6.2 Function parameters and arguments

When calling a function that takes *parameters* you should supply the function with values which the function can use to do something. *Parameters* are specified within parentheses during function definition. Multiple parameters are separated by commas. Mind the terminology, the names given to the values during function definition are called *parameters* and the values you supply to the function are called *arguments*.

You can think of parameters as variables which only exist in the function. By default, their values are assigned when you call the function and they cease to exist once the function has finished running. Their sole purpose is to serve internal computations within the function.

Another way of saying this is that variables inside functions are *local*. Local variables are not in any way related to other variables outside a function.

```
x = 100
def myFunction(x):
    x += 50
    print("Local x equals:", x)

print("Global x equals:", x)
```

```
## ('Global x equals:', 100)
```

```
myFunction(0)
```

```
## ('Local x equals:', 50)
```

In the above example, the *scope* of the variable `x` within `myFunction(x)` is local, whereas the scope of `x` outside the function is said to be *global*.

6.3 The `return` statement

But what if you want to use a value that has been assigned to some variable and manipulated within a function somewhere else in your program? Let's make a few changes to `myFunction(x)` so that the function *returns* a value that can be used outside `myFunction(x)`.

```
def myFunction(x):
    x += 50
    print("Local x equals:", x)
    return x
x = 100
print("Global x equals:", x)
```

```
## ('Global x equals:', 100)
```

```
x = myFunction(0)
```

```
## ('Local x equals:', 50)
```

```
print("After function call, global x equals:", x)
```

```
## ('After function call, global x equals:', 50)
```

The `return` statement enables us to retrieve one or more values from a function. When returning multiple values, Python makes use of *tuple assignment*. Make sure to provide enough variables on the left side of the assignment operator to unpack the returned tuple. Consult Chapter 4.1 for a revision on tuples and tuple assignment, or take a quick look at How to Think Like a Computer Scientist: Tuple Assignment.

To understand the example code above, it helps to realize that function definitions do not affect the flow of execution of a program. Remember that by default, Python faithfully executes a program in a top-down fashion. Function definitions need to precede function calls, but a function's statements are not executed until the function is called.

When a function is called, the execution flow of the program takes a detour. Instead of proceeding to the next program line, it jumps to the first line of the called function, executes all function statements in top-down order, and then jumps back to where it left off.

```
def echo(text="This is some random text made for the purpose of interruption"):
    print(text)

print ("Hi there, ")
```

```
## Hi there,
```

```
echo()
```

```
## This is some random text made for the purpose of interruption
```

```
print("how are you?")
```

```
## how are you?
```

6.4 Functions can call other functions, and themselves

This sounds simple enough, but it becomes slightly more complex when functions call one another, and even themselves.

```
def summation(alist):
    sum = 0.0
    for i in range(len(alist)):
        sum += alist[i]
    print "Sum of the list is:", sum
    return sum
def average(alist):
    return summation(alist)/len(alist)

print(average([1,2,3,4]))
```

```
## Sum of the list is: 10.0
## 2.5
```

Note how the above implementation of `average` calls the function `summation` to calculate the sum of the items in the list given as an argument (`alist`) to the function call of `average`. While executing the `return` statement in `average`, `summation` is called and the program's flow of execution jumps to the first line of `summation`. After all statements contained in `summation` are completed, the flow of execution jumps back to where it took

off and replaces the expression `summation(alist)` by the value returned by the function call to `summation` (in this case, `10.0`).

What's the morale of this detour? When reading and trying to understand a program, you should not simply read from top to bottom. Instead, follow the flow of execution to understand what is going on.

6.4.1 A word about *Recursion*

Functions can call other functions, including themselves. A function calling itself is said to take a *recursive* approach to problem solving. *Recursion* is a method involving the subdivision of a problem into smaller and smaller subproblems until a subproblem small enough is obtained that can be solved easily.

You already know how to calculate the sum of a list of numbers using *loops*. Let's pretend for a moment that there are no `while` or `for` loops in Python. How would you approach calculating the sum of a list of numbers then?

While the topic of *Recursion* is beyond the scope of this book, the interested reader is invited to take a look at Problem Solving with Algorithms and Data Structures: Chapter 4. Recursion, by Brad Miller and David Ranum (2011). In particular, Section 4.3. Calculating the Sum of a List of Numbers adresses the calculation of a sum of a list of numbers by taking a recursive approach. Recursion is a powerful technique that enables solutions to problems that would otherwise be difficult to express in code.

6.5 Stroop functions

6.6 Common errors

- Trying to use local variables globally. Remember that any function parameter and any variable declared within a function statement is *local*, meaning that it cannot be accessed from outside the function.

```
def average(alist):
    s = sum(alist)
    return s/float(len(alist))

m = average([1,2,3,4])
print(s)

NameError: name 's' is not defined

NameError: name 's' is not defined
```

- Forgetting to return a value at the end of the function definition. In truth, functions do not necessarily *return* values, depending on the purpose of the function. Functions which return one or more value(s) are called *fruitful* functions, those that do not have a `return` statement are called *void* functions. However, now and then we tend to forget to return values which we do want to use outside the function. Consider the example below.

```
def maximum(alist):
    if len(alist) == 1:
        return alist[0]
    elif len(alist) != 0 and len(alist) > 1:
        m = 0
        for i in range(len(alist)):
            if alist[i] > m:
                m = alist[i]
m = maximum([2,4,6,8,10])
print("m equals:", m)
```

```
## ('m equals:', None)
```

Instead of having assigned the maximum value contained in `[2,4,6,8]` to `m`, `m` apparently became `None`, whatever that is. Let's examine this `None` further.

```
type(m)
```

```
NoneType
```

```
print(m + 1)
```

```
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

We accidentally created an object of type `NoneType` by forgetting to return `m` in our function definition of `maximum`. As you see above, this can raise errors somewhere else in your code, whose cause can be tricky to trace back in more complex programs. Therefore, always make sure to double check whether your functions return all values you need outside of the function.

- Forgetting to assign a value returned by a function.

```
def average(alist):
    s = sum(alist)
    return s/float(len(alist))
a = 0
mylist = [1,2,3,4]
average(mylist)
print("average of mylist is:", a)
```

The average of `mylist` is obviously not `0`. However, we forgot to assign the value that the function `average` returns to a variable (in this case to the variable `a`). By doing so, the calculated average value is lost in memory and the variable `a` keeps its initial value, `0`.

- Disregarding tuple assignment when a function returns several values. Keep an eye on how you assign multiple return values. If handled carelessly, this can easily become a source of errors in the remainder of your program.

```
def sumAverage(alist):
    s = sum(alist)
    return s, s/float(len(alist))
result = sumAverage([1,2,3,4])
s, av = sumAverage([1,2,3,4])
print("result equals", result)
```

```
## ('result equals', (10, 2.5))
```

```
print("s equals", s, ", av equals", av)
```

```
## ('s equals', 10, ', av equals', 2.5)
```

Note how the first variant returns a tuple with both return values as elements, and the second variant assigns each return value separately to a variable name.

- Providing the wrong number of arguments during a function call.

```
def average(alist):
    s = sum(alist)
    return s/float(len(alist))

average([1,2],[3,4])
```

```
TypeError: average() takes 1 positional argument but 2 were given
```

- Providing a wrong data type as a function argument. The body of a function is written assuming that you provide the intended data type as argument during a function call. In software engineering, every piece of new code is intensively checked with the help of so-called unit tests before going into production. Amongst other things, these unit tests include *type checks* on the data input and output format of self-built functions. This minimizes the chance that a new piece of code crashes the rest of the program. Unit tests are beyond the scope of this introductory book, so you will probably encounter one or another instance where you accidentally provide a wrong data type to a function. Providing a value of an unintended data type can raise all kinds of follow-up errors.

```
def average(alist):
    s = sum(alist)
    return s/float(len(alist))

average(True)

TypeError: 'bool' object is not iterable

average("Hello")

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

6.7 Exercises

6.7.1 Exercise 1. Following the control flow

Read the following script carefully. Try to follow the flow of execution. Indicate the values of `x`,`y` and `anumber` after the program has finished executing. You may use a calculator to perform calculations. You can check your answers afterwards by running the script.

```
def mean(numbers):
    return float(sum(numbers))/max(len(numbers),1)

def sumMean(myList):
    return sum(myList), mean(myList)
def add(anumber,to_add):
    return anumber + to_add
def main():
    x = 0
    y = 0
```

```
    anumber = 0
    myList = range(11)
    x = add(x,37 + mean(myList))

    if x > 44:
        x, y = sumMean(myList)
    else:
        x, y = sumMean([1,2,3,4])

    y += add(x,5)
    anumber += add(y,5)

    if y > 50:
        y = mean([x,y])
    else:
        anumber = mean([x,max(myList)])

    print("x equals", x)
    print("y equals", y)
    print("anumber equals", anumber)

main()
```

6.7.2 Exercise 2. An imperfect list sorting attempt

Given the following instructions

- Take the two lists `myList1 = [5,2,1,3,0]` and `myList2 = [9,7,8]` and implement an algorithm that sorts them in ascending order so that a sorted list containing all integers from 0 to 10 is obtained.
- Fill in any missing numbers if applicable.

A fellow student made an attempt of sorting the lists. In order to do so, they defined two helper functions, `insert` and `swap`. The attempt followed the approach of first bringing both lists into a correct order separately before combining them.

Unfortunately, the attempt still contains a number of errors. Find those errors and describe in your own words what goes wrong. *Hint*: There are six errors in the code.

```
import copy

# Insert an element at a given position in a list
def insert(a,position,alist):
  result = copy.deepcopy(alist)
```

```
  result = result[:position] + [a] + result[position:]

# Swap the position of two elements a and b in a list
def swap(a,b,alist):
  index_a, index_b = alist.index(a), alist.index(b)
  index_a, index_b = index_b, index_a
  result = copy.deepcopy(alist)
  result[index_a], result[index_b] = a,b
  return result

myList1 = [5,2,1,3,0]
myList2 = [9,7,8]

# sorting myList1
swap(5,0,myList1)
swap(2,1,myList1)
myList1 = insert(4,myList1)

#sorting myList2
myList2 = swap(myList2,8,9,7,8)
myList2 = insert(10,3,myList2)

result = myList1 + myList2
print(result)
```

6.7.3 Exercise 3. An erroneous sorting algorithm

Instead of sorting a list manually as in exercise 2 which you may figure is quite labour intensive (especially when your amount of data is too large to allow manual sorting), people have come up with a number of more efficient sorting algorithms. In essence, these algorithms are functions that take an unsorted list as input, perform a number of internal manipulations and return a sorted list. Below you find an (erroneous) implementation of *bubble sort*. The main idea of bubble sort is iteratively going through a list and exchanging the position of adjacent elements if they are out of order.

Feel free to take a look at Problem Solving with Algorithms and Data Structures: Chapter 5. Sorting and Searching, 5.7: The Bubble Sort for a more detailed description of the bubble sort approach.

Correct all errors in the below implementation of bubble sort so that the script runs error-free. The implementation makes use of the `swap` helper function you first encountered in exercise 2. There are no errors in the swap function.

```
import copy
```

```
import random

def swap(a,b,alist):
    index_a, index_b = alist.index(a), alist.index(b)
    index_a, index_b = index_b, index_a
    result = copy.deepcopy(alist)
    result[index_a], result[index_b] = a,b
    return result

def bubbleSort(alist):
    result = copy.deepcopy(alist)
    for iteration in len(result):
        for index in range(len(result)-1,0,-1):
            if result[index] > result[index-1]:
                result = swap(result[index],result[index-1])
    return result

myList = range(51)
random.shuffle(myList)
print(myList)
print(bubbleSort(myList))
```

6.7.4 Exercise 4. Chunking

6.7.5 Exercise 5. Add functionality to the stroop task

6.7.6 Exercise 6. Insertion sort algorithm

In exercise 3, you encountered one of the infamous sorting algorithms known to the programming community: *bubble sort*. There are, however, more efficient sorting algorithms. In this exercise, you will take your sorting skills to the next level and implement *insertion sort*.

Problem Solving with Algorithms and Data Structures: Chapter 5. Sorting and Searching, 5.9: The Insertion Sort explains the insertion sort approach excellently. The main idea behind insertion sort is that it builds a sorted sublist in the left part of an unsorted list and continuously updates that list by comparing a designated element of the unsorted list to each element in the sublist.

The chapter on insertion sort by Miller and Ranum includes a sample implementation of the algorithm. You are, however, strongly advised to build the algorithm yourself so that you understand what happens at each step.

You are therefore required to add an elaborate documentation to your code (in the form of comments explaining what you do at each step). *Elaborate* means that if you were to

read back your code a month later, you would still be able to explain how insertion sort works.

There is, by the way, a nice graphical illustration of the insertion algorithm on its Wikipedia page. It may help to get a grasp of how the algorithm works.

6.8 Think further

For an alternative approach to explaining *functions* in Python and additional information, you are invited to take a look at the following resources:

- Wentworth, P., Elkner, J., Downey, A. B., & Meyer, C. (2015). How to think like a computer scientist: Learning with Python 3. Chapter 4. Functions
- Tutorials Point India Private Limited Company: Python Basic Tutorial - Functions
- Harrington, A. N. (2017). Hands-on Python Tutorial: 1.11 Defining Functions of your Own

6.9 References

- Miller, B. N., & Ranum, D. L. (2011). Problem Solving with Algorithms and Data Structures Using Python SECOND EDITION. Franklin, Beedle & Associates Inc..

7 Working with data

7.1 Elements of data processing

In the previous chapter we have dealt with functions as reusable pieces of code. There is another common notion of what a function is: *a function processes data*. The basic model of data processing is the following:

So, what is data? As psychologists we are used to think of data as observations from empirical studies that we process by statistical procedures. That precisely matches our above model:

In Python, as a general programming language, there exist libraries that implement tables of observations, as well as the t-test (and more modern statistical procedures, luckily). However, we don't want to bother you with statistics for the moment. Instead, take another look at two function that we have introduced in 6:

```
def summation(alist):
    sum = 0.0
    for i in range(len(alist)):
        sum += alist[i]
    print "Sum of the list is:", sum
    return sum
summation([1,2,3,4])
```

```
## Sum of the list is: 10.0
```

The function `summation` takes a list of values and computes the sum. Again, that satisfies the basic model of data processing:

It is easy to see how input and output are implemented with Python functions:

- function arguments provide the input
- the function body does the processing
- the return statement gives the output

Consequently, every function that takes arguments and has a return statement can be considered data processing. Furthermore, data processing can take place as chains. Let's take the variance of a list of numbers as an example. The formula for variance, shown below, reveals that the variance statistic is produces in three steps: first, the residuals are computed, these are squared an finally the mean value is taken.

