A Guide to Writting Codex Exercises

Pedro Vasconcelos pbv@dcc.fc.up.pt, University of Porto, Portugal.

July 2019 (version 0.9.5)

1 Introduction

1.1 Overview

Codex is a web system for setting up programming exercises with automatic assessment. The main features are:

Simple exercise authoring The exercise statement is written human-readable plain-text file; that can easily be transfered, kept in a version repository, compared for changes, etc.;

Assessing program fragments Exercise can assess only individual functions, classes or methods rather than complete programs;

Provides automatic feedback Rather than report just an *accept/reject* result, Codex can use automatic testing techniques such as properties or documentation-tests to report failure example to students.

Multiple types of exercises Codex supports testing code, mutiple-choice and fill-in questionaries.

Codex is intended for learning environments rather than programming contests (for the later, check out Mooshak). The system is currently being used at the Faculty of Science of the University of Porto for teaching introductory courses on Python and C programming.

1.2 Pages and exercises

A Codex repository is organized as set of interlinked hypertext pages. *Pages* can contain text, links, tables, images and mathematical formulas written in Markdown.

Markdown is a lightweight text markup syntax designed by John Gruber. Codex uses the Pandoc library for parsing Markdown which suports many extensions. In particular, it support LaTeX mathematics (rendered using MathJaX) For

more details on the Pandoc-flavoured Markdown syntax accepted, check the Pandoc user manual.

Pages can be just plain documents or marked as *exercises*; the later allow student's submissions and trigger automatic assessment. Submissions are kept in a persistent database and students can review previous their submissions attempts (thought not anyone else's). The administrator can view and manage all submissions (e.g. re-evaluate or generate printouts).

Comments can be written in HTML-style:

```
<!-- This is the begining of comment;
     it can go on for multiple lines
-->
```

Note that, unlike other Markdown readers, raw HTML markup is **not** allowed (it will be rendered as plain text).

Codex uses Pandoc's YAML extension to specify metadata for pages, (e.g. set the type of execise, the programming languages allowed, the period for valid submissions, etc). Metadata blocks are delimited between "`---`" and "`...`":

```
---
tester: doctest
language: python
valid: "after 25/05/2019 and before 15/06/2019"
...
```

Such blocks can occur anywhere in a document; several metadata blocks are equivalent to a single block of all collected fields. (We suggest placing metadata only at the beginning or the the end of the document.)

Exercises are marked by the `tester` field; the particular tester specifies the type of exercise and how assement is done (see the section on Testers).

1.3 Document structure

All page files, images, etc. should be maintained inside the `public` directory and the starting page after login is the `index.md` page; this should link exercises or other sub-directories. For example, for 3 exercises `worksheet1.md`, `worksheet2.md` and `worksheet3.md`, a minimal `index.md` page could be:

```
# Welcome!

This is minimal index page. Here is a list of available exercises:

* [](worksheet1.md){.ex}
* [](worksheet2.md){.ex}
* [](worksheet3.md){.ex}
```

Links to exercise pages are marked with class attribute `.ex`; this fills-in automatically the exercise title (read from the markdown document) and a summary of previous submissions by the current user (read from the submission database).

The adminstrator can edit the index page to the order of exercises, or group exercises using sections and sub-pages. It is also possible to add plain Markdown pages for documentation, or links to external resources.

Note that:

1. Exercise pages can be accessed by their URL even if they are not explicitly linked to the index; this can be used test exercises before making them visible to students;
2. If you intend to group exercises using sub-directories make sure *you include an `index.md` for each sub-directory.*

2 Writting exercises

2.1 An example programming exercise

Consider the following simple Python programming exercise: *write a function `root` to compute rounded-down integral square roots.*

First we set up an *exercise page* `sqroot.md`; this contains the exercise description shown to the student (and some metadata fields):

```
---
tester: doctest
language: python
...

# Compute rounded-down integer square roots

Write a function `root(x)` that computes the square root
of an integer number rounded-down to the nearest integer
value.

If the argument `x` is negative the function should
throw a `ValueError` exception.
```

