Advanced Software Engineering

Pet Project

Game Decision Tree

Felicitas König
(Matr-No. 886778)

Contents

1. Introduction

The Game Decision Tree is a little program which should help to find a decision regarding to the evening or weekend plans. For now, the game is designed for two players and includes two steps of decisions. First the players choose one of three categories that reflects their mood. If they have chosen the same one, the game comes directly to the next step. If the chosen categories are different tic tac toe is played. A coin toss decides who may start first. After the category is chosen via tic tac toe, the players select an actual activity within this category. Is this option again the same, there is no need to compromise. However, if the activities are different, the players have to find a decision via rock-paper-scissors. Finally, you will have an independently decided evening or weekend plan and hopefully have a lot of fun.

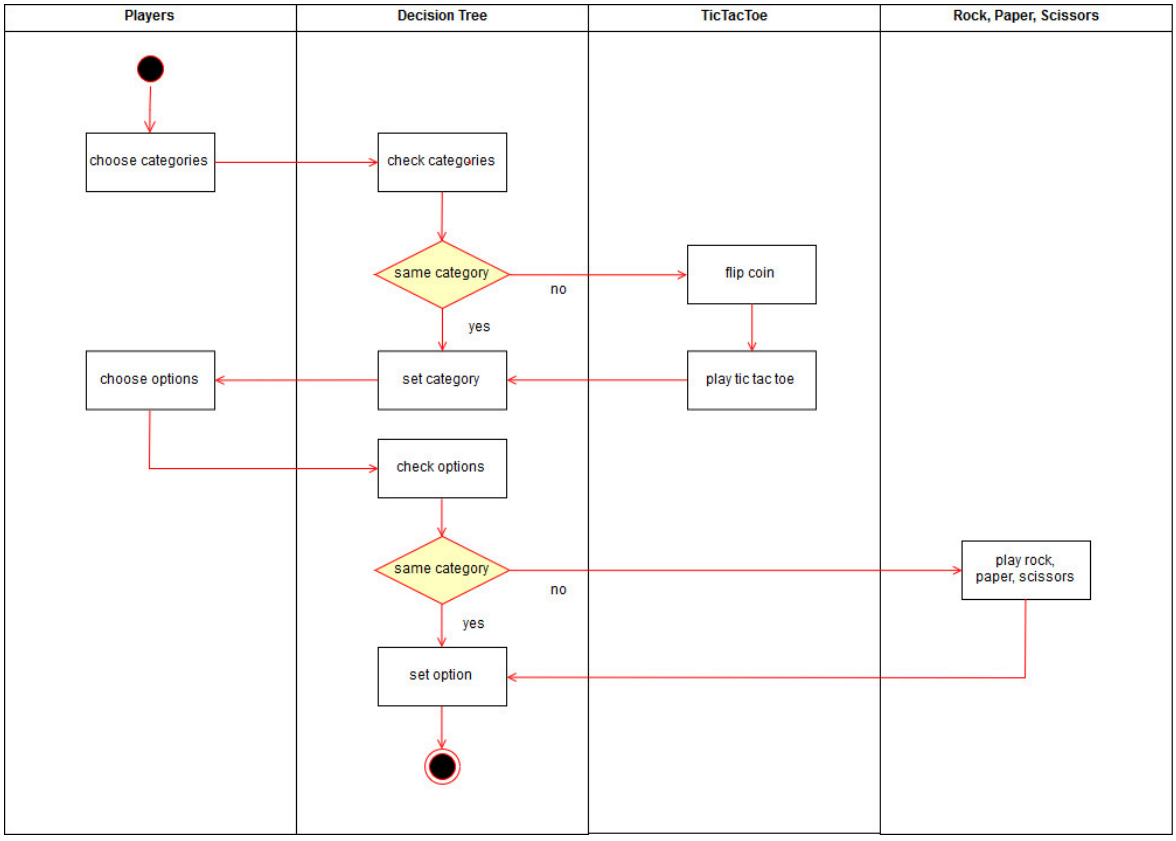
The Game Decision Tree is still a very rudimental version and could be extended by e.g. more players, a whole data base of activities or concrete events/restaurants/other places in a specific city.