Let's see how this is implemnented in Python. For the squared residuals, we create our own function, but resort to the function `mean` in the `numpy` library. Note how the two defined functions operate on a dedicated list for output, rather than modifying the original one. Recall that lists are called by reference, which means the original list would otherwise be altered by the function.

```
from numpy import mean
def residuals(lon):
    avg = mean(lon)
    output = []
    for i in lon:
          output.append((i - avg))
    return output
def square(lon):
    output = []
    for i in lon:
          output.append(i**2)
    return output
LON = [1, 8, 2, 9]
RES = residuals(LON)
SQR = square(RES)
VAR = mean(SQR)
print VAR
```

```
## 12.5
```

The above code represents such a data processing in such a way that the intermediate results are stored in dedicated variables (`RES`, `SQR`). While this makes the processing chain clear in the code, for longer chains there is the downside of length and cluttering the program with variables that are just used once. A more compact form to write the chain is by *nesting function*:

```
from numpy import mean
def residuals(lon):
    avg = mean(lon)
    output = []
    for i in lon:
          output.append((i - avg))
    return output
def square(lon):
    output = []
    for i in lon:
          output.append(i**2)
    return output
LON = [1, 8, 2, 9]
VAR = mean(square(residuals(LON)))
print VAR
```

```
## 12.5
```

Nested function are evaluated inside-out. Consequently, when setting up such a chain, one begins with the first processing step and builds the subsequent one "around" it. While this is conveniently compact, it lacks the intuition of the data processing chain. Therefore, it is best in many situations, to encapsulate the whole line as a dedicated function:

```
from numpy import mean
def residuals(lon):
    avg = mean(lon)
    output = []
    for i in lon:
          output.append((i - avg))
    return output
def square(lon):
    output = []
    for i in lon:
          output.append(i**2)
    return output
def variance(lon):
```

```
    return mean(square(residuals(lon)))

LON = [1, 8, 2, 9]
print variance(LON)
```

```
## 12.5
```

This simple example highlights another important aspect about the data processing model: models can be nested. The function variance is itself a data processing chain, but it can take a role in a more global data processing chain. For example, when you prepare a summary statistics table for your report, you use the variance function as part of it. You do care about its required input and know what it will produce, but you don't worry about the above processing chain, anymore. That is called a *black box*.

7.2 Data tables

The purpose of psychological experiments is to collect data from participants under various conditions. Imagine you wanted to analyze the data from a single participant in the Stroop task. Perhaps, you wanted to analyze it as a factorial linear model (or ANOVA, if you like). Your data analysis program most likely expects the data in the following form:

```
## # A tibble: 8 x 3
##   Condition   Correctness    RT
##   <chr>       <lgl>       <dbl>
## 1 incongruent FALSE        801.
## 2 incongruent FALSE        697.
## 3 congruent   TRUE         765.
## 4 incongruent FALSE        814.
## 5 incongruent FALSE        631.
## 6 incongruent TRUE         686.
## 7 incongruent TRUE         693.
## 8 congruent   FALSE        732.
```

How would you create such a table in Python? The most simple form is to regard every column as a separate list and you would initialize the data gathering by creating three empty lists. In the main part of the program you can then populate the three lists separately, for example:

```
Condition = []
Correctness = []
RT = []
Condition.append(this_condition)
Correctness.append(this_correctness)
RT.append(this_reaction_time)
```

7.3 Down the sink: Writing data files

In the case of psychological experiments, the main part of the data processing is most likely done in other programs, such as R, SPSS or Excel. These programs can make little use of data that is stored in Python variables. Rather, data is read in from files. While all programs have their own data format for tables (and specialized functions exist for some), the Lingua Franca of file based data exchange is the *comma-separated-values (CSV)* form. This is almost as easy as it sounds:

- every row of data is represented as a row of text
- all values in a row are separated by commata

The example data set above would look like the following in CSV:

```
write.csv(D_Stroop, file = "", eol = "\n\n")
```

```
## "","Condition","Correctness","RT"
##
## "1","incongruent",FALSE,800.921185693852
##
## "2","incongruent",FALSE,696.864295047379
##
## "3","congruent",TRUE,765.243482711174
##
## "4","incongruent",FALSE,814.332269635055
##
## "5","incongruent",FALSE,630.556964944383
##
## "6","incongruent",TRUE,686.060561659131
##
## "7","incongruent",TRUE,693.333933180317
##
## "8","congruent",FALSE,731.797519903504
```

In order to write such a file, we could use a loop function, together with string concatenation. Fortunately, the `csv` library that comes with Python implements CSV export

right-away. However, it is not as simple as issueing just one command. Dealing with files is an issue on its own. Unlike variables, which are linked to a running program, files are just dumb sinks on the hard disk on your computer. What must not happen is that, for example, two programs modify the file simultaneously, as this would create a mess. Therefore, the operating system takes special care of files and ensures that only one programn can write it at a time (simultaneous reading is less of a problem). To signal the operating system that a file is being written, some decoration is necessary. More specifically, the programmer must first create a *fileg handler* on the file, which is what the command `open` does.

Then the csv library commands operate on the writing handler. This happens by means of a *writing handler*, which is created by the function `csv.writer`. Then, the function `writerow` adds the rows of a table, one by one. That is slightly inconvenient, because our data table is arranged by columns (the three lists) in the first place. For that reason, the `for` loops first gathers every row of the table as a list `this_row`. And finally, the `close` command signals the operating system that the file is no longer in use.

```
Condition = ["incongruent", "incongruent", "congruent"]
Correctness = [False, False, True]
RT = [800, 696, 765]
import csv
myfile =  open("Stroop.csv", mode = "w")
writer = csv.writer(myfile)
for i in range(0, len(Condition)):
    thisrow = [Condition[i], Correctness[i], RT[i]]
    writer.writerow(thisrow)
myfile.close()
```

As a general rule, it is good to keep the period between open and close short. Especially, one should not open the file at the start of the program and close it when it quits. Programs in development (and written by apprentices) tend to crash and that means data loss and potential file corruption. The safest way probably is to open the file after an observation has been gathered, append the observation and close it immediatly.

So how do we append rows to an existing file? A closer look at the `open` function reveals that it takes a second parameter, the `mode`, which is of type string. Three major mode exists (plus a bunch of fine-tuning options):

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
```

```
##
##     filter, lag


## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
data_frame(mode = c("r", "w", "a"),
           ` ` = c("reading", "writing", "appending")) %>%
knitr::kable()
```

mode	
r	reading
w	writing
a	appending

Hence, the code for writing an observation one-by-one could be:

```
import csv
myfile =  open("Stroop.csv", mode = "a")
writer = csv.writer(myfile)
writer.writerow([thisCondition, thisCorrectness, thisRT])
myfile.close()
```

7.4 May the source . . . reading data files

A processing chain starts with a data source, no matter what. The source can be one of three kinds:

1. a function call, for example one that produces random numbers
2. user input, such as the response time, which is derived from a button press
3. data files

The first two options we have already covered in this course. In the following we will see how to read in data from files. Files are typically produced by other programs. For tables that could be Excel (or a programming editor, if you like). As we have seen, CVS is a good choice.

In the Stroop program, we currently use only one external source of input, the user responses. That is definitely not a source we could replace by a prepared file. Reading data from files seems unncecessary at first, but in fact can make the program more flexibel when used by non programmers. A possible scenario for the Stroop task is that different experimenters use it, but with their own sets of colors. Here is how one could

do it. First, we prepare a table that assigns color words to their respective RGB codes and stores it as a CSV file. Any spreadsheet program is suited for the job.

word	R	G	B
violet	180	0	180
green	0	150	150
orange	255	150	0

To give a more compelling example for when it is useful to load stimuli at runtime: A modified version of the experiment is the primed Stroop task that by which one can assess the strength of certain semantic associations. It differs from the original in that it

- shows a priming picture upfront (e.g. a wolf)
- uses no color words, but target words that the participant may associate with the picture (e.g., the word "grandmother")

The assessment of association strength typically takes place within a certain domain, e.g. attitude towards computers or fairy tales. Researchers may therefore want to use same program, but feed it with different sets of images and target words. What one could do, is:

- import one table with the target words
- import one table with the filenames of all images
- read the images in one-by-one

7.5 Exercises

7.5.1 Exercise 1

Write a simple image presentation program, that takes a table of the following form as input and changes to the following picture on key press.

```
data_frame(File = c("Image01.jpg", "Image02.jpg", "Image03.jpg"))
```

```
## # A tibble: 3 x 1
##   File
##   <chr>
## 1 Image01.jpg
## 2 Image02.jpg
## 3 Image03.jpg
```

When that works, make the program automatic. Add a column to the table that gives the presentation time for an image in seconds.

7.5.2 Exercise 2

With the Python library `openpyxl` you can read and write Excel files, directly. Work through (this tutorial)[https://automatetheboringstuff.com/chapter12/]. Now modify your presentation program to use an Excel files as input.

7.6 Think further

7.7 References

8 Interaction

When you decided to learn programming you probably had something different in mind than putting some values in variables or compare apple with pears in conditionals. Most likely, your were more thinking of creating something with a little more magic then ASCII letters printed to the console. Perhaps, you already have an idea for a smartphone app on your mind. Who hasn't?

We won't push you that far. But, with conditionals in your hand, you are prepared to get to the next level and develop *interactive programs* Interactive programs are those that run continuously and do something when the user presses a button or moves and clicks the mouse. They don't do something completely random, except for some artistic installations, perhaps, but react in a purposeful way to user input. We can say, that a program executes an *interaction model*, a set of rules that describe the programs behaviour. Then we show you how to write interactive programs very efficiently by turning the transition table into *transitionals*.

In this chapter we do more than just show you how to use a conditional or a loop. You will notice that writing an interactive program is more than just coding. It is a *development process* with a sequence of steps where the output of one step is input for the next.

```
    Your brain does              The result is
    [Interaction modelling]  --> Transition table
--> [Coding transitionals]   --> Interactive prototype
--> [Coding presentationals] --> Pygame program.
```

First, we will teach you how to analyze an interaction problem and note down your ideas in a structured way, a *transition table*. Then we'll show you how to implement a prototype for testing the interaction protocol by turning the transition table into *transitionals*. Finally, we'll explain in more detail how to put the elements of interaction together in Pygame and give them a fluent appearance.

8.1 Interaction models

An interaction model describes when the program does what. As simple as this definition is, making an interaction model is not. Rather, creating this model is an important part

of sincere software development and happens long before you start coding. In essence, when starting to work on an interactive program, the first you do is: switch off your computer, pull out pencils and paper and scratch your head.

What we are asking you to do is envisioning the program you are going to write, eventually. The formal output of the interaction analysis is just one table that somewhat abstractly describes the flow of the program. That does not mean, that during the analysis you may do use other representations, such as flow charts or screen scribbles. The opposite is the case: seeing your program from different angles, will help you to come up with a better interaction model and the results can be used downstream, such as screen scribbles for the visual design.

As complex it is, an interaction model can be broken down into three sets of elements: states, transitions and events:

- states are the different modes a program is in.
- transitions let the program move from one state to another
- events make these transitions happen

The scope of this book covers rather simple interactive programs only, psychological experiments. In order to explain the basics of interaction programming, we go for something even more simple: a Dutch traffic light.

8.1.1 Identifying states

The first step in the analysis is to *identify states* of the program. A state is a rather abstract term, so let's see a few examples:

- a light switch has the two states On and Off
- the Dutch traffic light has the three signals Go, Attention and Stop
- In MS Word, there are several different modes to view a document: Read, Print, Web.
- ...

If you are making a model of an existing system in front of your eyes, that's called *reverse engineering*. One good heuristic to reverse engineer states is to simply observe the different visual forms the display takes. Then, the states are exmained further and are given a description and a label. So, let's reverse engineer the Dutch traffic light by watching one cycle of signals:

Here are the three displays of the traffic light:

```
.    .    R
.    Y    .
G    .    .
```

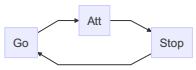
Then we capture the purpose of state and give it What we have to capture in the first place in our analysis is the *purpose of states*:

1. Go: green light indicates that drivers may pass
2. Att(ention): yellow light indicates that drivers should prepare to stop
3. Stop: red loght indicates that drivers must wait

Notice how we named the states of the traffic light, or better how we did not call them Green, Yellow and Red. The description of the state should address the purpose of the state and not its appearance. Visual design comes at a much later point in the development process.

8.1.2 Identifying transitions

The second step in the analysis is about identifying the *transitions* between states. The Dutch traffic light knows three transitions:



Cycling through a set of states is the most simple interaction protocol. It is unique, in that states are arrenged in a closed cycle, which implies that every state appears exactly once on the left-hand side (the original state) and the right-hand side (the destination). As we will later see, this is why we also only need one button for a manual traffic light.

Most of the time it is more complicated and one easily overlooks a possible transition. If the number of states is not overwhelmingly large, transitions can systematically be found using a *transition table*. A transition table is just a square matrix, where you put the states into rows and columns. The rows denote the *origin* state, the columns are *target* states.

```
      Go  Att Stop
Go
Att
Stop
```

According to the empty transition table, the traffic light has nine possible transitions. But which of them make any sense and should be implemented? We start in the first row and ask: "Does it make sense that a state Go is followed by itself?" Here, the answer is No and you leave the cell just empty. But, there are situations, where you would want to make such a "selfish" transition, think of a dia show program, that, whenever a button is pressed, shows the next picture. The state Show_picture is followed by Show_picture. That also implies, that states can be somewhat abstract. It would make little sense to have a separate state defined for every picture in the show.

```
         Welcome  Present Goodbye
Welcome           X
Present           X       X
Goodbye
```

Let's move on: "Does a transition from Go to Att make sense?" Absolutely! You make a cross. Then you do the same for the second and the third row, leaving blank the impossble transitions and marking the possible.

```
      Go  Att Stop
Go          X
Att             X
Stop  X
```

It is important to realize, that transitions are not generally symmetric. In the case of a traffic light, you can go from Go to Att, but there is no transition back. In interaction models other than a uni-directional cycle, transitions can well be symmetric, think of the checkout process on webshops. Usually, they guide you through the process in a number of steps (billing address –> delivery address, credit card number), but allow you to go back to the previous step to do corrections.

	Delivery	Billing	Credit_card
Delivery		x	
Billing	x		x
Credit_card		x	

Notice that in the traffic light example, every state is origin of exactly one transition, whereas in the checkout process there are two transitions originating at Billing. This is called *branching*. If you wonder how the program eventually decides which transition to take, read on about events.

8.1.3 Events: internal conditions and user input

The transition table tells you which of the possible transitions are allowed and should be implemented. It does not yet tell you when this is going to happen, exactly. An *event* is a condition that triggers a transitions. Events can be internal conditions or user input.

Internal conditions do not require user input. The program decides solely by itself, when to transit. This is why we call transitions on internal conditions *automatic transitions*. A typical case of automatic transitions is control by a timer. If we assume that the Dutch traffic light runs automatically, without someone standing next to it (in a yellow vest) and pushing buttons, all transitions must be automatic. Certainly, we do not want the traffic light to change states instantly, but only after a certain time has passed. For the interaction model that means we have to specify the resting time in every state. We do that right on the transition table by replacing the crosses with a brief description of the condition that makes the system leave the origin state:

```
      Go  Att   Stop
Go        20s
Att             5s
Stop  20s
```

In words, the interaction model is expressed like the following:

1. When in state Go, wait 20s, then transit to Att
2. When in Att, wait 5s, then transit to Stop
3. When in Stop, wait 20s, then transit to Go

In psychological experiments, such time-controlled transitions happen a lot. The purpose of some experiments is to investigate subliminal priming, where presnetation time is so short, that people cannot consciously process the stimuls, but there still are subconscious effects. For example, when very briefly seeing a picture of an apple, participants reportedly react faster to the question: "Is this a word? PEAR." In subliminal primning experiments, in order to control cognitive processing time, stimuli are presented for a very brief moment, followed by a visual mask (to erase visual sensory memory). Finally, the response is acquired as a key press (which is user input) and the program proceeds to the next trial.

