Codex User Guide

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

January 2017 (version 0.8)

Codex is a web system for setting up programming exercises with automatic assessment. Codex is intended for learning environments rather than programming contests (for the later, Mooshak is a better tool). Its aims are:

- *simple exercise authoring*; seting a new exercise requires writing just two text files (a text description and test specification);
- *allow testing fragments* e.g. functions or classes rather than complete programs;
- *provide good automatic feedback*; rather than just run a handful of tests and report an *accept/reject* result, Codex uses state-of-art automatic testing tools to run tens or hundrends of tests; if an error is found, it reports a *short failure example.*

Althought Codex is stil in an early development stage, it is already in use at the Faculty of Science of the University of Porto for teaching introductory courses on Python programming.

This guide describes how to write exercises and test specifications for Codex.

1 Pages

Codex exercise repositories are organized into *pages.* A page can contain formated text, links, tables, images, mathematics, etc. Codex pages are simple text files with the `.md` extension; Markdown is used for formating.

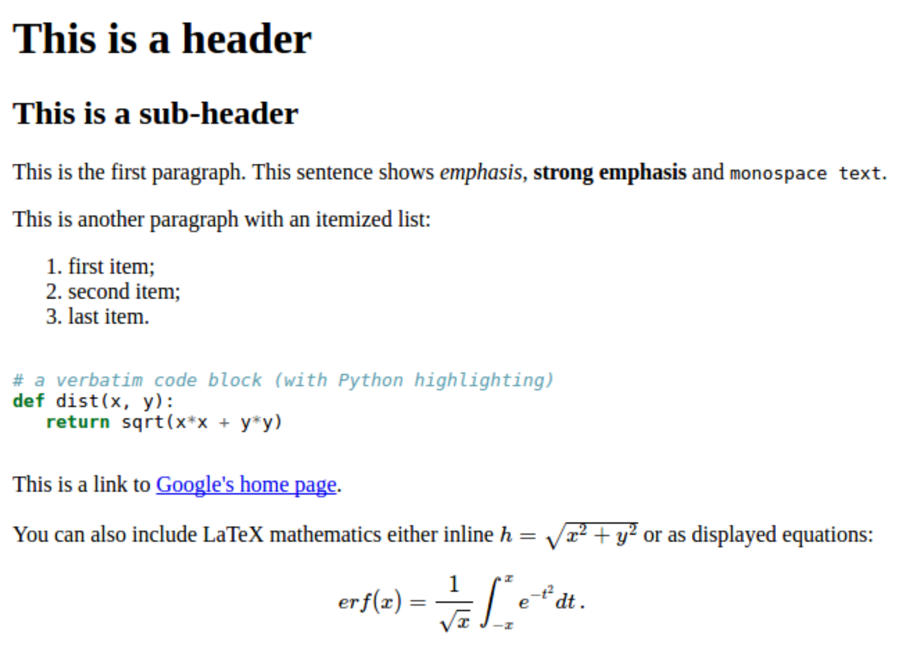
1.1 Markdown files

Let us start with an example page file:

```
# This is a header

## This is a sub-header
```

```
This is the first paragraph. This sentence shows *emphasis*,
**strong emphasis** and `monospace text`.

This is another paragraph with an itemized list:

1. first item;
2. second item;
3. last item.

~~~python
# a verbatim code block (with Python highlighting)
def dist(x, y):
   return sqrt(x*x + y*y)
~~~

This is a link to [Google's home page](http://www.google.com).

You can also include LaTeX mathematics either
inline $h = \sqrt{x^2+y^2}$ or as displayed equations:
$$ erf(x) = \frac{1}{\sqrt{x}}\int_{-x}^x e^{-t^2} dt\,. $$
```

The above text could be rendered into HTML as follows:



This is a header

This is a sub-header

This is the first paragraph. This sentence shows *emphasis*, **strong emphasis** and `monospace text`.

This is another paragraph with an itemized list:

1. first item;
2. second item;
3. last item.

```python
# a verbatim code block (with Python highlighting)
def dist(x, y):
   return sqrt(x*x + y*y)
```

This is a link to Google's home page.