2. UML

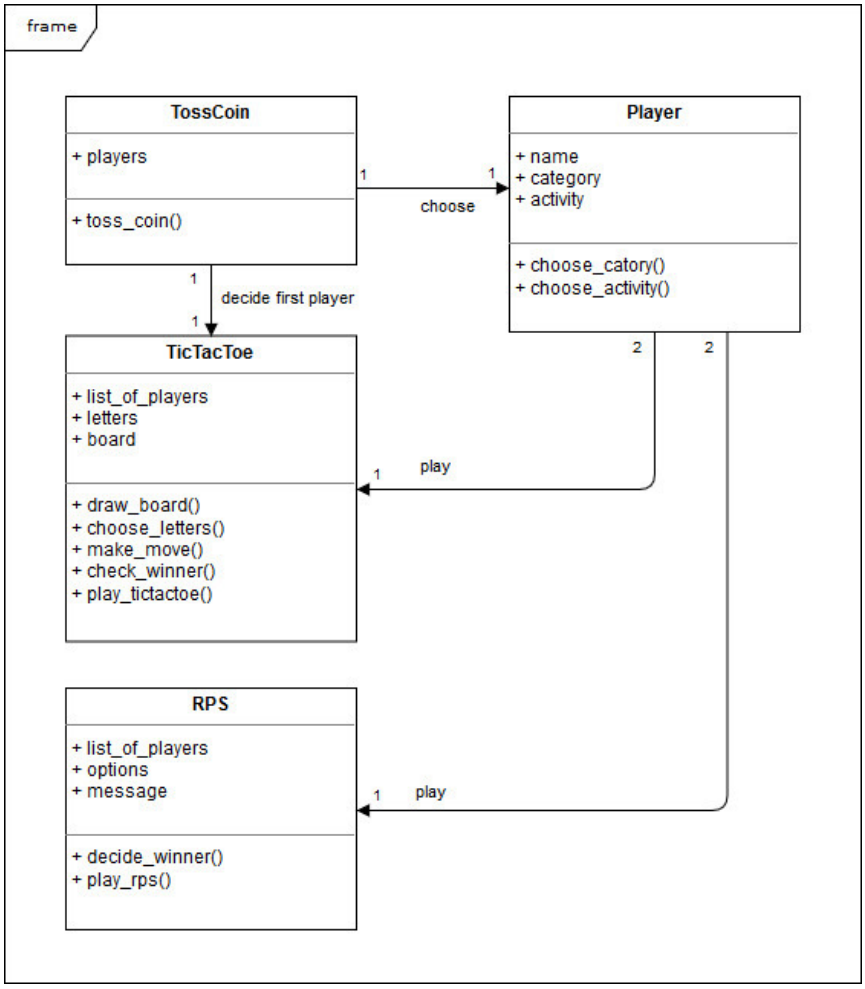
The UML diagrams a made with draw.io.