We also need a *doctest script* `sqroot.tst` specifying the tets cases to try and expected results:

```
>>> root(0)
0
>>> root(1)
1
```

```
>>> root(2)
1
>>> root(4)
2
>>> root(5)
2
>>> root(10)
3
>>> root(25)
5
>>> root(-1)
Traceback (most recent call last):
  ...
ValueError: math domain error
```

These two files are all that we need. For each student submission the tests in the *doctest* script will be tried in order; testing terminates immediately if any of tests fails (with a wrong answer or a runtime exception) and the failed test case is used to produce a report, e.g.:

```
Failed example:
    root(2)
Expected:
    1
Got:
    1.4142135623730951
```

Some remarks on writting *doctest* cases:

1. The first test case that fails is reported, thus so the *doctest* script should the include the simplest cases first;
2. Be aware that the `doctest` library employs a textual matching of outputs (e.g. `0` and `0.0` are considered distinct); make sure you normalize floating-point results (e.g. using `round`)
3. To discourage students from "overfitting" solutions to pass just the failing tests, it is best to generate a large number (50-100) of cases (write a Python script);
4. Alternatively, you can hide test cases by setting the metadata field "`feedback: no`";
5. It is also possible test error handling by requiring that proper exceptions are thrown — see the `doctest` documentation;

2.1.1 An example quiz

Setting up a multiple-choice or fill-in quiz requires only a Markdown exercise page:

```
---
tester: quiz
shuffle-questions: no
shuffle-answers: yes
...

# Example quiz

This is an example of multiple-choice and fill-in questions.

<!-- Each question starts with a header with class "question" -->

## {.question answer=a}

Consider a relation $R \subseteq \mathbb{Z}\times\mathbb{Z}$
definide by $x R y \iff \exists k \in\mathbb{Z} : y = k\times x$.
What properties does this relation have?

(a) reflexive, transitive e anti-symetric
(b) reflexive, transitive e symetric
(c) reflexive, not transitive and symetric

<!-- The next question allows multiple selections -->

## {.question .multiple answer=a answer=c answer=e}

Select **all** true statements from the choices bellow.

(a) Dogs have wings if cats have wings
(b) Birds have wings if cats have wings
(c) If cats have wings then birds have wings
(d) Snakes have legs if and only if mice have tails
(e) If frogs have fur and mice have eyes, then sharks have no teeth

<!-- A fill-in question; answers are compared textually ignoring spaces -->

## {.question .fillin answer="42"}

What is the answer to the fundamental question about ife, the universe
and everything?
```

2.1.2 Testing Haskell code using Quickcheck

A QuickCheck specification is a Haskell main module that acts as a "test harness" for the student's code. It can define properties and possibly data generators; the

Codex QuickCheck library includes default generators for basic types (tuples, lists, etc.) and combinators for building custom ones.

Consider an example exercise:

> Write a Haskell function `strong :: String -> Bool` that checks if a string is a *strong password* using the following criteria:
>
> 1. it must have at least 6 characteres in length;
> 2. it must have an uppercase letter, a lowercase letter and a digit.

The QuickCheck specification for this exercise is as follows:

```
module Main where

import Codex.QuickCheck
import Data.Char
import Submission (strong) -- student's solution

-- | reference solution
strong_spec :: String -> Bool
strong_spec xs
   = length xs >= 6 && any isUpper xs && any isLower xs && any isDigit xs

-- | a generator for suitable strings
asciiString = listOf (choose ('0', 'z'))

-- | correctness property: for all above strings,
-- the submission yields the same result as the reference solution
prop_correct
  = forAllShrink "str" asciiString shrink $
    \xs -> strong xs ?== strong_spec xs <?> "strong"

main = quickCheckMain prop_correct
```

Some remarks:

1. The student's code is always imported from a separate module; names should be explicitly imported or qualified to prevent name colisions;
2. `asciiString` is a custom generator for strings with charateres from `0` to `z`: this generate letters, digits and some other simbols as well;
3. The operator `?==` asserts that the left-hand side equals the (expected) right-hand side
4. The operator `<?>` names the test for reporting;
5. Finally, the `quickCheckMain` driver function handles setting up of the testing parameters and checks the property.