You can also include LaTeX mathematics either inline $h = \sqrt{x^2 + y^2}$ or as displayed equations:

$$erf(x) = \frac{1}{\sqrt{x}} \int_{-x}^{x} e^{-t^2} dt\,.$$

Figure 1: Example page rendering

Codex uses the Pandoc library for reading and rendering Markdown and MathJaX for displaying mathematics. For details on the Markdown syntax accepted, check the Pandoc user manual. Note that (unlike the usual Markdown behaviour) raw HTML markup commands will be escaped and rendered as ordinary text; this ensures that the generated HTML pages are always well-formed.

1.2 Metadata blocks

Markdown text can also include YAML metadata blocks delimited between 3 dashes (`---`) and 3 full stops (`...`); for example:

```
---
author: Pedro Vasconcelos
title: A sample exercise
exercise: true
language: python
...
```

Metadata blocks can occur anywhere, but the convention is to put them at the beginning of the document. Several metatata blocks are allowed and equivalent to a single one with all collected fields.

Some fields (like `author` and `title`) are generic, while others (like `exercise` and `language`) are specific to exercise testing in Codex. These are described in detail in a later section.

1.3 Exercise pages

A page marked with metadata `exercise: true` is an *exercise page*; this means that users will be able to:

- *submit* solutions for automatic assessment;
- *get feedback* on their submissions;
- *view past submissions* and feedback;
- *edit and re-submit* past submissions.

Users must be logged-in to view pages and submit solutions; other than that, users can submit any number of attempts for any exercise. Previous submissions are kept in a persistent disk database; only the adminstrator can remove or re-evaluate submissions.

Note that an exercise is identified by the exercise page's *request path* relative to the `/pub` handle; e.g. an exercise at URL `https://server.domain/pub/foo/bar.md` is identified as `foo/bar.md`. This means that the adminstrator is free to edit the exercise file and/or tests even after submissions have started (e.g. to correct errors); on the other hand, if the path is modified, any previous submissions are still recorder, but will no longer be associated with the modified exercise.

1.4 Linking exercise pages

The initial view for users is the `index.md` page at the root public directory; the adminstrator should edit this page to link other pages and exercises.

For example, supose you have created 3 exercises `work1.md`, `work2.md` and `work3.md`; a minimal `index.md` page could be:

```
# Welcome!

Here is a list of available exercises:

1. [](work1.md){.ex}
2. [](work2.md){.ex}
3. [](work3.md){.ex}
```

Exercise links should be marked with a special class `.ex`; this automatically fills the link anchor text with the exercise title. A short summary of previous submissions done by the logged-in user is also added.

The adminstrator can edit the index page to choose 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 exercise pages can be accessed and submitted even if they are not explicitly linked to the index page by simply typing the URL directly in the browser (after a login). You can use this feature to test new exercises before making them visible for users.

1.5 Exercise metadata fields

`title` Specify a title for exercise links; if this is field missing, the first header in the document is used instead.

`exercise` Mark a page as an exercise (true/false); this should be complemented by specifying the language and test cases.

`language` Specify the programming language for an exercise, e.g. `python`, `haskell`, `c`

`valid` Specify valid submission time interval; the default is `always` which means submissions are always valid. Some alternatives:

- `after 08:00 15/02/2017`
- `between 08:00 15/02/2017 and 12:00 15/02/2017`
- `until 16/02/2017`

Note that date (DD/MM/YYYY) and times (HH:MM) are relative to the server local timezone.

`feedback` Specify the level of feedback to report (0-100); 0 means no feedback, 50 shows classifications only, 100 shows classifications and failed test cases (default).

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

```
code: |
  ~~~
  # distance between two points in the plane
  def distance(x1, y1, x2, y2):
      # complete this definition
  ~~~
```

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

The following fields are specific to programming languages.

1.5.1 Python-specific fields

`doctest` Specifies the file path for a *doctest* script for testing Python submissions; if omitted, this defaults to the file name for exercise page with extension replaced by `.tst`, e.g. the default doctest for `foo/bar.md` is `foo/bar.tst`.

1.5.2 Haskell- and C-specific fields

`quickcheck` Specifies the file name for a Haskell file containing QuickCheck script for Haskell or C submission testing.

`maxSuccess` Number of QuickCheck tests run.

`maxSize` Maximum size for QuickCheck generated test data.

`maxDiscardRatio` Maximum number of discarded tests per successful test before giving up.