Files: pet-project\uml

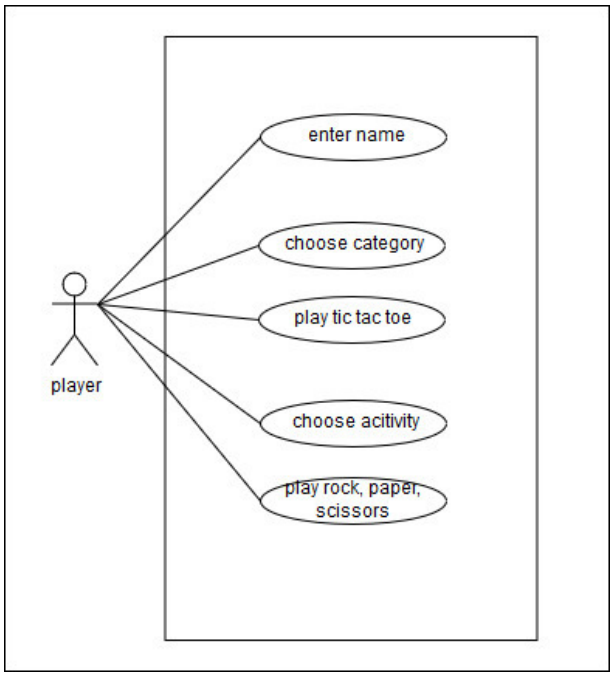
2.1 Activity Diagram



2.2 Class Diagram

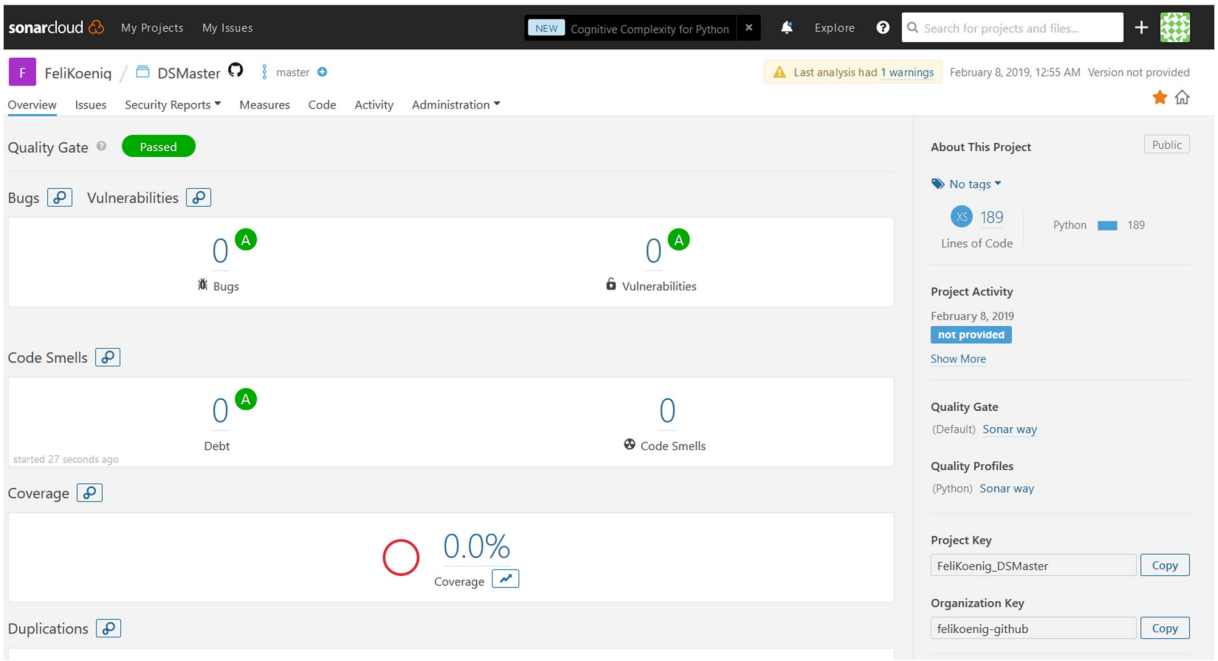


2.3 Component Diagram

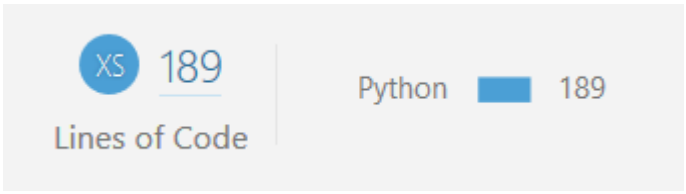


3. Metrics

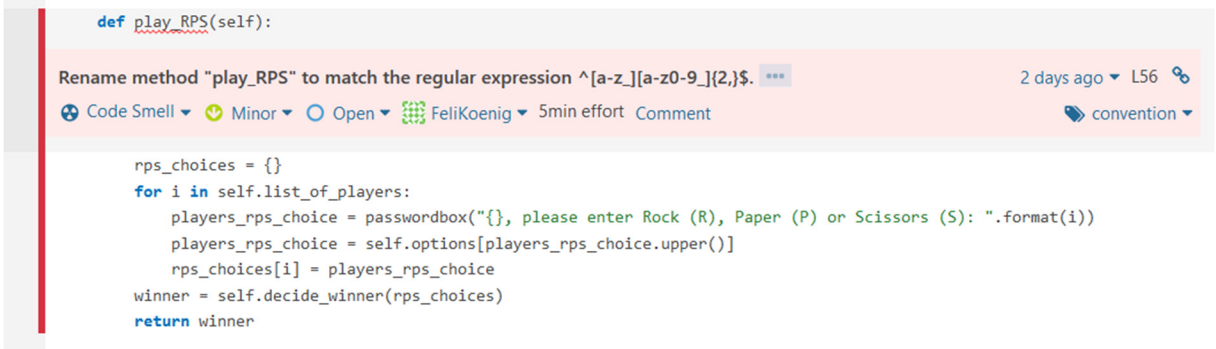
I used Sonarcloud to investigate the metrics of my python-code.