```
      Stim  Mask   Resp
Stim        100ms
Mask               100ms
Resp  X
```

That brings us to the second form of events: *user input* events. A transition that is triggered by user input is called an interactive transition. Only if there is at least one interactive transition, will we call it an interactive program. In most cases, user input is just pressing a certain key. Let's first look at how a manual traffic light can be specified. Taking another look at the transition table tells us:

- there are three possible transitions
- at every origin state there is only one transition possible

Because there is only one transition possible at every moment, it seems we can go with just one key, let's call it Next.

```
      Go  Att   Stop
Go        Next
Att             Next
Stop  Next
```

In this case, it seems fairly practical to give the traffic light operator a remote control with just one button. But note that for more complex systems that is not a good idea in terms of usability. Why? Because of what is known as mode confusion. The ideal example probably is one of these 1980 clunky digital watches, which not just told you the time, but also let you use stop watches, timers, altitude alarms, step counters and brewing a nice cup of coffee. And all that functionality is controlled by just four flimsy buttons. What these buttons do, changes with the state the watch is in. For example, the lower left button could switch to another time zone in Watch, but do a reset in stop watch mode. The user has to learn from the manual what every button does in every state and has to be completely aware which mode the watch currently is in.

When desgning more complex interactions it is therefore better to use different user inputs (keys presses) for different actions. That does not mean you have to use a totally new control for every cell in the transition table. There can be actions that are very similar across states, such as an action Next or Previous in the checkout process example:

	Delivery	Billing	Credit_card
Delivery		Next	
Billing	Previous		Next
Credit_card		Previous	

Note that we keep the name of the controls rather abstract, denoting the action, not the physical key or mouse button. Like with the analysis of states, we defer the peoblem to a later state. The reason is to not get distracted at this stage, but also it makes the transition table easier to read, and let's us identify actions that could use the same control. How many controls you finally use in your program is a matter of careful choice. However, there is at least one rule: The number of separate controls must be equal to or larger than the maximum number of interactive transitions in any row. According to this rule, the traffic light can have one or more controls, whereas the checkout program requires at least two (Billing).

In the checkout and traffic light examples, there is only one transition per state, but of course, there can be more than one. In the subliminal priming example, the participant's response is whether the shown letter combination is a word, or not. That implies that two controls are needed at that transitions, say the controls Word and Not_word. As the program will continue to state Stim, irrespectively of which of the two keys have been pressed.

	Stim	Mask	Resp
Stim		100ms	
Mask			100ms
Resp	Word or Not_word		

This needs to be considered carefully, because if you end up putting two controls in the same cell it could be that you have not yet discovered all states. In the subliminal priming example, Resp is followed by Stim, irrespectively of user input. But, that is a special case.

The transition table is now complete. It covers states (in rows and columns), transitions (occupied cells) and events (what is in the cell). In the next section, we will build interactive programs by translating each cell in the table into a *transition conditional*.

8.1.4 Exercises

1. german traffic lights differ from Dutch traffic lights in that they light Red and Yellow before switching to Go. Create a transition table.
2. Identify states, events and transitions of a stopwatch program and create a transition table.
3. A smart traffic light switches from Stop to Go automatically, when a car is approaching and the crossroads are clear. Analyze and create a transition table.
4. Review the Stroop program and find the states in the code. Then, take a closer look at where in the code states are used.

8.2 Programming interactive prototypes

One of the worst things that can happen in software development is that a flaw that has been introduced during an early state of planning is discovered when the program is almost done. For that reason, it is good practice in software development to test models and catch flaws as early as possible. So, how can we test our interaction model? Of course, we could just implement the whole program and then see, if it does what was intended. But, if there truly are flaws, a lot of effort would have been wasted and a lot of work had to be redone. This catch-22 situation can be resolved by implementing just the aspect of the system that you want to test, for example the flow of interaction. This is called a *prototype* and, as you will see, the transition table can quickly be translated into a simple piece of software, an *interactive prototype*, using just a few standard coding patterns.

For the interactive prototype, we will turn the transitions from the table into *transition conditionals* or short: *transitionals*. As we have two kinds of transitions, automatic and interactive, we will also have to deal with interactive and automatic transitionals.

8.2.1 Interactive transitionals

The following code snippet implements two interactive transitional with the same origin, namely the transition from Billing to Delivery and Credit_card:

```
print(STATE)
key = input("Press a key: ")
if STATE == "Billing" and key == "p":
  STATE = "Delivery"
if STATE == "Billing" and key == "n":
  STATE = "Credit_card"
```

Notice that

1. the state of a program is maintained as a plain String variable
2. the starting state is set as “Billing”
3. interactive transitionals check for two conditions: current state and whether the respective key has been pressed
4. a transition occurs by setting the variable `STATE` to another value

The above code will only ask for user input once, change the state (or stay in Billing, if no valid input was entered) and terminate. Instead, we want this program to truly flow and be able to react on longer series of user actions. Basically, the only thing we have to do is put the transitionals into a loop. However, in this particular case, what would happen is that the program would never truly stop. For that purpose, we implement a general state called Exit, that ends the program. The transitions of Exit are special, in that this state always is the destination, never the origin. This is why we can omit the row. In many cases you also want to respect the users freedom and allow them to jump to Exit at any time they want:

	Delivery	Billing	Credit_card	Exit
Delivery		Next		X
Billing	Previous		Next	X
Credit_card		Previous		X, Next

The interaction prototype of Checkout looks like the following.

```
STATE = "Delivery"
while True:

  print(STATE)
  key = input("Press a key: ")

  # interactive transitionals

  if STATE == "Delivery" and key == "n":
    STATE = "Billing"
    continue
```

```
  if STATE == "Billing" and key == "p":
    STATE = "Delivery"
    continue

  if STATE == "Billing" and key == "n":
    STATE = "Credit_card"
    continue
  if STATE == "Credit_card" and key == "p":
    STATE = "Billing"
    continue

  if STATE == "Credit_card" and key == "n":
    STATE = "Exit"
    continue

  if key == "x":
    STATE = "Exit"
    continue

  if STATE == "Exit":
    break


exit()
```

Notice that:

1. the loop itself comes with no conditions itself. It just plainly loops, until someone proves that the truth is not true.
2. the print statement at the top of the while loop entirely replaces the graphical user interface at this stage.
3. after every transitional, we have put a `continue` statement. Why? Because, when the State veriable is updated, it can fire up a second transitional. Check yourself, by reading the conditionals: within one iteration we would go from Delivery to Exit all the way through and we would not even see the intermediate steps printed to the console. The `continue` statement forces the loop to begin a new iteration, immediatly, omitting all the transitionals after the one that has fired. That makes sure that per iteration no more than one state transition takes place.
4. We use a transitional that does not ask for the state and therefore always triggers Exit when x is pressed
5. the Credit_card –> Exit transitional uses `continue` in order to reach the `print(STATE)` command one more time. During that final iteration only, the last transitional (the one without destination) is encountered and the `break` statement ends the loop.

In general, interactive transitional emerge from the transition table by the following pattern:

```
if STATE == <origin> and <event>:
  STATE = <state>
  continue
```

Notice that every combination of state and event gets its own `if ... continue` block. We call this *flat transitionals*. Another way is to use *nested transitionals*, which have an an outer `if` block to interrogate the state and uses the inner `if` block to deal with all possible events, The general form is:

```
if STATE == <origin>:
  if   <event_1>:
    STATE = <next_state_1>
  elif <event_2>:
    STATE = <next_state_2>
  elif ...
  continue
```

There is no difference in functionality between these two forms of transitionals, but for longer programs it may be easier to read. With nested transitionals, the checkout program would look like the following:

```
STATE = "Delivery"
while True:

  print(STATE)

  key = input("Press a key: ")
# interactive transitionals
  if STATE == "Delivery" and key == "n":
    STATE = "Billing"
    continue

  if STATE == "Billing":
    if key == "n":
      STATE = "Credit_card"
    if key == "p":
      STATE = "Delivery"
    continue
  if STATE == "Credit_card":
    if key == "n":
```

```
      STATE = "Exit"
    if key == "p":
      STATE = "Billing"
    continue

  if STATE == "Exit":
    break
```

Notice that

1. the indentation visually groups all transitions that belong to the same origin state
2. we reduced the number of `continue` statements. As `key` is never changed inside a transitional, only a single event conditional is triggered at a time.

8.2.2 Automatic transitions

Sometimes we want a transition to happen due to internal conditions of the program, for example:

- stay in state Att, when 5 seconds have passed move on to Stop
- show the stimulus for 100ms, then move on to the distraction mask.
- after 20 trials of an experiment, make a pause.
- after all dias have been shown, give a Goodbye

A frequent case of automatic transitions is staying in a state for a certain amount of time, then move on to the next state. Take as an example a dia show program that automatically shows every picture for, say, 30 seconds. Fortunately, automatic transitions almost look the same, the only difference being that `<event>` becomes `<condition>`.

As you have experienced at the Stroop task theater play from chapter 1 dealing with time requires some extra tricks. For example, how do we let the automatic traffic light rest in the Go state for exactly 20 seconds? We set a timestamp and wait for the difference to become 20 seconds or longer. Fortunately, for interaction prototypes, there is a cheap and dirty trick to do that more straight-forwardly, which is the command `sleep(20)`. It suspends the whole program and wakes it up after twenty seconds. Why is that dirty? For two reasons: First, it only works for interaction prototypes, but not Pygame programs. For example, when you are showing animated stimuli, putting the whole program on halt won't work. Second, you will not be able to bypass the waiting time, say, the user wants to prematurely proceed to the next picture in the dia show.

```
from time import sleep
STATE = "Stop" # Att, Go
while True:
    print(STATE)
```

```
    if STATE == "Go":
      sleep(2)
      STATE = "Att"
      continue

    if STATE == "Att":
      sleep(1)
      STATE = "Stop"
      continue

    if STATE == "Stop":
      sleep(2)
      STATE = "Go"
      continue
```

A program can be (and typically is) a mix of interactive and automatic transitionals. For example, we could create a semi-automatic traffic light, where the operator only has to switch Stop and Go. When set to Stop, the program first goes to state Att, rests there for a few seconds and than automatically moves on to Stop.

```
from time import sleep
STATE = "Stop" # Att, Go
while True:
    print(STATE)
    if STATE == "Go":
      input("Hit Return")
      STATE = "Att"
      continue

    if STATE == "Att":
      sleep(2)
      STATE = "Stop"
      continue

    if STATE == "Stop":
      input("Hit Return")
      STATE = "Go"
      continue
```

Notice that

1. the `input` commands moved into the transitionals as otherwise the loop would prompt for input for automatic transitions, as well, which is just not the idea.

Later, we will use the event handling mechanism of Pygame, which will provide a better solution for mixing interactive and automatic transitionals.

2. we have simplified the event handling. This traffic light needs just one control button to toggle Stop and Go, which here is any key (followed by Return) or just the Return key. That is not a general solution, but only works, because there is just one transition per state.

8.2.3 The timestamp method

As we have said, for a prototype, the `sleep(s)` is fine in most cases to cause a delayed reaction, but not when we start coding with Pygame. Then we have to use the timestamp method, which has little benefits for interactive prototypes. Still, it seems in order to introduce this method on the somewhat cleaner code of the interactive prototypes.

The following code shows, how to implement a time delay by setting a timestamp and comparing the difference with current time in the automatic traffic light example.

```
from time import time
STATE = "Go" # Att, Go
time_stamp = time()
while True:

    current_time = time()

    if STATE == "Go" and current_time - time_stamp >= 2:
      STATE = "Att"
      time_stamp = time()
      print(STATE)
      continue

    if STATE == "Att" and current_time - time_stamp >= 1:
      STATE = "Stop"
      time_stamp = time()
      print(STATE)
      continue

    if STATE == "Stop" and current_time - time_stamp >= 2:
      STATE = "Go"
      time_stamp = time()
      print(STATE)
      continue
```

Notice that

- the second condition of time-controlled transitionals carries the waiting time
- the function `time()` returns the number of seconds since 1 January 1970. That is called the Unix epoch.
- because the starting state Go is time-controlled, we have to set a first timestamp before the `while` loop starts.
- the print statements have been moved into the transitionals. Read more on that below.
- `current_time` is set once at the beginning of every iteration. Read more on that below.

It appears as if going from the `sleep` method to the `time` method is just some superficial change in code. In fact, there is one dramatic change once the program runs, although you may not even realize it on the first glance. With `sleep`, the `while` loop does exactly one iteration per state change, that is one per five seconds (combined sleeping time). When entering a transitional, the sleep is initiated and the program rests. Then it wakes up and finds the `continue`, eventually. In the timestamp version, the program never sleeps. The while loop cycles as fast as it can, which is incredibly fast, at every iteration picking up `current_time` and comparing it to the timestamp. Find out yourself, by putting the `print(STATE)` back to its old place, at the top of the loop.

We will use the exact same method in Pygame. But, here it is, why the method does not work well with the prototype. The system we have chosen was the traffic light. That was on full purpose as it is fully automatic. The problem is the `input` command, as it waits for input. While it does, the loop stands still. In order to combine user input with a fast while loop, we would need another way of handling input events. That is what Pygame will provide.

8.2.4 Presentation conditionals

If you inspect the interactive prototyping code so far, you will notice that all output is produced by a single `print` statement, which resides outside of the transitionals. That is fully on purpose, deriving from a golden rule of user interface development, as we will see later.

The sole purpose of a traffic light is to transit between states and make these states visible to the user In general, interactive programs do more than just transit from state to state. They usually process some sort of data or information. Very often, the user input at one state, affects the display of a subsequent state.

The one It suffices to check the interaction flow, but lacks the ability This is a very poor solution, as it only checks the primary state the program is in. In order to check the logic of the program it is useful to enrich tehz output further. For example, in the traffic light example, we may want to show the tr5affic light, not just its state.

It is tempting to just the print statements into the transitionals, but that will finally not work at all with Pygame programs. Recall what we said in the beginning of this chapter

about breaking a complex problem into manageable pieces. In software development it is accepted best practice to clearly separate two aspects of an interactive program: the flow of states and the presentation on screen. There even is a name for this approach, its called a *two-tier architecture*. While this may seem over done at the toy examples, we use here, but it has significant advantages in the long run:

- You can focus on one thing at a time, typically starting with the flow of the program and once that is done, move on to screen design.
- You can even separate the two tasks: one expert is doing the interaction flow, and another expert creates the screens.
- It works with Pygame.

In order to be more flexible with what the prototype presents to the screen when in a state, we will expand the `print(STATE)` statement into a set of *presentation conditionals (presentitionals)*, allowing us to create different output for different states. We demonstrate the idea by the example of the checkout process.

Notice that:

1. Presentationals *never* change the state as that is the domain of transitionals.
2. presentitionals are triggered by states, simply. If you find yourself wanting to add another condition, say whether the response was correct, that can either go into teh presentitional itself or you can split the state into two separate ones.

3. we use `if` statements without `continue`. If we would do otherwise, the iteration would jump over our carefully crafted transitionals and nothing will ever happen anymore. We also don't need to break the iteration, because the program always is in exactly one state and the presetitionals never change state. That makes sure that exactly one presentitional will fire during a single iteration.
4. presentitionals come before transitionals, as otherwise they would fall prey to the `continue` statement and never show up. With Pygame, due to its better event handler mechanism, this is no longer necessary and it seems more logical to put presentitionals last.