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

2 Assement and Feedback

Codex assesses submissions by testing them against test cases (either provided by the author or randomly-generated). The result is a *classification label*, a *timing label* and a (possibly empty) *detail text report*. Classification labels are similar to the ones used for ICPC programming contests (e.g. *Accepted*, *WrongAnswer*, etc.).

When submissions are rejected because of wrong answers, the text report includes a human-readable description of a failed test case; this is intended for the student to use as a starting point for understanding the problem and debugging.

Note that Codex will *always* evaluate submissions (and report feedback if enabled) regardless of the time interval specified in the exercise; however:

- it will hide feedback for early submissions until the start of submission interval;
- late submissions are assessed as usual but additionally labelled *Overdue*[1].

2.1 Feedback

2.1.1 Classification labels

Accepted The submission passed all tests.

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

CompileError The submission was rejected because it caused a compile-time 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 while evaluation is pending; end-users should never see this.

2.1.2 Timing labels

Early Received before the start of submission interval.

Valid Received within the valid submission interval.

Overdue Received after the end of the submission interval.

[1]This behaviour allows both students to retry past exercises and leaves teachers freedom to decide how to rate late submissions.

2.2 Specifying test cases

2.2.1 Python

Test cases for Python submissions are specified using the *doctest* library.

Consider an hypothetical simple exercise: compute rounded-down integral square roots. The exercise file `root.md` could be as follows:

````
---
exercise: true
language: python
doctest: root.tst
...

# 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.

Examples:

~~~
>>> root(4)
2
>>> root(5)
2
>>> root(10)
3
~~~
````

Note that we illustrate some expected behaviour examples mimicking the Python shell interaction.

The metadata field `doctest` specifies a *doctest* script with input values and output results; this a separate text file `root.tst`:

```
>>> 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
```

For each student submission, the above tests 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 the student report, e.g.:

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

Some advices:

- order the test cases such that simplest input values occur first;
- be aware that *doctest* employs a straight textual matching of outputs (e.g. `0` and `0.0` are distinct);
- make sure you normalize floating-point results to avoid precision issues, e.g. use `round`(. . . , *n-decimals*)
- to discourage students from "fixing" submissions by copy-pasting failed tests, it is best to gerate a large number (50-100) of test cases (write a Python script);
- you can test the correct handling of invalid situations by requiring that proper exceptions are thrown;
- you can control the level of feedback using the `feedback` metadata field (e.g. omit actual test inputs).

2.2.2 Haskell

Haskell submissions can be tested using the QuickCheck library. A QuickCheck specification consists of a set of *properties* and possibly *test data generators*; the QuickCheck library includes default generators for basic types, tuples, lists, etc.

Consider the an example exercise: merging elements from two ordered lists; the page text file is as follows:

```
---
```

```
exercise: true
language: haskell
quickcheck: merge.hs
...

# Merge two ordered lists

Write a function `merge :: Ord a => [a] -> [a] -> [a]`{.haskell}
that merges two lists in ascending order; the result list
should preserve the ordering and contain all elements from
both lists.
```

The QuickCheck script `merge.hs` is a straightforward translation of the above specification:

```haskell
import Test.QuickCheck
import Data.List ((\\))

-- 1) merge preserves ordering
prop_ordered :: OrderedList Int -> OrderedList Int -> Bool
prop_ordered (Ordered xs) (Ordered ys)
    = ascending (Submit.merge xs ys)

-- 2) merge preserves elements
prop_elements :: OrderedList Int -> OrderedList Int -> Bool
prop_elements (Ordered xs) (Ordered ys)
    = permutation (Submit.merge xs ys) (xs ++ ys)

-- auxiliary definitions
-- check if a list is in ascending order
ascending xs = and (zipWith (<=) xs (tail xs))
-- check if two lists are permutations
permutation xs ys = null (xs \\ ys) && null (ys \\ xs)
```

Some remarks:

- the user submission is implicitly imported qualified as a module `Submit`;
- all properties (e.g. functions starting with `prop_`) will be tested;
- note that although `merge` is polymorphic, we need to choose a monomorphic type for testing (`Int`);
- we used the default generators for `Int` and `OrderedList` wrapper defined in the QuickCheck library (no need to define a custom generator);

2.2.3 C

TO BE DONE

Pedro Vasconcelos, 2017.