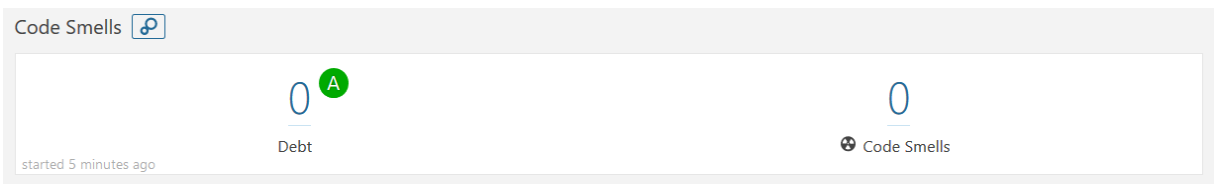
Lines of Code:



Code Smells:



→ After correcting the Code Smells:



4. Clean Code Development

4.1 Meaningful Naming

Functions and variables should be named as precise but still simple as possible. In the example below a function for playing Tic Tac Toe is shown. All steps could be clearly identified by their name.

```
def play_tictactoe(self):
    # show the board and first player select a letter
    self.draw_board()
    self.choose_letters()

    # start playing
    move = input("What is your first move? Choose 1 - 9. ")
    self.make_move(move, self.letters[0])
    move = input("Now, it's your turn, {}. Choose a position. ".format(self.list_of_players[1]))
    self.make_move(move, self.letters[1])
    for i in range(7):
        j = i % 2
        move = input("Next {}: ".format(self.list_of_players[j]))
        self.make_move(move, self.letters[j])

        # check if someone won
        winner = self.check_winner()
        if winner != 0:
            print("Yeah, {} you won!".format(winner))
            break
    return winner
```

4.2 Don’t repeat yourself

One should not repeat code artefacts, but use refactorization to avoid repetitions. In the example below I used a for-loop in connection with a class-function instead of repeating the code for each player.

```
# function to get and store the category choice
def choose_categories(players):
    players_categories={}
    for p in players:
        print("\n{}, in which mood are you today? (1) {}, (2) {} or (3) {}"
              .format(p.name, [*dict_cat_act][0], [*dict_cat_act][1], [*dict_cat_act][2]))
        p.choose_category()
        players_categories[p.name] = p.category
    return players_categories
```

4.3 Comments

Using meaningful comments are important for the understanding of the code. It saves a lot of time reading the code.

```
# toss a coin to decide who should start playing tic tac toe
players_ordered = TossCoin(players).toss_coin()
time.sleep(2)

# playing tic tac toe to decide the category
winner = TicTacToe(players_ordered).play_tictactoe()
chosen_category = players_categories[winner]
```

4.4 Exception Handling

One should check inputs to be valid to avoid errors. In the example below I checked the input of a player's choice and used also exception handling.

```
def check_input(self, choice):
    while (type(choice) != int) or (int(choice) not in range(1,4)):
        choice = input("Please choose number between 1 and 3! ")
        try:
            choice = int(choice)
        except:
            pass
    return choice

def choose_category(self):
    category = input("Choose your preferred category: ")
    category = self.check_input(category)
    self.category = [*dict_cat_act][category-1]

def choose_activity(self, chosen_category):
    activity = input("Choose your preferred activity: ")
    activity = self.check_input(activity)
    self.activity = dict_cat_act[chosen_category][activity-1]
```