```
STATE = "Delivery"
while True:

  # presentation conditionals

  if STATE == "Billing":
    print("n for proceed")
    billing  = input("Please, enter your billing address")

  if STATE == "Delivery":
    delivery = input("Please, enter your delivery address")
```

```
    print("p for previous, n for next")

  if STATE == "Credit_card":
    print("n for next, p for previous")
    credit_card  = input("Please, enter your billing address")

  if STATE == "Exit":
    print("We received your order")
    break


  key = input("Press a key: ")
  # interactive transitionals
  if STATE == "Delivery" and key == "n":
    STATE = "Billing"
    continue

  if STATE == "Billing":
    if key == "n":
      STATE = "Credit_card"
    elif key == "p":
      STATE = "Delivery"
    continue
  if STATE == "Credit_card":
    if key == "n":
      STATE = "Exit"
    elif key == "p":
      STATE = "Billing"
    continue
```

8.3 Interactions in Pygame

8.3.1 Overall structure

```
BACKGR_COL = col_gray
SCREEN_SIZE = (700, 500)
pygame.init()
pygame.display.set_mode(SCREEN_SIZE)
pygame.display.set_caption("Stroop Test")
screen = pygame.display.get_surface()
font = pygame.font.Font(None, 80)
```

```
font_small = pygame.font.Font(None, 40)
def main():
  # fast while loop
  while True:

          # refreshing the display
          screen.fill(BACKGR_COL)

          # Event loop

              # interactive transitionals

          # automatic transitionals

          # Presentitionals

          # Updating the display
          pygame.display.update()
main()
```

Notice that:

1. The code for setting up the display looks complicated, but it usually suffices to just copy that part of the code whenever we start a new project. Still, you probably want to configure a few aspects to your requirements:
   - choose a background color, e.g. `col_black` or `#FFFFFF` (white)
   - set the width and height of the screen by changing `SCREEN_SIZE`
   - change the caption of the screen (to appear in the top bar of the window)
   - change font and font size
2. the main program is nbot invoked directly, but encapsulated in a function `def main():`. This function is then executed in the very last line of code.

8.3.2 The screen and the refresh loop.

With Pygame you can create interactive programs with fancy geometries and fluent animations. In fact, all computer animations are just pseudo-fluent. It is like in the movies: a fluent movement, like that of a jumping ball, is captured in a high speed. It is just a sequence of still images, but by the sluggish human visual system it is perceived no less fluent than the real scene.

In a Pygame program, an animation is created by a series of stills, too, with the only difference that these stills are not on celluloid. They get computed in an instance and

transfered to the computer screen in fast succession, like 60 per second (60Hz). This happens as followed:

1. setting up the display
2. a fast while loop runs continuously, at every iteration:
3. the screen is refreshed by painting the background color over the previous image
4. the next image is computed by transitionals and presentitionals
5. the display is updated with the new image

A single iteration of the fast while loop we call *a refresh*.

8.3.3 Handling user input

With Pygame you can create interactive programs that react fluently to user input. There no longer is a prompt that waits for input and requires Return to continue. User input in Pygame programs is handled by a very clever mechanism called the *event handler queue*. You don't actually see it, as it is set up behind the scenes by `pygame.init()`. It acts a lot like a descretive secretary: it stays in teh background and collects the incoming messages, hands them over one by one to the executive and lets her do her own thing with it.

While the fast `while` loop runs in the foreground, the event handler is a separate mechanism that continuously listens to input from keyboard or mouse. Whenever something happens, a key is pressed or the mouse is pushed, the event handler registers this event and puts it in a queue. At every regresh this queue is retrieved and emptied by `pygame.event.get()`, which returns the queue as a list. The event `for` loop iterates over this list, pulling out the events one by one out and handing them over to the interactive transitionals.

```
# Event handler
for event in pygame.event.get():
    # interactive transitionals
    if STATE == "welcome":
        if event.type == KEYDOWN and event.key == K_SPACE:
            STATE = "prepare_next_trial"
            print(STATE)
            continue
```

Notice that

1. when there is no input, the event loop never runs and none of the interactive transitionals wil fire. The event handler loop acts like a conditional most of the time. It sits in the fast refresh loop and therefore is idle most of the iterations. That is why you should never put automatic transitionals into the event loop.

2. `pygame.event.get()` returns the event queue, but also empties it. Otherwise, events would be eveluated over and over again.
3. an event is a complex piece of information. For a key press, the type of the event is `KEYDOWN` and it has a property key that holds the key that has been pressed.
4. if more than one event has been recorded, they are all being processed in a single refresh. That can mean more than one transition per refresh.
5. We still need `continue` to make sure that only one transitional per event is fired. While the event handler queue is consumed (emptied) by `pygame.event.get()`, the variable `event` is not. If two successive transitionals use the same event (e.g. n for next), they would both be triggered.

8.3.4 Transitionals

Interactive transitionals differ only slightly. Because of the event handler mechanism they no longer reside under the fast `while`, but one level deeper in the event handler loop. If you use a print statement to see the state transitions on the console, these must go into the individual transitionals, see yourself:

```
# Event loop
for event in pygame.event.get():

    # interactive transitionals
    if STATE == "welcome":
        if event.type == KEYDOWN and event.key == K_SPACE:
            STATE = "prepare_next_trial"
        print(STATE)
        continue

    if STATE == "wait_for_response":
        if event.type == KEYDOWN and event.key in KEYS.values():
            time_when_reacted = time()
            this_reaction_time = time_when_reacted - time_when_presented
            this_correctness = (event.key == KEYS[this_color])
            STATE = "feedback"
        print(STATE)
        continue
    if STATE == "feedback":
        if event.type == KEYDOWN and event.key == K_SPACE:
            if trial_number < n_trials:
                STATE = "prepare_next_trial"
            else:
                STATE = "goodbye"
        print(STATE)
```

```
        continue


    if event.type == QUIT:
        STATE = "quit"
        print(STATE)
        break
```

Notice that:

1. the event condition gets a little more complicated (more on that below) and that lets us prefer nested transitionals
2. the second transitional does a little data processing, namely capturing the response time by the time stamp method.
3. the third transitional contains two transitions, which we call a `branching transitional`. The inner most conditional interrogates the number of trials that have already been presented and either moves on to another trial or teh end of the experiment.
4. the last transitional breaks the event handler loop. If the user wants to quit, he wants to quit and not wait for the event handler loop to finish the queue.

Automatic transitionals in Pygame differ in no way from automatic transitionals in console prototypes, except when you have to use the time_stamp method to cause a delay. They have the same structure and they remain directly under the fast `while` loop.

There is one automatic transitional in the Stroop program, that is a little special on closer examination: With every new trial, the transitional `prepare_trial` is triggered and automatically the program moves on to `show_trial`. On closer examination this transitional is different to the ones we have seen before, because the state `prepare_trial` does not have a presentational. It is completely invisible to the user, a *ghost transitional*. We also notice that there is quite some extra code inside. Some serious data processing is done, before the actual transition.

```
# automatic transitionals
if STATE == "prepare_trial":
        trial_number = trial_number + 1
        this_word  = pick_color()
        this_color = pick_color()
        time_when_presented = time()
        STATE = "show_trial"
        print(STATE)
```

This transition actually prepares all aspects of the stimulus, before it is shown. But, why aren't these computations just be done during a direct transition to `show_trial`?

In fact, for the little program we have here, it does not matter. It is just there to have one automatic transitional.

But, there are situations, where using ghost transitionals can be useful

- when the data processing is complex. Then, it seems more natural to think of the programm being in data processing transition.
- when the data processing lets the program branch. For example, we could define two separate states for trial feedback, for correct and incorrect responses. Then, it seems more natural to put the computations and branching into a "knot" on its own.

IN both cases, the program will be easier to understand and debig, but there is no difference in functionality.

8.3.5 Pygame events

The event handler collects the events between two refreshes and hands it over to the event loop. We have already seen on the transitionals that events are more complex than just literal characters. That makes sense, when you consider that also mouse events are being captured. They cannot be expressed as a characters, but as either coordinates (when the mouse is moved) or as button presses at a certain position on the screen.

But let's look at keyboard input, first, as they appear in the Stroop program. They are all have the same in structure in their inner event conditions:

```
if event.type == KEYDOWN and event.key == K_<key>:
```

So, this conditional asks for the type of the event, which is that a key has been pressed down. Note that this is more basic than that a key has been entered, as that would also mean it has been released. The secod condition is about the key. Different to our prototypes, this is not a literal character, but a constant, that Pygame has predefined. That helps to deal with keys that do not have a character assigned, such as Return, Alt, Ctrl and the arrows on your keyboard. The full list of key definitions can be found in the [Pygame documentation]:(https://www.pygame.org/docs/ref/key.html).

When theer is a `KEYDOWN` there must also be a `KEYUP` event type, that is recorded, when the key is released again. That can be quite useful, when the goal is to do something as long as the key is pressed. Just try out what the key arrow_up does when you are editing a document. It moves the cursor up the lines until you release it.

The full list of event types is given in the [Pygame documentation]:(https://www.pygame.org/docs/ref/event.html). Here we will only briefly explain the most important, which is the event type QUIT and all mouse event types:

```
if event.type == QUIT:
  STATE = "quit"
  break
```

This actually corresponds with the alltime exit that we once implemented in the prototype. Usually, the QUIT event is the same as a key press on the ESC key. And if you try it out, you will see that you can cancel the Stroop program at any time pressing ESC. It is no more than a short cut.

Mouse events come as several types with different properties, the two most important ones are:

- a MOUSEMOTION event is recorded when the mouse pointer has moved during the refreshes. It has an attribute `event.pos`, which is a list of two numbers, the x and y coordinates of the new pointer position.
- MOUSEBUTTONDOWN is recorded when a mouse button is pressed down. The attribute event.button identifies which button it was (LEFT, RIGHT).

See these two events in action in the following little program:

```
import pygame
import sys
from time import time
from pygame.locals import *
import random
from pygame.compat import unichr_, unicode_
##### VARIABLES #####
# Colors
col_black = (0, 0, 0)
col_green = (0, 255, 0)
col_green_dim = (0, 60, 0)
BACKGR_COL = col_black
SCREEN_SIZE = (500, 500)
pygame.init()
pygame.display.set_mode(SCREEN_SIZE)
pygame.display.set_caption("Mouse events")
screen = pygame.display.get_surface()
screen.fill(BACKGR_COL)
def main():
    STATE = "not_clicked"
    mousex = 0
    mousey = 0
    while True:
        screen.fill(BACKGR_COL)
```

```
        for event in pygame.event.get():
            # IT
            if STATE == "not_clicked":
                # mouse moved, new pointer position is stored
                if event.type == MOUSEMOTION:
                    mousex = event.pos[0]
                    mousey = event.pos[1]
                    continue
                # left mouse button clicked
                if event.type == MOUSEBUTTONDOWN and event.button == LEFT:
                    STATE = "clicked"
                    time_when_clicked = time() # timestamp for "unclicking"
                    continue
            if event.type == QUIT:
                pygame.quit()
                sys.exit()
        # AT
        # unclicking after 500ms
        if STATE == "clicked" and time() - time_when_clicked > 0.5:
            STATE = "not_clicked"
        #  Presentitionals
        if STATE == "clicked":
            draw(mousex, mousey, True)

        if STATE == "not_clicked":
            draw(mousex, mousey, False)

        pygame.display.update()
def draw(posx, posy, clicked):
    if clicked:
        color = col_green
    else:
        color = col_green_dim
    text_surface = font.render(str(int(posx)) + ':' + str(int(posy)), True, color, 
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,SCREEN_SIZE[1]/2.0)
    screen.blit(text_surface, text_rectangle)
main()
```

Notice that

1. `event.pos` is a list of two values, the x and y coordinates
2. Pygame provides the mouse buttons as constants, which are not to be put in quotation marks.

3. it is unsafe to create new variables inside an event, as you never know when it happens. It is better to explicitly create variables (mousex, mousey) at the top of the main program
4. there are two presentitionals, but they both use the same function to do the drawing

8.4 Presentitionals in Pygame

And, here we're getting to the candy! With Pygame presentitionals we give our program a face. The presentitionals of Pygame have the same formal structure as the prototype presentitionals, so there is nothing more to say about it. This chapter shows you how to create graphical user interface that render stimuli, show pictures, play animations and have buttons.

The Pygame system has been developed for arcade games and is very powerful when it comes to creating and transforming graphical objects. Unfortunately, that also makes it very complex, which is why the functions have so many dots in their names. Fortunately, many psychological experiments are at most as complex as the simplest arcade game. Introducing a small subset of Pygame functionality is sufficient. For when you are a geek take a look at other books on game development with Python.

Recall that at every refresh the screen is almost literally painted over. In fact, every one presentitional can be considered a drawing on its very own. That allows us to demonstrate the elements of Pygame graphics in simple programs, that produce a still picture. That also matches the way you should proceed, when writing the transitionals: first you write the scaffolding code, which consists of all the presentitional and their empty draw function. Then you grab the draw functions one by one and program the visuals, using text, geometric figures and pictures. Here is how the scaffolding looks like:

```
if state == "some_state":
  draw_some_state()

if state == "some_other_state":
  draw_some_state(some_property)

pygame.display.update()

def draw_some_state():
  pass

def draw_some_other_state(some_property):
  pass
```

Notice that:

1. we do *not* use print statements in presentitionals, as they are in the fast loop.
2. often, it is useful to make the draw function more generic and use function arguments to fix some properties.

8.4.1 Display and surface

Creating the appearence of states means to draw things to the surface (also known as screen or canvas). However, before we can do so, the surface itself must be created. Also, we have to understand the "physical properties" of the surface.

Like almost any other program, a Pygame program appears as a window, just like the one in which this text is displayed, be it a browser or a pdf viewer. This windows is composed of the frame, which consists of a rectangular edge, a title bar with a caption and the Windows controls |_|| ||X| in the upper right corner. Between the edge sits the surface. The first thing that needs to be done is creating this window, which is called the *display* in Pygame terminology. The following code creates a minimal Pygame display with a surface that is 600 pixel wide and 400 pixel high and puts "A Pygame display" into the title bar. The only thing it can do is quit properly.