The use of shrinking simplifies failing test cases automatically; for example, consider the following submission exhibiting a common logical error:

```
forte :: String -> Bool
   = length xs >= 6 && any (\x -> isUpper x || isLower x || isDigit x) xs
   -- WRONG!
```

Because of shrinking, the specification above QuickCheck *always* finds a simple counter-example:

```
*** Failed! (after 12 tests and 6 shrinks):
Expected:
    False
Got:
    True
Testing forte with:
str = "aaaaaa"
```

2.1.3 Testing C code Using QuickCheck

It is also possible to use QuickCheck to test C code using the Haskell FFI. The test specification written in Haskell and uses the Haskell-to-C FFI to call the student's code.

Consider the C version of the strong password example:

> Write a C function `int strong(char *str)` that checks if a NUL-terminated string is a *strong password* using the following criteria:
>
> 1. it must have at least 6 characteres in length;
> 2. it must have an uppercase letter, a lowercase letter and a digit.
>
> The result should be 1 the string satifies the above criteria and 0 otherwise.

The previous specification is adapted to test C as follows:

```
module Main where

import Codex.QuickCheck
import Control.Monad
import Control.Exception
import System.IO.Unsafe
import Data.Char

foreign import ccall "forte" c_forte :: CString -> IO CInt
   -- use FFI to import students' submission

-- | functional wrapper over C code
c_forte_wrapper :: String -> CInt
c_forte_wrapper str = unsafePerformIO $
  withCString str $ \ptr -> do
```

```
     r <- c_forte ptr
     str' <- peekCAString ptr
     unless (str == str') $
          throwIO $ userError "modified argument string"
     return r

-- | functional specification
forte_spec :: String -> Bool
forte_spec xs
   = length xs >= 6 && any isUpper xs && any isLower xs && any isDigit xs

prop_correct
  = forAllShrink "str" asciiString shrink $
    \xs -> c_forte_wrapper xs ?== fromIntegral (fromEnum (forte_spec xs))
           <?> "forte"

asciiString = listOf (choose ('0', 'z'))

main = quickCheckMain prop_correct
```

Note that, along with function correctness, the wrapper code above also checks that student code does not overwrite the string buffer.

3 Reference guide

3.1 Results

Codex assesses submissions using a *tester*; for programming exercises this typically requires running a test suite (either provided by the instrutor or randomly-generated). The result is a *classification label*, a *validity check* and a *detail report*.

Classification labels are similar to the ones used for ICPC programming contests:

Accepted The submission passed all tests.

WrongAnswer The submission was rejected because it failed at least one test.

CompileError The submission was rejected because it caused a static error (e.g. compiler error)

RuntimeError The submission was rejected because it caused a runtime error (e.g. runtime exception, segmentation fault)

RuntimeLimitExceeded*, *MemoryLimitExceeded The submission was reject because it tried to use too much computing resources; this usually signals an erroneous program (e.g. non-terminating).

MiscError Some other error (e.g. incorrect metadata, test files, etc.)

Evaluating Temporary label assigned during evaluation.

Independentely of the classification above, the submission is marked *Valid* if it respects the constraints on time and maximum attempts (specified as metadata) and *Invalid* otherwise (with a suitable message).

Note that Codex always evaluates submissions (and reports feedback if is enabled); this means that a late submission that passes all tests will be classified *Accepted (Invalid: Late submission)*.[^1] This behaviour is intencional: it allows students to retry past exercises and leaves the instructor freedom on how to consider such submissions.

3.2 General metadata

The following metadata fields apply to all exercises types.

`title` Specifies the title for exercise links; if this is missing, the title is the first header in the document.