4.5 Version Control

For version control I used Git and Github.

```
C:\Users\felic>cd C:\Users\felic\Documents\DataScience-Master\1_Semester\SoftwareEngineering\pet-project

C:\Users\felic\Documents\DataScience-Master\1_Semester\SoftwareEngineering\pet-project>git add .

C:\Users\felic\Documents\DataScience-Master\1_Semester\SoftwareEngineering\pet-project>git commit -m "test"
[master 8915c11] test
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename clean-coding/correct-codesmells.PNG => images/4_cleancoding_corrected-codesmells.PNG (100%)

C:\Users\felic\Documents\DataScience-Master\1_Semester\SoftwareEngineering\pet-project>git pull https://github.com/FeliKoenig/pet-project.git
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 3), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), done.
From https://github.com/FeliKoenig/pet-project
 * branch            HEAD       -> FETCH_HEAD
Merge made by the 'recursive' strategy.
 uml/usecase-diagram.xml | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

C:\Users\felic\Documents\DataScience-Master\1_Semester\SoftwareEngineering\pet-project>git push -u origin master
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 571 bytes | 571.00 KiB/s, done.
Total 5 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To https://github.com/FeliKoenig/pet-project.git
   ecaa565..4daaf2f  master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.

C:\Users\felic\Documents\DataScience-Master\1_Semester\SoftwareEngineering\pet-project>
```

5. Build Management

As a build management tool I used PyBuilder and went through the following tutorial:

https://pythonhosted.org/pybuilder/walkthrough-new.html

After installing the PyBuilder I created a build.py-file which includes some plugins used for the project and also some properties of it.

```
from pybuilder.core import use_plugin, init

use_plugin("python.core")
use_plugin("python.unittest")
use_plugin("python.install_dependencies")
use_plugin("python.flake8")
use_plugin("python.coverage")
use_plugin("python.distutils")


name = "mypybuilder"
default_task = "publish"


@init
def set_properties(project):
    project.set_property("coverage_break_build", False)  # default is True
    project.build_depends_on("mock")
```

I implemented a function in the __init__.py-file, which I accessed through a unit test.

__init__.py

```
def greet(filelike):
    filelike.write("Hello world!\n")
```

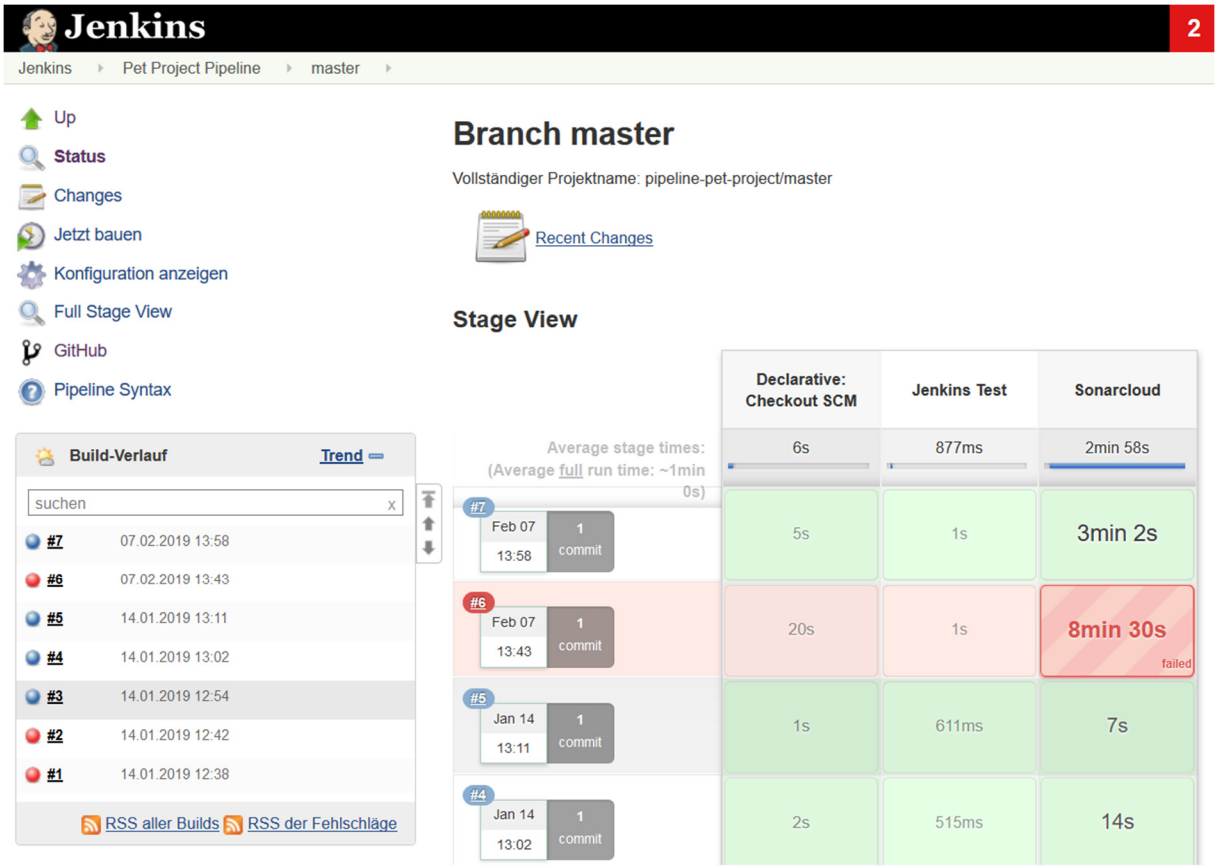
mypybuilder-test.py

```
from unittest import TestCase

from mock import Mock

from mypybuilder import greet


class Test(TestCase):

    def test_should_write_hello_world(self):
        mock_stdout = Mock()

        greet(mock_stdout)

        mock_stdout.write.assert_called_with("Hello world!\n")
```