```
# Required libraries
import pygame
import sys
from pygame.locals import *
from pygame.compat import unichr_, unicode_
# Initializing the display
pygame.init()
pygame.display.set_mode((600, 400))
pygame.display.set_caption("A Pygame display")
# Creating the surface
surface = pygame.display.get_surface()
surface.fill((0,0,0))
def main():
    while True:
        for event in pygame.event.get():
            # IT
            if event.type == QUIT:
                pygame.quit()
                sys.exit()
        pygame.display.update()
main()
```

Notice that:

1. When initializing the display, there is no assignment operator involved. There is only one display object `pygame.display` which magically appears by initializing pygame. However, to some extent you can configure it to your liking.
2. As we do it here, the display is set to a fixed size and you won't be able to resize the window during runtime. There is a way to make it resizeable, but that would complicate matters when it comes to drawing to the surface.
3. The surface is an object that is created by `pygame.display.get_surface()` and assigned the name `surface` here. Thus, you can give it another name and, in fact, we call it "screen" or "canvas" in other examples of this book.
4. With `surface.fill()` you give the surface a color. For more on colors, read on.
5. It is the display that is updated (refreshed), not the surface, which is just a little counter-intuitive, as the surface is where we draw.
6. Although the program does nithing noteworthy, we included the event handler with a quit transition. If we do not, closing the window becomes a surprisingly difficult task.

There is more to say about the surface than you can actually see in this example. One thing to notice is that the size of the surface is given with *pixel* as a unit. What is a pixel? Take a loupe and look at your computer monitor closely. You will see tiny dots, each of which is composed of three differently colored areas. These dots are the pixel and everything you draw or render to the screen is actually composed of pixel. It seems natural to use pixel as a unit of measurement when programming graphics, as long as we precisely know the size of the surface. One reason for not using metric (mm, cm) as units is that pixel size can differ a lot between monitors. Use the same loupe on teh display of your smartphone to check this yourself. This is why, here, we always work with fixed size displays.

Objects are placed on the canvas by *coordinates*, and these are given in pixel, too. While we are used to coordinate systems that have the origin (x = 0, y = 0) in the lower left corner, Pygame surfaces have the *origin (0,0) in the upper left corner*. The historical reason is that the old clunky electron ray tubes start a new refresh in the first line, from left-to-right. Perhaps, it is easier to remember that it is like writing a text.

By `pygame.update.display()` the refresh loop updates the display at every iteration. This update collects all the graphical elements that have been produced during that iteration and puts them to the surface. However, speaking figuratively this is *unlike* grabbing a fresh canvas and do the drawing. It is much like how the old masters worked with oil on canvas. New or updated elements are over-painted. Areas that are not affected by over-painting stay just as they are. In computer games that perfectly makes sense. When navigating a character, say the famous Pacman, the maze is constant and it makes little sense to update it every 60th of a second. In the interactive programs we are dealing with, here, the interface layout changes more often and, in order to play it safe, we always do the `surface.fill()` in order to create a blank sheet at every refresh.

Speaking of *colors*: after creating the surface, we filled it with our background color of choice, which is the tupel `(0, 0, 0)`. Why these three numbers? Remember what

you saw through the loop: every pixel is composed of three subpixel of different colors, namely: Red, Green and Blue. That is how every computer in the world composes colors: as an *additive mixture* of these three colors. In short, we call it the *RGB system*. Every subpixel has a fixed color, but the brightness of the color can vary, taking values from 0 to 255 (which happens to be called a byte). For a few examples, check out the Stroop program, where a number of colors have been predefined as tupels. So, `(0, 0, 0)` means that all three color channels are dark and the results is Black. On the very opposite `(255, 255, 255)` means all channels fire at maximum brightness, which results in White.

8.4.2 Drawing figures

```
import pygame
import sys
from pygame.locals import *
from pygame.compat import unichr_, unicode_
pygame.init()
width = 1000
height = 800
pygame.display.set_mode((width, height))
pygame.display.set_caption("Drawing figures")
screen = pygame.display.get_surface()
screen.fill((0,0,0))
def main():
    while True:
        screen.fill((0,0,0))
        for event in pygame.event.get():
            if event.type == QUIT:
                pygame.quit()
                sys.exit()

        draw_circ(250, 200, 200, (255, 255, 255), 5)
        draw_circ(250, 200, 150, (255, 255, 255), 20)
        draw_circ(250, 650, 100, (255, 0, 0), 0)

        draw_rect(500, 50,  400, 600, (0, 0, 255), 0)
        draw_rect(500, 50,  200, 200, (0, 255, 0), 10)

        draw_tria(250, 200, 250, 650, 500, 50)

        pygame.display.update()
def draw_circ(x, y, radius,
```

```
                color = (255,255,255),
                stroke_size = 1):
    pygame.draw.circle(screen, color,
                       (x,y), radius, stroke_size)
def draw_rect(x, y,
                width, height,
                color = (255,255,255),
                stroke_size = 1):
    pygame.draw.rect(screen, color, (x, y, width, height), stroke_size)
    pass
def draw_tria(x_1, y_1,
                x_2, y_2,
                x_3, y_3, color = (255,255,255),
                stroke_size = 1):
    points = ((x_1, y_1), (x_2, y_2), (x_3, y_3))
    pygame.draw.polygon(screen, color, points, stroke_size)
    pass
main()
```

Notice that:

1. The order of drawing commands determines how figures are painted over one another.
2. Sometimes it is useful to write your own wrapper functions for Psygame commands. Here, it is mainly for the purpose of creating a matrix-like appearance by function names of equal length.
3. Even static programs should have a minimal event loop that reacts properly when the user wants to quit the program.

8.4.3 Placing text and pictures

Putting text on the screen is something that we experience everyday, when we read the internet, write emails or write code. One might think that this is as straight-forward in Pygame as creating figures. It is not that easy, though, and in order to understand why these complications arise, think about the following: When you want to write on a note "DON'T PANIC!" with a surrounding box. What do you do first, the writing or the box?

You put the writing first, because only after you have written do you see how high and wide the written text is. And the same problem happens when rendering text on Pygame surfaces. Fonts differ and even individual characters differ. There is no more reasonable way, to determine the size of the text box, after it has been rendered. To solve the problem, we divide the drawing process in two parts: the *rendering* part creates

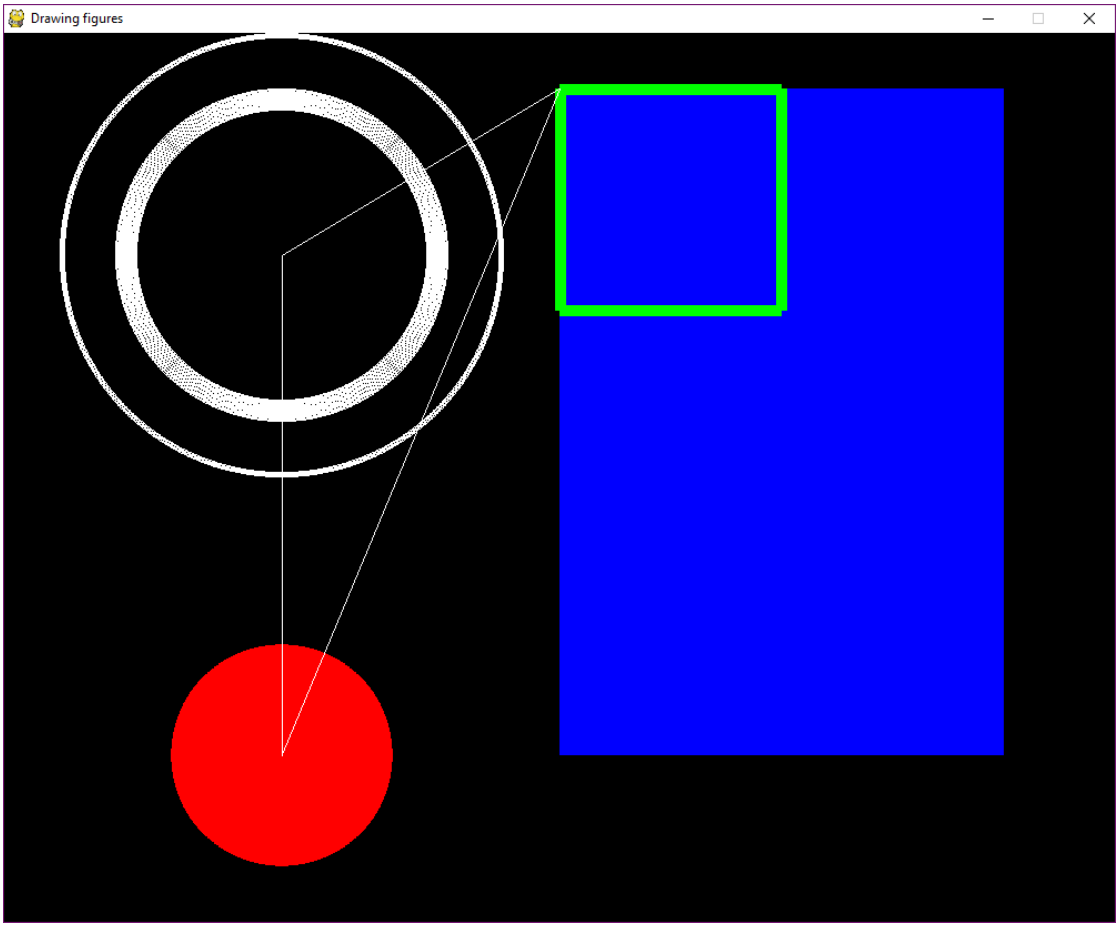


Figure 8.1: Drawing_figures

an object of the text box in the computer memory, without sending it to the surface, yet. From this object, we can retrieve the properties of the text box, most notably its dimensions. Even better, this object can be manipulated in many ways, for example resizing or moving it Pygame doc. Only after that, the text object is send to the surface.

In the following program we merged both steps into a more programmer-friendly function that almost behaves like the figure drawing commands. The only difference: you don't know in advance how large the textbox will be.

```
import pygame
import sys
from pygame.locals import *
from pygame.compat import unichr_, unicode_
pygame.init()
width = 1000
height = 800
pygame.display.set_mode((width, height))
pygame.display.set_caption("Drawing figures")
FONT = pygame.font.Font('freesansbold.ttf',40)
SURF = pygame.display.get_surface()
SURF.fill((0,0,0))
def main():
    while True:
        SURF.fill((0,0,0))
        for event in pygame.event.get():
            if event.type == QUIT:
                pygame.quit()
                sys.exit()

        draw_text(x = 0, y = 0, text = "Pygame says:")
        draw_text(x = width/2, y = height/2, text = "Hi there!",
                  center = True)

        pygame.display.update()
def draw_text(x, y, text,
              color = (255, 255, 255),
              center = False):
    rendered_text = FONT.render(text, True, color)
    # retrieving the abstract rectangle of the text box
    box = rendered_text.get_rect()
    # this sets the x and why coordinates
    if center:
        box.center = (x,y)
    else:
```

```
        box.topleft = (x,y)
    # This puts a pre-rendered object to the screen
    SURF.blit(rendered_text, box)

main()
```

Notice that:

1. First, we actually have to select a font for the text. Here. this is set globally.
2. The coordinates of the function refer to the upper left corner by default, but can be set to refer to teh center of the box.
3. The `SURF.blit()` command sends the rendered text and the textbox to the surface.
4. The box is first retrieved from the rendered text, then moved and finally send to the surface.
5. The function `draw_textbox()` uses `pygame.transform.smoothscale()` to force the text into a given width and height. The results don't neccessarily look good.
6. We have consistently put all global variables into capital letters to avoid confusion between globalö and local variables.

Working with pictures is very similar to text boxes. Again, you only know the size once you have loaded the file from your hard drive.

```
def main():
    while True:
        SURF.fill((0,0,0))
        for event in pygame.event.get():
            if event.type == QUIT:
                pygame.quit()
                sys.exit()

        draw_picture(x = WIDTH/2, y = HEIGHT/2, file = "Beach.png",
                     center = True,
                     scale = 0.5)
        pygame.display.update()
def draw_picture(x, y, file, scale = 1, center = False):
    picture = pygame.image.load(file)
    # retrieving the box
    box = picture.get_rect()
    # transformation
    if scale != 1:
        new_width = int(box.width * scale)
        new_height = int(box.height * scale)
        picture = pygame.transform.smoothscale(picture,
                                               (new_width, new_height))
```

```
        box = picture.get_rect()
    # getting the new box
    if center:
        box.center = (x,y)
    else:
        box.topleft = (x,y)
    SURF.blit(picture, box)

main()
```

Notice that

1. the picture file resides in teh same directory as the program. If that is different you have to use relative paths. Avoid absolute paths, as this makes the program less portable.
2. in the function definition, we put the scale arguent at the end, although an earlier position would be more logical. The reason is that all argument without a default have to come first, such as file.
3. `picture.get_rect()` is called twice. After the scaling the box gets an update
4. Putting the scaling part into a conditional is not necessary, but saves computation where it is not needed.

8.5 Exercises

8.5.1 Exercise C. Transition tables

For the following code snippet, make a transition table.

```
STATE = "A"
while True:
    #ITC
    for event in pygame.event.get():
        if STATE == "A":
            count = 0
            if event.type == KEYDOWN and event.key == K_w:
                STATE = "B"
                print(STATE)
        elif STATE == "B":
            count = count + 1
            if event.type == KEYDOWN and event.key == K_a:
                timer = time()
                STATE = "C"
```

```
                print(STATE)
        elif STATE == "D":
            if event.type == KEYDOWN and event.key == K_SPACE:
                if count < 10:
                    STATE = "B"
                else:
                    STATE = "quit"
                print(STATE)
    #ATC
    if STATE == "C":
        present_picture()
        if time() - timer > 3:
            STATE = "D"
        print(STATE)
    elif STATE == "quit":
        pygame.quit()
        sys.exit()
```

8.6 Common errors

1. You forgot to add `continue` transitionals.
2. You used drawing functions inside the event handler loop.
3. You used the wait command in a Pygame program.

8.7 Think Further

9 Solutions

9.1 Chapter 1. Introduction

9.2 Chapter 2. Variables, values and types

9.2.1 Exercise 1. Operators

```
x = 2
y = 1.0
print(x + y)
```

```
## 3.0
```

```
x = 4.0
y = x + 1
print(x + y)
```

```
## 9.0
```

```
x = 1
y = "1.0"
print(x + y)
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
x = 4
y = 4.0
print(x == y)
```

```
## True
```

```
x = 12
y = x / 2
print(y >= 5)
```

```
## True
```

9.2.2 Exercise 2. Values

Statements 2 and 4 are true

- Each value exclusively belongs to one data type
- Variables can be assigned any type of value

9.2.3 Exercise 3. Mini programs

```
x = "1.0"
y = "1"
print(x + y)
```

```
## 1.01
```

```
x = 5
y = 2
print((x / y) == 2.5)
```

```
## False
```

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

```
## 0
```

9.2.4 Exercise 4. Debugging

What goes wrong in the following code snippets?

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

- An illegal variable name is used

```
x = 3
y = "3"
print(x = y)
```

- The assignment operator is used instead of a comparative operator

```
name = "Colin"
print(This is + name)
```

- The content inside the brackets of the `print` statement lacks quotation marks

9.2.5 Exercise 5. String concatenation

1.

- The statement 'Veni, vidi, vici.' was coined by Gaius Julius Caesar.
- The statement 'Veni, vidi, vici.', 'I came, I saw, I conquered.', was coined by Gaius Julius Caesar.
- The statement 'Veni, vidi, vici.' was coined by Gaius Julius Caesar, supposedly around 47 BC. *Alternatively*, it is also correct to say that the execution stops before the print statement due to the errornous string concatenation in the line before.

2. The `+` operator joins two strings, resulting in one string. The string at the left side of the operator precedes the string at the right side of the operator in the resulting string.

3. `myString = myString + s1 + s2 + year + s3`: the variable `year`is of type `int` which causes a TypeError during concatenation; integer objects and string objects cannot be concatenated. The error can be solved by either explicit typecasting or changing the variable `year` into a string to start with.

```
myString = myString + s1 + s2 + str(year) + s3
year = "47"
```

9.2.6 Exercise 6. Variable names

- `2ndMan`: the variable name starts with a digit
- `m@n3`: the `@` signs is an illegal character
- `man 5`: the space between `man`and `5` is illegal
- `man_no._7`: the fullstop is illegal. Python will think that `man_no` is a class (which it is not by default) and therefore throw a NameError
- `8thMan`: starting with a digit

9.2.7 Exercise 7. Stroop task welcome message

Note: The below description of what the specified code does is not expected from students. The intention is that they think about what happens internally in PyGame when implementing a task such as 'printing something to the screen'.