`tester` Specify the type of exercise (described in the following subsection).

`valid` Specify constraints on the time and maximum number for submission attempts; the default is no time constraint and an unlimited number of submssions. Some examples:

```
valid: "after 08:00 15/02/2019"
valid: "after 08:00 15/02/2019 and before 12:00 15/02/2019"
valid: "before 16/02/2019"
valid: "before 16/02/2019 and attempts 20"
```

Date and times are relative to the server timezone.

`feedback` A boolean value (`yes`/`true` or `no`/`false`) controlling whether detailed feedback is shown (e.g. show the test case in case of test failure); the default is feedback enabled.

`code` Specify an initial code skeleton for programming exercises; use an indented block for multiple lines, e.g.:

```
code: |
  ~~~
  def distance(x1, y1, x2, y2):
      #
      # complete this definition
      #
  ~~~
```

Note the vertical bar (`|`) and indentation in the above example.

3.3 Testers

3.3.1 `stdio`

Tests complete programs that read input from `stdin` and produce output on `stdout`; specific fields:

`languages` List all allowed languages for this exercise; e.g.:

```
languages: [c, java, python, haskell]
```

If only a single language is allowed, the singular option `language` may be used instead:

```
language: c
```

`inputs` List of input files in order; glob patterns are allowed; e.g.

```
inputs: [ "exercise1/tests/example-in.txt", "exercise1/tests/input*.txt"
]
```

`outputs` List of expected output files in order; glob patterns are allowed; e.g.

```
outputs: [ "exercise1/tests/example-out.txt", "exercise1/tests/output*.txt"
]
```

The number of output files much be the same as the number of inputs.

`files` List of extra files that will be copied to the temporary working directory while running the tests.

`arguments` List of files with command-line arguments for each of the test cases defined by input/output pairs; by default, the command-line arguments are empty.

3.3.2 `doctest`

Test Python code using the `doctest` library for unit-testing simple functions, methods or classes. Extra options:

`language` Should be `python` (must be included).

`tests` Optional filename for doctests; by default this is the same file as the exercise with `.tst` as extension.

`linter` Boolean value specifying whether to run an external compile-time linter such as an optional static type checker); the linter command is specified in the global configuration file; by default, the linter is not run.

`linter-args` Optional command-line arguments to pass the linter.

3.3.3 `quickcheck`

Test Haskell or C code using a custom version of the QuickCheck library. Extra metata options:

`language` Languages accepted: `haskell` or `c`.

`properties` Haskell file with QuickCheck specification (generators and properties); defaults to the page filename with the extension replaced by `.hs`.

`maxSuccess` Number of tests to run.

`maxSize` Maximum size for generated test data.

`maxDiscardRatio` For conditional generators: the maximum number of discarded tests per successful test before giving up.

`randSeed` Integer seed value for pseudo-random data generation; use if you want to ensure reproducibility of tests.

3.3.4 `quiz`

Multiple-choice and fill-in questionaries with optional shuffling of questions and answers. Extra options:

`shuffle-questions` Boolean value; specifies whether the order of questions should be shuffled; defaults to `false`. By default shuffling uses a deterministic algorithm based on the user login.

`shuffle-answers` Boolean value; specifies whether the order of answers within each question should be shuffled; default to `false`. By default shuffling uses a deterministic algorithm based on the user login.

`random-seed` Integer value; fix the PRNG seed used for shuffled instead of computing it from the user login.

`correct-weight` Fractional number from 0 to 1; this is the scoring weight for a correct item in a multiple-choice question. By default the correct item(s) score 1.

`incorrect-weight` Fractional number from 0 to 1; this is the scoring weight for an incorrect to a multiple-choice question. By default the score for each incorrect item is such that the sum of incorrect items equals minus the sum of corrects ones, e.g. for a three option question with a single correct option, an incorrect option scores $-1/2$. To change change the penalty for wrong opttion to $-1/3$:

```
incorrect-weight: -1/3
```

Pedro Vasconcelos, 2019.