6. Continuous Delivery

I created a Pipeline in Jenkins which for example includes to run Sonarcloud-program.

File: pet-project\Jenkins



7. DSL

DSL (Domain Specific Languages) are languages that are used to communicate with computers in a certain domain. Within this specific domain, DSLs can be used for all purposes and by different users.

One well-known example for DSL is SQL (Structured Query Language).

In further development of the game decision tree one can use a database e.g. to store concrete proposals for restaurants, bars or other interesting places in town. Such a data base could be executed via SQL.

Example for a SQL query to find all restaurants in Charlottenburg:

SELECT * FROM data_table

WHERE type = “restaurant”

AND borough = “Charlottenburg”

ORDER BY name ASC

8. Functional Programming

8.1 Final Data Structures

A final data structure are e.g. variables which are assigned only once and could not be changed. In Python tuples are such immutable variable structures.

```
# function to set the players
def set_players():
    player1 = input("Name of Player 1: ")
    player1 = Player(player1)
    player2 = input("Name of Player 2: ")
    player2 = Player(player2)
    return (player1, player2)
```

8.2 (Mostly) Side Effect Free Functions

Side effect free functions returns with same input parameters always the same output. The example below detects the winner of Paper-Rock-Scissors, which is clearly deterministic.

```
def decide_winner(self, rps_choices):
    for i in self.list_of_players:
        print("\n{}, you selected: {}".format(i, rps_choices[i]))

    i = self.list_of_players[0]
    j = self.list_of_players[1]

    if rps_choices[i] == rps_choices[j]:
        print(self.message["tie"])
        return 0
    elif (rps_choices[i] == self.options["R"] and rps_choices[j] == self.options["S"])\
            or (rps_choices[i] == self.options["P"] and rps_choices[j] == self.options["R"])\
            or (rps_choices[i] == self.options["S"] and rps_choices[j] == self.options["P"]):
        print(self.message["decision"].format(i, j))
        return i
    else:
        print(self.message["decision"].format(j, i))
        return j
```

Since the project also includes a coin toss, its function produces a random return value.

```
def toss_coin(self, trial):
    toss = random.randint(0, 1)
    return toss
```

8.3 Higher Order Functions - Functions Parameters and Return Values

Higher order functions allow a function as input parameter or return value.

In Python map() is an example for a function which takes a function as parameter.

```
# toss a coin
toss_list = []
for i in map(self.toss_coin, range(3)):
    toss_list.append(i)
```

Another example for a function which returns a function is shown in the next section about closures.

8.4 Closures

A closure is a function which remembers variables in enclosing scopes even if the block has finished executing. In the example below the inner function print_message() is returned by the outer function print_winner(). The returned inner function is assigned to the variable winner_message and it persists within its variables after the outer function has been exited.

```
def print_winner(self, starter):
    message = "\nOk, the coin shows {}, so {} will start. Let's play Tic Tac Toe.\n"\
        .format(starter[0], starter[1])

    def print_message():
        print(message)

    return print_message
```

```
# select the beginner
starter = starter_dict[toss]
winner_message = self.print_winner(starter)
winner_message()
```

8.5 Anonymous Functions

An anonymous function is declared without adding a name to it. An example for Python would be a lamda-function.

```
cum_toss = reduce((lambda x, y: x + y),toss_list)
```