```
text_surface = font.render(msgText,True,msgColor,BACKGR_COL)
text_rectangle = text_surface.get_rect()
text_rectangle.center = (SCREEN_SIZE[0]/2.0,225)
screen.blit(text_surface,text_rectangle)
```

- Line 1 initializes a new Surface object with the pre-defined message `msgText` and color `msgColor` rendered on it and assigns it to the variable `text_surface`. PyGame draws text on a new Surface to create an image (Surface) of the text, then blit this image onto another Surface.
- Lines 2 and 3 transform the surface to rectangular shape and specify the center of this rectangular shape in relation to the screen size of the program window.
- Line 4 copies the text Surface and position to the screen Surface, making it visible.

For further clarification, consult the PyGame documentation.

9.2.8 Exercise 8. Printing

```
participants = 52
trials = 200
experimental_sessions = 3
trials_pp = trials * experimental_sessions
conditions = 4
condition1 = "easy/limited"
condition2 = "easy/unlimited"
condition3 = "difficult/limited"
condition4 = "difficult/unlimited"
print "In total,", participants, "participants participated in the study."
```

```
## In total, 52 participants participated in the study.
```

```
print "A 2x2 factorial between-subjects design was employed."
```

```
## A 2x2 factorial between-subjects design was employed.
```

```
print "The study examined the interaction of two independent variables: "
```

```
## The study examined the interaction of two independent variables:
```

```
print "task difficulty (easy, difficult) and time (limited, unlimited)."
```

```
## task difficulty (easy, difficult) and time (limited, unlimited).
```

```
print conditions, "conditions were devised, plus a control condition."
```

```
## 4 conditions were devised, plus a control condition.
```

```
print "The conditions were:",condition1,",",condition2,",",condition3,"and",condition4
```

```
## The conditions were: easy/limited , easy/unlimited , difficult/limited and difficult/unlimi
```

```
print "Participants were tested in", experimental_sessions, "experimental sessions."
```

```
## Participants were tested in 3 experimental sessions.
```

```
print "Each session consisted of",trials,"trials."
```

```
## Each session consisted of 200 trials.
```

```
print "In total, each partcipant thus completed", trials_pp, "trials."
```

```
## In total, each partcipant thus completed 600 trials.
```

2. Initially, the variable `trials`is a string. Multiplying the variable by three thus results in a string consisting of three times the value of `trials`. This can be fixed either by initializing `trials` as an integer or using explicit typecasting.

```
trials_pp = int(trials) * experimental_sessions
```

9.2.9 Exercise 9. Using Python as calculator

```
n1 = 37
n2 = 456
n3 = 1027%n1
n2 = n2/n3
n2 += 4
n4 = n2%5
n4 -= 17
n4 = 65%n4/float(2)
print n4
```

```
## -1.5
```

9.2.10 Exercise 10. A Boolean puzzle

```
n1 = 238
n2 = 17
print n1 > n2
```

```
## True
```

```
print n1/17 == 14
```

```
## True
```

```
print n1*n2/float(n1) == n2
```

```
## True
```

```
print n1+(-n1) == n2 - n2 and n1+(-n1) == 972%243 and n2-n2 == 0
```

```
## True
```

```
print n2*(n1*47/n2) == n1*47
```

```
## True
```

```
print n1/4 == n2/1*3 or n1/4 == n2/17*59
```

```
## True
```

Hardcoded results are also valid, for example

```
print n1+(-238) == n2 - 17 and n1+(-238) == 972%243 and n2-17 == 0
```

Only for `n1*n2/float(n1) == n2` hardcoding will not work.

9.3 Chapter 3. Conditionals

9.3.1 Exercise 1. Boolean logic

What is the output of the following code snippets?

```
a = True
b = False
print(a and b)
```

```
## False
```

```
a = True
b = False
print(not a and b)
```

```
## False
```

```
a = True
b = False
print((not a) or b)
```

```
## False
```

```
a = True
b = False
print(not (a and b))
```

```
## True
```

```
a = True
b = False
print(not (a or b))
```

```
## False
```

```
a = 1
b = 2
c = 4
print(a >= b and c > a + b)
```

```
## False
```

```
a = 1
b = 2
c = 3
print(b % a >= c - (a + b))
```

```
## True
```

9.3.2 Exercise 2. State charts

```
# Variables
STATE = "A"
number_of_trials = 0
while True:
  #ITC
  for event in pygame.event.get():
      if STATE == "A":
          number_of_trials = number_of_trials + 1
          if event.type==KEYDOWN and event.key == K_SPACE:
              STATE = "C"
      elif STATE == "C":
```

```
            if event.type==KEYDOWN and event.key == K_A or event.key == K_D:
                arrival_at_B = time()
                STATE = "B"

    #ATC
    if STATE == "B":
        if time() - arrival_at_B > 3:
            if number_of_trials < 15:
                STATE = "A"
            else:
                pygame.quit()
                sys.exit()
```

9.3.3 Exercise 3. Transition tables

For the following code snippet, make a transition table.

```
STATE = "A"
while True:
    #ITC
    for event in pygame.event.get():
        if STATE == "A":
            count = 0
            if event.type == KEYDOWN and event.key == K_w:
                STATE = "B"
                print(STATE)
        elif STATE == "B":
            count = count + 1
            if event.type == KEYDOWN and event.key == K_a:
                timer = time()
                STATE = "C"
                print(STATE)
        elif STATE == "D":
            if event.type == KEYDOWN and event.key == K_SPACE:
                if count < 10:
                    STATE = "B"
                else:
                    STATE = "quit"
                print(STATE)
    #ATC
    if STATE == "C":
        present_picture()
```

```
        if time() - timer > 3:
            STATE = "D"
        print(STATE)
    elif STATE == "quit":
        pygame.quit()
        sys.exit()
```

		A	B	C	D	quit
FROM	A		w			
	B			a		
	C				after 3s	
	D	SPACE AND count $<$ 10				SPACE AND count $>=$ 10
	quit					

9.3.4 Exercise 4. Mini programs

What is the output of the following mini programs?

```
x = 5
y = 0
if x >= 5 and y != False:
    print(y/x)
```

- Nothing

```
x = 12
if x%3 == 0:
    print("x is divisible by 3")
elif x%4 == 0:
    print("x is divisible by 4")
```

- `x is divisible by 3`

```
x = 2
y = 0
if y + 1 == x or not x * 1 <= y:
   print(x + y)
```

- 2

9.3.5 Exercise 5. Following the control flow

```
myString = "Hello, World!"
if len(myString) >= 13 or "ello" in myString:
    myString = "Hi, programming aspirant!"
elif len(myString) <= 12 and "ello" in myString:
    myString = "Hello, from the other side."
print myString
```

```
## Hi, programming aspirant!
```

```
if "Hi" in myString:
    if len(myString) < 25:
        myString = "Wow, my computer seems to answer!"
    elif len(myString) > 25:
        myString = myString + " -- Your computer"
    else:
        myString = myString + " How are you?"
else:
    if len(myString) <= 29:
        myString = "How are you, my computer?"
    elif len(myString) == 27 and "Hello" in myString:
        myString = myString + " I must have called a thousand times."
    else:
        myString = myString + " -- Adele"
print myString
```

```
## Hi, programming aspirant! How are you?
```

```
if "computer" in myString or len(myString) == 38 or "HI" in myString:
    myString = myString + " I myself am doing fine."
else:
    myString = myString + " I am doing just fine."
print myString
```

```
## Hi, programming aspirant! How are you? I myself am doing fine.
```

9.3.6 Exercise 6. Indentation

```
myNumber = 4
if myNumber < 20 and myNumber > 0:
    if myNumber > 0 and myNumber < 15:
        myNumber = myNumber + 3
    else:
        myNumber = myNumber - 3
        print "This is a dead end statement."
elif myNumber > 20:
    print "This should only be printed if myNumber exceeds 20!"
else:
    myNumber = 17
```

9.3.7 Exercise 7. Pseudo code conditionals

```
1.
If not rain
    if Jan has time
        do swimming
    else
        do hiking
else
    do read

2.
if age >= 18 and not alcohol abuse
    set participation to True

3.
if rain
    set wet to True

4.
if eligible
    do briefing
    if condition equals A
        do lead to 001
    else
        do lead to 002
else
    do explain
```

Number 3 is an example of the infamous *Modus Ponens*, where a condition a *implies* condition b. Whenever a is `True`, b must also be `True`. Why is it so famous? Two reasons:

1. In mathematical proofs, *a* is called the *sufficient condition* for *b*, meaning that when a is True, b must be True as well. However, that does not exclude the possibility that b becomes True due to another condition c. For example:

   > When it rains the streets are wet

   does not exclude the possibility that the street is wet because it has been washed with water.

2. In psychology of reasoning it has been shown that people have difficulties with checking the modus ponens. When asked to verify the above example, most people would first check whether the street is wet, although the most relevant question is whether it is raining, indeed.

9.3.8 Exercise 8. Modify Stroop Task

```
import pygame
import sys
from time import time
import random
from pygame.locals import *
from pygame.compat import unichr_, unicode_
##### VARIABLES #####
# Colors
col_white = (250, 250, 250)
col_black = (0, 0, 0)
col_gray = (220, 220, 220)
col_red = (250, 0, 0)
col_green = (0, 200, 0)
col_blue = (0, 0, 250)
col_yellow = (250,250,0)
NTRIALS = 5
WORDS    = ("red", "green", "blue")
COLORS   = {"red": col_red,
            "green": col_green,
            "blue": col_blue}
KEYS     = {"red": K_b,
            "green": K_n,
            "blue": K_m}
BACKGR_COL = col_gray
```

```
SCREEN_SIZE = (700, 500)
pygame.init()
pygame.display.set_mode(SCREEN_SIZE)
pygame.display.set_caption("Stroop Test")
screen = pygame.display.get_surface()
screen.fill(BACKGR_COL)
font = pygame.font.Font(None, 80)
font_small = pygame.font.Font(None, 40)
def main():
    """ Start the Stroop task.
    """
    ## Variables
    STATE = "welcome"
    trial_number = 0
    # for gathering the response times
    RT = []
    while True:
        pygame.display.get_surface().fill(BACKGR_COL)
        # Changing states by user input
        for event in pygame.event.get():
            # welcome screen --> prepare next trial (space bar)
            if STATE == "welcome":
                if event.type == KEYDOWN and event.key == K_SPACE:
                    STATE = "prepare_next_trial"
                    print(STATE)
            # wait for response --> feedback (b, n, m)
            elif STATE == "wait_for_response":
                if event.type == KEYDOWN and event.key in KEYS.values():
                    # remember when the user has reacted
                    time_when_reacted = time()
                    # calculate the response time
                    this_reaction_time = time_when_reacted - time_when_presented
                    RT.append(this_reaction_time)
                    # was the response correct?
                    this_correctness = (event.key == KEYS[this_color])
                    STATE = "feedback"
                    print(STATE)
            if event.type == QUIT:
                STATE = "quit"
        # automatic state transitions
        # prepare next trial --> wait for response (immediatly)
        if STATE == "prepare_next_trial":
            trial_number = trial_number + 1
```

```
            # randomly pick word and color
            this_word  = pick_color()
            this_color = pick_color()
            # remember when stimulus was presented
            time_when_presented = time()
            STATE = "wait_for_response"
            print(STATE)
        # show feedback, then advance to next trial or goodbye  (for 1s)
        if STATE == "feedback" and (time() - time_when_reacted) > 1:
            if trial_number < NTRIALS:
                STATE = "prepare_next_trial"
            else:
                STATE = "goodbye"
            print(STATE)
        # Drawing to the screen
        if STATE == "welcome":
            draw_welcome()
            draw_button(SCREEN_SIZE[0]*1/4, 450, "Red: B", col_red)
            draw_button(SCREEN_SIZE[0]*2/4, 450, "Green: N", col_green)
            draw_button(SCREEN_SIZE[0]*3/4, 450, "Blue: M", col_blue)

        if STATE == "wait_for_response":
            draw_stimulus(this_color, this_word)
            draw_button(SCREEN_SIZE[0]*1/4, 450, "Red: B", col_red)
            draw_button(SCREEN_SIZE[0]*2/4, 450, "Green: N", col_green)
            draw_button(SCREEN_SIZE[0]*3/4, 450, "Blue: M", col_blue)

        if STATE == "feedback":
            draw_feedback(this_correctness, this_reaction_time)

        if STATE == "goodbye":
            draw_goodbye()

        if STATE == "quit":
            pygame.quit()
            sys.exit()
        pygame.display.update()

def pick_color():
    """ Return a random word.
    """
    random_number = random.randint(0,2)
    return WORDS[random_number]
```

```
def draw_button(xpos, ypos, label, color):
    text = font_small.render(label, True, color, BACKGR_COL)
    text_rectangle = text.get_rect()
    text_rectangle.center = (xpos, ypos)
    screen.blit(text, text_rectangle)
def draw_welcome():
    text_surface = font.render("STROOP Experiment", True, col_black, BACKGR_COL)
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,150)
    screen.blit(text_surface, text_rectangle)
    text_surface = font_small.render("Press Spacebar to continue", True, col_black,
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,300)
    screen.blit(text_surface, text_rectangle)
def draw_stimulus(color, word):
    text_surface = font.render(word, True, COLORS[color], col_gray)
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,150)
    screen.blit(text_surface, text_rectangle)
def draw_feedback(correct, reaction_time):
    if correct:
        text_surface = font_small.render("correct", True, col_black, BACKGR_COL)
        text_rectangle = text_surface.get_rect()
        text_rectangle.center = (SCREEN_SIZE[0]/2.0,150)
        screen.blit(text_surface, text_rectangle)
        text_surface = font_small.render(str(int(reaction_time * 1000)) + "ms", Tru
        text_rectangle = text_surface.get_rect()
        text_rectangle.center = (SCREEN_SIZE[0]/2.0,200)
        screen.blit(text_surface, text_rectangle)
        if reaction_time > 5:
            text_surface = font_small.render("Come on, you can be faster!", True, c
            text_rectangle = text_surface.get_rect()
            text_rectangle.center = (SCREEN_SIZE[0]/2.0,250)
            screen.blit(text_surface, text_rectangle)

    else:
        text_surface = font_small.render("Wrong key!", True, col_red, BACKGR_COL)
        text_rectangle = text_surface.get_rect()
        text_rectangle.center = (SCREEN_SIZE[0]/2.0,150)
        screen.blit(text_surface, text_rectangle)
        #text_surface = font_small.render("Press Spacebar to continue", True, col_b
def draw_goodbye():
    text_surface = font_small.render("END OF THE EXPERIMENT", True, col_black, BACK
```

```
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,150)
    screen.blit(text_surface, text_rectangle)
    text_surface = font_small.render("Close the application.", True, col_black, BACKGR_COL)
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,200)
    screen.blit(text_surface, text_rectangle)

main()
```

9.3.9 Exercise 9. Simplify nested conditionals

```
# participant details #
age = 20
gender = "Male"
study = "Psychology"
speaks_Dutch = True
coffee = True
condition = "not eligible for the experiment"
## version 1
### The order of checking for study, coffee and
### language proficiency are interchangable
if age >= 18:
    if study == "Psychology" or study == "Communication Sciences":
        if coffee == True and speaks_Dutch == True:
            if gender == "Female":
                condition = "A"
            else:
                condition = "B"
        else:
            print "Participant is not eligible to take part in the experiment."
    else:
        print "Participant is not eligible to take part in the experiment."
else:
    print "Participant is not eligible to take part in the experiment."

## version 2
if age >= 18 and study == "Psychology" or study == "Communication Sciences" and coffee == True
    if gender == "Female":
        condition = "A"
    else:
        condition = "B"
```

```
else:
    print "Participant is not eligible to take part in the experiment."
```

9.3.10 Exercise 10. Flow chart conditionals

```
n = -1
if n > 0:
    n = 2*n
else:
    n += 1
if n > 4*241%17:
    n -= 5
else:
    if n > 10:
        n += 50
    else:
        n = n%2
print n
```

9.4 Chapter 4. Lists

9.4.1 Exercise 1. Indexing

What is the output of the following code snippets?

```
a = [6, 7]
b = [4, 5]
b.append(a)
print(b)
```

```
## [4, 5, [6, 7]]
```

```
d = {1: 'a',
    'a': 1}

print(d[1])
```

```
## a
```

```
a = [1]
b = ('a', 2)
a.extend(b)
print(a[1])
```

```
## a
```

```
a, b = (['a', 'd'], ['b', 'c'])
print(b[0])
```

```
## b
```

```
a = [1, 2, 3]
print(a[-1])
```

```
## 3
```

```
a = (1, 2, [3, 4], 5)
print(a[2][1])
```

```
## 4
```

9.4.2 Exercise 2. Debugging

Will the following code snippets throw an error? If anything goes wrong, why does it go wrong?

```
a, b = 'Sigmund', 'Freud', (1856, 1939)
```

- A `ValueError` is thrown. There are too many values on the right side of the equals sign `=` to unpack.

```
a = [1, 2]
b = (3, 4)

a.append(b)
print(a[3])
```

- An `IndexError` is thrown. After appending `b`, `a` has three elements. `3` is not a valid index.

```
a = ('a', [1, 2])

a[1].append(3)
```

- No error is thrown, `3` is appended to `[1, 2]`.

```
a = {('a', 'b'): 3}

print(a.keys())
```

- No error is thrown, the list of keys contains one element, `('a', 'b')`.

9.4.3 Exercise 3. Mini programs

What is the output of the following mini programs?

```
a = {}

print(a.keys())
```

- `[]`

```
a = []
b = ([1, 2], [3, 4], [5, 6])

a.extend(b)
print(a[1])
```

- `[3, 4]`

```
a = ('a', ('b', 'c'))

print('b' in a)
```

- `False`

```
a = [{'a': 1 }, {'a': 2}]

if 'a' in a[0]:
    print(a[0]['a'])
elif 'a' in a[1]:
    print(a[1]['a'])
```

- `1`

9.4.4 Exercise 1. Shopping list

```
['buns', 'cheese', 'milk', 'oatmeal', 'blueberries', 'oranges',
'apples', 'chocolate', ['eggs', 'yoghurt', 'salmon'], ['icing sugar',
'whipping cream', 'lemons', 'flower', 'vanilla sugar', 'eggs',
'baking powder', 'margarine']]
```

9.4.5 Exercise 2. Dictionnaries

```
('Erik', 'Erikson', (1902, 1994))
('Theory of Cognitive Development',)
1856
"Frederic"
"Skinner Box"
"positive reinforcement"
```

9.4.6 Exercise 3. Selecting elements

```
shoppingList[2]
shoppingList[0][0]
grades["p3"][3]
grades["p5"][1]
grades["p6"][0]
grades["p1"][4][2]
grades["p5"][4][0]
grades["p6"][4][1]
```

9.4.7 Exercise 4. Adjust a dictionnary data set

```
#1
dataset['p1'][5] = 6.1
dataset['p7'][5] = 7.4
#2
dataset['p5'][0] = 23
dataset['p5'][1] = "Female"
#3
dataset['p4'][4][2] = True
dataset['p4'][4][3] = True
```

```
## or
dataset['p4'][4] = [True,True,True,True,True]
#4
dataset['p2'][3] = "B-Health_Sciences"
#5
dataset['p4'].append("expelled")
## or
dataset['p4'] = dataset['p4'] + ["expelled"]
#final dataset
dataset = {'p1':[21,"Female","Dutch","B-Psychology",[True,False,True,False,False],6
           'p2':[20,"Female","Dutch","B-Health_Sciences",[True,True,True,False,True
           'p3':[21,"Female","Dutch","B-Applied_Mathematics",[False,True,True,False
           'p4':[23,"Male", "German","B-Communication_Science",[True,True,True,True
           'p5':[23,"Female","Dutch","M-Business_Administration",[False,False,True,
           'p6':[19,"Male","Swedish","B-Computer_Science",[True,False,False,True,Fa
           'p7':[19,"Male","German","B-Communication_Science",[True,True,False,True
           }
```

9.4.8 Exercise 5. A dictionnary data set

```
import numpy as np
participants = {'p1':("Male",19,"Dutch","Student"),
                'p2':("Male",47,"Dutch","Pharmacist"),
                'p3':("Male",31,"Italian","PhD Student"),
                'p4':("Female",22,"German","Student"),
                'p5':("Female",46,"Dutch","Florist"),
                'p6':("Male",27,"Dutch","Student"),
                'p7':("Female",22,"Dutch","Police trainee"),
                'p8':("Female",26,"Indian","Architect"),
                'p9':("Male",18,"American","Student"),
                'p10':("Male",20,"Chinese","Student")
               }
""" Calculate the average age """
mean_age = (participants['p1'][1]+participants['p2'][1]+participants['p3'][1]+parti
# A more elegant, manual solution you will learn about in "Chapter 5. Loops"
# would be as follows:
age_sum = 0
for key in participants.keys():
    age_sum += participants[key][1]
loop_mean = age_sum/float(len(participants))
# Using Numpy greatly simplifies calculations, but
# you first need to transform the data to fit your needs.
```

```
# You will see, however, that extracting the age variable and saving it
# separately will simplify other statistical operations later on.
age = [participants['p1'][1],
       participants['p2'][1],
       participants['p3'][1],
       participants['p4'][1],
       participants['p5'][1],
       participants['p6'][1],
       participants['p7'][1],
       participants['p8'][1],
       participants['p9'][1],
       participants['p10'][1]]
numpy_mean = np.mean(age)
""" Calculate the standard deviation of the age variable """
std_age = np.std(age)
""" Calculate the minimum and maximum of the age variable """
minimum = np.nanmin(age)
maximum = np.nanmax(age)
```

9.4.9 Exercise 6. Stroop extension

```
# -*- coding: utf-8 -*-
import pygame
import sys
from time import time
import random
from pygame.locals import *
from pygame.compat import unichr_, unicode_
##### VARIABLES #####
# Colors
col_white = (250, 250, 250)
col_black = (0, 0, 0)
col_gray = (220, 220, 220)
col_red = (250, 0, 0)
col_green = (0, 200, 0)
col_blue = (0, 0, 250)
col_yellow = (250,250,0)
col_pink = (250,0,127)
NTRIALS = 5
WORDS    = ("red", "green", "blue", "yellow", "pink")
COLORS   = {"red": col_red,
```

```
            "green": col_green,
            "blue": col_blue,
            "yellow": col_yellow,
            "pink": col_pink}
KEYS     = {"red": K_b,
            "green": K_n,
            "blue": K_m,
            "yellow":K_v,
            "pink":K_c}
BACKGR_COL = col_gray
SCREEN_SIZE = (700, 500)
pygame.init()
pygame.display.set_mode(SCREEN_SIZE)
pygame.display.set_caption("Stroop Test")
screen = pygame.display.get_surface()
screen.fill(BACKGR_COL)
font = pygame.font.Font(None, 80)
font_small = pygame.font.Font(None, 40)
p_numbers = range(1,11)
conditions = {"Stroop_3":[1,2,4,8,10],
              "Stroop_5":[3,5,6,7,9]}
def main():
    """ Start the Stroop task.
    """
    ## Variables
    STATE = "welcome"
    trial_number = 0
    # initialize participant number
    p_number = 0
    # for gathering the response times
    RT = []
    while True:
        pygame.display.get_surface().fill(BACKGR_COL)
        # Changing states by user input
        for event in pygame.event.get():
            # welcome screen --> prepare next trial (space bar)
            if STATE == "welcome":
                if event.type == KEYDOWN and event.key == K_SPACE:
                    STATE = "enter_participant_number"
                    print(STATE)
            # wait for response --> feedback (b, n, m)
            elif STATE == "wait_for_response":
                if event.type == KEYDOWN and event.key in KEYS.values():
```

```
                # remember when the user has reacted
                time_when_reacted = time()
                # calculate the response time
                this_reaction_time = time_when_reacted - time_when_presented
                RT.append(this_reaction_time)
                # was the response correct?
                this_correctness = (event.key == KEYS[this_color])
                STATE = "feedback"
                print(STATE)

        elif STATE == "enter_participant_number":
            p_number = prompt()
            STATE = "transition_experiment"
            print STATE + "\nRETURN TO PYGAME WINDOW"

        elif STATE == "transition_experiment":
            if event.type == KEYDOWN and event.key == K_SPACE:
                STATE = "prepare_next_trial"
        if event.type == QUIT:
            STATE = "quit"
    # automatic state transitions
    # prepare next trial --> wait for response (immediatly)
    if STATE == "prepare_next_trial":
        trial_number = trial_number + 1
        # randomly pick word and color
        # depending on condition
        if p_number in conditions["Stroop_3"]:
            this_word  = pick_color()
            this_color = pick_color()
        else:
            this_word = pick_color5()
            this_color = pick_color5()
        # remember when stimulus was presented
        time_when_presented = time()
        STATE = "wait_for_response"
        print(STATE)
    # show feedback, then advance to next trial or goodbye  (for 1s)
    if STATE == "feedback" and (time() - time_when_reacted) > 1:
        if trial_number < NTRIALS:
            STATE = "prepare_next_trial"
        else:
            STATE = "goodbye"
        print(STATE)
```

```
        # Drawing to the screen
        if STATE == "welcome":
            draw_welcome()
            draw_button(SCREEN_SIZE[0]*1/6, 450, "Pink: C", col_pink)
            draw_button(SCREEN_SIZE[0]*2/6, 450, "Yellow: V", col_yellow)
            draw_button(SCREEN_SIZE[0]*3/6, 450, "Red: B", col_red)
            draw_button(SCREEN_SIZE[0]*4/6, 450, "Green: N", col_green)
            draw_button(SCREEN_SIZE[0]*5/6, 450, "Blue: M", col_blue)

        if STATE == "enter_participant_number":
            draw_enter()

        if STATE == "transition_experiment":
            draw_transition()
            if p_number in conditions["Stroop_3"]:
                draw_button(SCREEN_SIZE[0]*1/4, 450, "Red: B", col_red)
                draw_button(SCREEN_SIZE[0]*2/4, 450, "Green: N", col_green)
                draw_button(SCREEN_SIZE[0]*3/4, 450, "Blue: M", col_blue)
            else:
                draw_button(SCREEN_SIZE[0]*1/6, 450, "Pink: C", col_pink)
                draw_button(SCREEN_SIZE[0]*2/6, 450, "Yellow: V", col_yellow)
                draw_button(SCREEN_SIZE[0]*3/6, 450, "Red: B", col_red)
                draw_button(SCREEN_SIZE[0]*4/6, 450, "Green: N", col_green)
                draw_button(SCREEN_SIZE[0]*5/6, 450, "Blue: M", col_blue)

        if STATE == "wait_for_response":
            draw_stimulus(this_color, this_word)
            if p_number in conditions["Stroop_3"]:
                draw_button(SCREEN_SIZE[0]*1/4, 450, "Red: B", col_red)
                draw_button(SCREEN_SIZE[0]*2/4, 450, "Green: N", col_green)
                draw_button(SCREEN_SIZE[0]*3/4, 450, "Blue: M", col_blue)
            else:
                draw_button(SCREEN_SIZE[0]*1/6, 450, "Pink: C", col_pink)
                draw_button(SCREEN_SIZE[0]*2/6, 450, "Yellow: V", col_yellow)
                draw_button(SCREEN_SIZE[0]*3/6, 450, "Red: B", col_red)
                draw_button(SCREEN_SIZE[0]*4/6, 450, "Green: N", col_green)
                draw_button(SCREEN_SIZE[0]*5/6, 450, "Blue: M", col_blue)

        if STATE == "feedback":
            draw_feedback(this_correctness, this_reaction_time)

        if STATE == "goodbye":
            draw_goodbye()
```

```
        if STATE == "quit":
            pygame.quit()
            sys.exit()
        pygame.display.update()
def prompt():
    p_number = 0
    while p_number == 0:
        p_number = int(raw_input("Please enter participant number here:"))
    if p_number in range(1,len(p_numbers)+1):
        return p_number
    else:
        print "Unknown participant number, valid participant numbers are 1 to 10"
        prompt()

def pick_color():
    """ Return a random word.
    """
    random_number = random.randint(0,2)
    return WORDS[random_number]
def pick_color5():
    """ Return a random word,
    5 color Stroop version
    """
    random_number = random.randint(0,4)
    return WORDS[random_number]
def draw_button(xpos, ypos, label, color):
    text = font_small.render(label, True, color, BACKGR_COL)
    text_rectangle = text.get_rect()
    text_rectangle.center = (xpos, ypos)
    screen.blit(text, text_rectangle)
def draw_welcome():
    text_surface = font.render("STROOP Experiment", True, col_black, BACKGR_COL)
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,150)
    screen.blit(text_surface, text_rectangle)
    text_surface = font_small.render("Press Spacebar to continue", True, col_black, BACKGR_COL
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,300)
    screen.blit(text_surface, text_rectangle)
def draw_enter():
    text_surface = font_small.render("Please enter participant number in console", True, col_b
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,250)
```

```
    screen.blit(text_surface, text_rectangle)
def draw_transition():
    text_surface = font_small.render("Press Spacebar to start the experiment", True
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,250)
    screen.blit(text_surface, text_rectangle)
def draw_stimulus(color, word):
    text_surface = font.render(word, True, COLORS[color], col_gray)
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,150)
    screen.blit(text_surface, text_rectangle)
def draw_feedback(correct, reaction_time):
    if correct:
        text_surface = font_small.render("correct", True, col_black, BACKGR_COL)
        text_rectangle = text_surface.get_rect()
        text_rectangle.center = (SCREEN_SIZE[0]/2.0,150)
        screen.blit(text_surface, text_rectangle)
        text_surface = font_small.render(str(int(reaction_time * 1000)) + "ms", Tru
        text_rectangle = text_surface.get_rect()
        text_rectangle.center = (SCREEN_SIZE[0]/2.0,200)
        screen.blit(text_surface, text_rectangle)
    else:
        text_surface = font_small.render("Wrong key!", True, col_red, BACKGR_COL)
        text_rectangle = text_surface.get_rect()
        text_rectangle.center = (SCREEN_SIZE[0]/2.0,150)
        screen.blit(text_surface, text_rectangle)
        #text_surface = font_small.render("Press Spacebar to continue", True, col_b
def draw_goodbye():
    text_surface = font_small.render("END OF THE EXPERIMENT", True, col_black, BACK
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,150)
    screen.blit(text_surface, text_rectangle)
    text_surface = font_small.render("Close the application.", True, col_black, BAC
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (SCREEN_SIZE[0]/2.0,200)
    screen.blit(text_surface, text_rectangle)

main()
```

9.5 Chapter 5. Loops

9.5.1 Exercise 1. Following the control flow

```
['p2', 'p3', 'p1', 'p6', 'p7', 'p4', 'p5', 'p10', 'p8', 'p9']
(20, 'Female', 'Dutch', 'B-Psychology')
(20, 'Female', 'Dutch', 'B-Psychology')
20
5
4
21.8
19.75
```

9.5.2 Exercise 2. Debugging

```
data = {'p1':(21,"Female","Condition B",[0.675,0.777,0.778,0.62,0.869]),
        'p2':(20,"Female","Condition A",[0.599,0.674,0.698,0.569,0.7]),
        'p3':(21,"Female","Condition A",[0.655,0.645,0.633,0.788,0.866]),
        'p4':(23,"Male", "Condition A",[0.721,0.701,0.743,0.682,0.654]),
        'p5':(20,"Male","Condition B",[0.721,0.701,0.743,0.682,0.654]),
        'p6':(19,"Male","Condition B",[0.711,0.534,0.637,0.702,0.633]),
        'p7':(19,"Male","Condition B",[0.687,0.657,0.766,0.788,0.621]),
        'p8':(24,"Female","Condition A",[0.666,0.591,0.607,0.704,0.59]),
        'p9':(23,"Female","Condition B",[0.728,0.544,0.671,0.689,0.644]),
        'p10':(18,"Male","Condition A",[0.788,0.599,0.621,0.599,0.623])
        }
fastest = ("initialization",100)
for participant in data:
    RTsum = 0
    for i in range(len(data[participant][3])-1):
        RTsum += data[participant][3][i]
    RTmean = RTsum/len(data[participant][3])
    if RTmean < fastest[1]:
        fastest = (participant,RTmean)

slowest = ("initialization",0)
for participant in data:
    RTsum = 0
    for RT in data[participant][3]:
        RTsum += RT
    RTmean = RTsum/len(data[participant][3])
```

```
    if RTmean > slowest[1]:
        slowest = (participant,RTmean)

all_mean = 0
all_sum = 0
number_of_trials = 0
for participant in data:
    counter = 0
    while counter < len(data[participant][3]):
        all_sum += data[participant][3][counter]
        number_of_trials += 1
        counter += 1
all_mean = all_sum/number_of_trials*len(data)

print "fastest:", fastest
print "slowest:", slowest
print "all_mean:", all_mean
```

9.5.3 Exercise 3. Nested loops

```
participants = [
    ("p1", ["Condition 1","Location A","withdrew","no audio"]),
    ("p2", ["Condition 1","Location A","no audio","no video"]),
    ("p3", ["Condition 2","Location B"]),
    ("p4", ["Condition 1","Location A","withdrew"]),
    ("p5", ["Condition 2","Location A"]),
    ("p6", ["Condition 2","Location B","withdrew"]),
    ("p7", ["Condition 1","Location A","no video"]),
    ("p8", ["Condition 1","Location B"]),
    ("p9", ["Condition 2","Location B","withdrew"]),
    ("p10",["Condition 1","Location A","withdrew"])]
counter = 0
for (participant,expInfo) in participants:
        for info in expInfo:
            if info == "withdrew":
                counter += 1

print "The number of participants who withdrew their participation is", counter
```

9.5.4 Exercise 4. Data transformation using loops

```
participants = [
    ("p1", ["Condition 1","Location A","withdrew","no audio"]),
    ("p2", ["Condition 1","Location A","no audio","no video"]),
    ("p3", ["Condition 2","Location B"]),
    ("p4", ["Condition 1","Location A","withdrew"]),
    ("p5", ["Condition 2","Location A"]),
    ("p6", ["Condition 2","Location B","withdrew"]),
    ("p7", ["Condition 1","Location A","no video"]),
    ("p8", ["Condition 1","Location B"]),
    ("p9", ["Condition 2","Location B","withdrew"]),
    ("p10",["Condition 1","Location A","withdrew"])]
dict_participants = {}
for participant in participants:
    dict_participants[participant[0]] = participant[1]
counterA = 0
counterB = 0
for key in dict_participants:
    if dict_participants[key][1] == "Location A":
        counterA += 1
    else:
        counterB += 1
## or using nested loops
counterA1 = 0
counterB1 = 0
for key in dict_participants:
    for info in dict_participants[key]:
        if info == "Location A":
            counterA1 += 1
        elif info == "Location B":
            counterB1 += 1
print "The number of participants who were tested at location A is", counterA, counterA1
print "The number of participants who were tested at location B is", counterB, counterB1
```

9.5.5 Exercise 5. Calculating a mean

```
import numpy as np
seq = range(1000)
counter = 0
sum = 0.0
```

```
while counter <= 999:
    sum += seq[counter]
    counter += 1
mean = sum/len(seq)
print mean, np.mean(seq)
```

9.5.6 Exercise 6. A guessing game

```
import random
import sys
number = random.randint(0,1000)
guesses = 0
while(True):
    try:
        user_input = input("Please enter a number between 0 and 1000")
    except SyntaxError:
        sys.exit()
    if user_input == number:
        print "Ding Ding Ding! Correct! The number was", number
        guesses +=1
        print guesses, "guesses needed"
        break
    elif user_input > number:
        print "My number is smaller"
        guesses +=1
    elif user_input < number:
        print "My number is larger"
        guesses +=1
sys.exit()
```

9.6 Chapter 6. Functions

9.6.1 Exercise 1. Following the control flow

```
"x equals 10"
"y equals 17.5"
"anumber equals 10.0"
```

9.6.2 Exercise 2. An imperfect list sorting attempt

- *Line 6/7.* The `insert` function does not return the resulting list.
- *Line 20 and 21.* The output of the `swap` function is not assigned to any variable and thereby, the manipulation performed on `myList1` is not stored in memory.
- *Line 22.* Only two arguments are provided during the function call of `insert`. The function, however, requires three arguments: an element to be inserted, a position indicating where to insert the element, and a list into which the element is to be inserted.
- *Line 25.* The `swap` function is used incorrectly. First of all, the function only takes three arguments, but five arguments are provided. The person tried to swap several element pairs at once while the function is only suitable for swapping one pair at a time!
- *Line 25.* Second, the order of the arguments provided to the swap function is messed up. The function first takes one element to be swapped, then the element with which the first element should be swapped and only then the list which contains the two elements.
- *Line 26.* The `insert` function is as it is defined not suitable for appending elements at the end of a list. This can be solved in one of two ways: either, the function is left as is and instead of using insert, the built-in function `append()` can be used. Or, and this makes the insertion function more robust, a check is added to the function, appending any element that is supposed to be inserted at a position that exceeds the index range of the list.

```
def insert(a,position,alist):
    result = copy.deepcopy(alist)
    if position >= len(result):
        return result.append(a)
    else:
        return result[:position] + [a] + result[position:]
```

9.6.3 Exercise 3. An errorneous sorting algorithm

```
import copy
import random
def swap(a,b,alist):
    index_a, index_b = alist.index(a), alist.index(b)
    index_a, index_b = index_b, index_a
    result = copy.deepcopy(alist)
    result[index_a], result[index_b] = a,b
    return result
def bubbleSort(alist):
```

```
    result = copy.deepcopy(alist)
    for iteration in range(len(result)-1):
        for index in range(len(result)-1,0,-1):
            if result[index] < result[index-1]:
                result = swap(result[index],result[index-1],result)
    return result
myList = range(51)
random.shuffle(myList)
print myList
print bubbleSort(myList)
```

9.6.4 Exercise 4.

9.6.5 Exercise 5.

9.6.6 Exercise 6. Insertion sort algorithm

```
import copy
import random
def insertionSort(alist):
    result = copy.deepcopy(alist)
    for index in range(1,len(result)):

        # Temporarily assign the element that is to be compared to the
        # (sorted) sublist at the left of the element's position
        value = result[index]
        # Remember the position in the list of the element under investigation
        position = index

        # Stepwise, compare each element left to the designated element (b) and the
        # designated element (value). Whenever b is larger than value, update posit
        # by shifting the position of b to the current value of position, thus in f
        # one place to the right. Continue until position equals 0 and there are no
        # left to compare value to.
        while position > 0 and result[position-1] > value:
            result[position] = result[position-1]
            position -= 1

        # Insert the value that has been compared to at the right position in the l
        result[position] = value
    return result
```

```
myList = range(51)
random.shuffle(myList)
print insertionSort(myList)
```

10 Debugging and good programming practices

10.1 Debugging

Debugging is one of the most intellectually challenging aspects of programming. It is often also perceived as the most frustrating aspect. To a large extent, the problems that people face while debugging are not so much technical as they are psychological. Debugging starts with the realization that a computer does nothing wrong. In fact, it does exactly what you tell it to do. The fault lies thus in how the programmer conceived or implemented a solution. You should always remember that even though debugging can be a difficult and frustrating endeavour, it can be done. At times, it will require all the creativity and skill at your disposal, but you will succeed if you act systematically and do not give up on the task.

There is no recipe-like cookbook approach to debugging, but adhering to some general principles will probably help.

10.1.1 The print statement is an aspiring programmer's best friend

In debugging you take on the role of a detective. As such, you focus on what the errorneous program is doing, rather than why it is not doing what you wanted it to do. You work from the facts. Variables and the flow of your program are the key to finding the crucial pieces of information that will lead you to the clue to what is going wrong.

The `print` statement is a commonly used tool for debugging. Even if you have no clue as to what is going wrong, you often have a hunch where the culprit might be. If you do not have any idea where to start, simply start from the beginning of your program. It also helps to realize that the bug is not moving around in your code, trying to trick or to evade you (even though it might feel as if it has a will of its own at times). It is sitting in the same place, doing the wrong thing in the same way every time. You start from your hunch and take a closer look at your variables and whether the program is doing what it is supposed to do in this place. Here, `print` statements come in handy. You can use `print` statements to make the current values assigned to your variables visible. By following the control flow of the program in your mind (that is calculating the expected values of your variables as you go through the program), you should verify whether your variables were assigned their expected values. If yes, continue to search for the error

further downward in the flow of the program. If not, look further upward for the bug. Also take a close look at the values themselves. Do they belong to the expected *data type*? Are their values in any way remarkable (too high, too low, etc.)? This will give some indication of what is going wrong.

10.1.2 Frequent unit-testing

Rome was not built in a day and the same is very true about computer programs. Programming, as any other task, can and should be split into subtasks. These subtasks can be chunks of code like *functions* that generalize operations you frequently need in your program, or the division between graphical and state transition dynamics of your program. Individual components should frequently and extensively be tested for bugs *before* merging them into the rest of your code. One simple bug in a small component of your program can become untracable in your main code and cause the weirdest errors. It is generally a good idea to pull out code from your main program and put it into functions to keep the main flow of your program readable and to prevent yourself from writing out the same lines of code over and over again. Just make sure that you frequently test these units of code before putting them into the main flow of your program and of course, that the main flow of your program is still working nicely. Inserting `print` statements before and after calling a function to check whether the involved variables were mutated in the intended way is one way to make sure that the different chunks of your program interact with each other nicely.

10.1.3 The copy-paste syndrome

Copying code is generally a bad idea. Extensive copying of code takes one of two forms. Some programmers over-copy their own code, others copy other people's code. Little is more frustrating than realizing that you are debugging the same bug multiple times. Whenever you copy undebugged chunks of code, you also copy the bugs within them. You forget which chunks of code you copied where and voilà; chances are high that you encounter the same bug in different places. Putting chunks of code that your program repeatedly uses into functions is a good way to counteract over-copying one's own code. There is no point in re-inventing the wheel, so there is nothing wrong about trying out another person's debugging approach to one's own code. However, over-copying other people's code is not only bound to cause more errors in the long run, it will also leave you with a great deal of unanswered questions about *why* another's solution fixes the bug and *why* your own attempts do not. In the worst case, you do no longer understand your own code! Copying code works against understanding code. Instead, try to program something similar, adjust small parts and see what happens. That way, you work towards understanding (another's) code.

10.1.4 Standing on the shoulders of giants

That being said, chances are high that, whatever you try to accomplish in programming, somebody else has already tried it (and may have run into the same errors and bugs). Looking on the internet for a solution can prove to be useful and point your debugging efforts into the right direction. Truth be told, you will rarely find the perfect solution to your individual problem in the context of your personal, specific code. Still, you can figure out on the internet whether you are dealing with a known bug or read up on more specific information in the Python documentation. To do so, you should search for keywords relevant to your problem, such as *operator precedence* or *float division*. Just be aware that you will always need to transfer whatever you read about to your individual situation which can be a difficult task as a programming beginner. Taking on the role of detective and figuring out yourself what is going wrong is usually the way to go. Reading up on the internet is merely a tool.

10.2 Boosting your programming learning curve

When you start programming, it is easy to become intimidated by the amount of information you find in programming books and the multitude of error messages you encounter. Initially, you may also have difficulty adjusting to how a programmer thinks. Really, to some extent programming is but another way to approach the problem at hand. Programming means seeing a problem from another perspective. To cheer you up, even the most experienced programmers make mistakes, beginner's mistakes included. They encounter the same error messages as you do. I dare to say that, in total, they spend about as much time on debugging code than any beginner. That is the reality of programming. As your programming skills improve and you become acquainted with thinking with the mind of a programmer, you will be enabled to tackle more advanced problems and solve easier problems in more efficient ways. Luckily, there are some good practices that will speed up the development of your programming skills. In truth, without these good practices, your journey as an aspiring programmer will be much harder. Therefore, it is best to internalize the following practices from the beginning.

10.2.1 Going with the flow

Whenever you read code, try to mentally follow the flow of the program step by step.

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

```
## 2
```

In the code snippet above, the *variable* `x` is assigned the *value* `1` in line one. You would thus think for yourself "variable x is assigned the value of 1" when reading the first line of code. Chapter 3 Variables, values and types explains in detail what *variables* are and how they are assigned a *value*. For now, we focus on how to approach such a code snippet. Back to the code snippet, you read line two. In line two, `x` is assigned another value, namely `2`. You will learn in Chapter 2 that variables can only be assigned one value at a time and that assigning a new value to a variable erases any earlier value assignment. Variable `x` has thus a new value once the program executes line two, namely `2`. In the last line of the program, the value of the variable `x` is printed to the console. When it is executed, `2` thus appears in the console.

Why do neither `1` nor `3` not appear in the console output? The console output is not `1` because the `print` statement comes after the second value assignment. `x` is assigned the value `2` already. The output is neither `3` because any value assignment after an initial assignment simply overwrites a variable's earlier value.

Following the flow of a program will help you understand the algorithm in front of you. You can decipher even complex algorithms by just following the flow of code. Comprehending the flow of a computer program is a crucial step in understanding what the program does and how it achieves its purpose. The best thing is, approaching code by following its flow develops your own programming skills much like reading a book in a foreign language improves your language skills. By reading a book your vocabulary expands. You learn new words, how a word is used in a sentence and in which context a word is used. Often, when you encounter an unfamiliar word, you understand its meaning based on context alone. Reading code and following its flow is to your programming skills what reading a book in a foreign langauge is to your capability to understand and produce that language. At first, you encounter numerous unfamiliar constructs, but following the flow of the program will help you understand these. *Printing* to the console is a useful tool for making the flow of a program visible. Do not hesitate to insert `print` statements in any code in front of you. `print` statements can show you the value of a variable at any time or whether the program reaches a certain part of the program before it is terminated (possibly prematurely due an error).

10.2.2 Just do it!

When you read about something new, be it in this book, on the internet or in class, try it yourself and see what happens! This is an essential learning attitude when learning how to program. Definitely, you cannot learn how to program by just reading books. It is above all a "learning by doing" skill. Chapter 2 [Variables, values and data types] introduces *float division*. Float division is a peculiarity of all Python 2 versions. When an integer division does not result in another integer, the result is truncated instead of represented as a floating point. Your first reaction when reading about float division should be to open your Python interpreter and try out what happens when you divide two integers which do not share a common multiple.

```
x = 7
y = 3
print(x / y)
```

```
## 2
```

After having executed the above code snippet in your Python interpreter, does it not become more apparent what I meant by *truncated*? Being a psychologist you know that writing (by hand) facilitates structuring your thoughts and subsequently, memorizing. Trying out code snippets is no different in this respect.