# microanalyst Documentation

***Release 1.0.0***

**Bartosz Zaczyński**

June 03, 2016

Contents

Contents:

CHAPTER 1

# Introduction

## 1.1 About

Microanalyst is both a Python module and a set of utility scripts for the analysis and visualization of data acquired with Tecan® i-control™ microplate reader.

## 1.2 Motivation

The original experiment which led to the inception of this project was held at the Institute of Environmental Sciences of the Jagiellonian University.

The goal of the experiment was to determine genes whose expression gives a strong effect on survival of non-dividing cells in a repeatable manner.

There were around 6,000 yeast strains (*Saccharomyces cerevisiae*) with different sets of genes. Yeast extinction and survival would be observed by keeping the samples in a carbon-free environment. The entire collection of 65 microplates (8 rows by 12 columns each) was measured at roughly two-weeks time intervals and repeated in a few redundant series.

### 1.2.1 Microplates

- There were 65 microplates in the whole experiment.
- Each microplate had a unique name.
- Each microplate consisted of 96 wells with samples.
- Wells were arranged in 8 rows denoted with letters [A-H] and 12 columns denoted with numbers [1-12].
- Some microplates did contain empty control wells, whose presence and location had been determined by the manufacturer upfront.

### 1.2.2 Spreadsheets

- Microplates were scanned with a Tecan® Infinite® 200 reader device, which produced .xls files (Microsoft® Excel™).
- A single Excel™ document could contain readouts taken over a period of multiple days, although typically there would be a one-to-one relationship, i.e. one Excel™ file per day.

- It took approximately 40 seconds to scan one microplate.
- The readout of a single microplate included the following:
  - name of the microplate
  - date and time
  - temperature in Celsius degrees (usually around 30 degrees)
  - 96 floating point values (e.g. associated with the absorbance - optical density - at a particular wavelength and bandwidth)
- Name of an Excel™ sheet coincided with the corresponding microplate name.
- Additional sheets with metadata were ignored.
- Excel™ sheets were stored in no particular order.

### 1.2.3 Interpretation

- Values below 0.2 were considered starvation of a sample.
- Values above 0.8 were most likely an unwanted infection.
- Values above 0.06 in a control well also indicated a high probability of an infection in the microplate since that is the absorbance of the material comprising the microplates.

CHAPTER 2

# Downloads

This is an open-source project distributed as a public repository on GitHub. Follow the link and use the download button to get a fresh copy of microanalyst.

Alternatively if you prefer the command line and have previously installed git client:

```
$ git clone https://github.com/bzaczynski/microanalyst.git
```

CHAPTER 3

# Installation Instructions

Before commencing the analysis of data your work environment needs to be prepared. Depending on the platform and operating system these steps can vary slightly. Despite only Linux and Windows operating systems have been covered below a savvy Mac user should be able to follow without problems.

## 3.1 Python Interpreter

Microanalyst is distributed in the form of source code files intended to be executed by a Python interpreter. There are currently two major versions of the language, i.e. Python 2 and Python 3 which are largely incompatible. You want to make sure to be using the latest release of Python 2 which is going to be Python 2.7.x.

In addition to this a specific interpreter version may be available either for 32-bit or 64-bit computer architecture. However, it is recommended to choose the 32-bit one even on a 64-bit computer due to better availability of extension modules.

### 3.1.1 Linux

Chances are that Python is already installed. Otherwise use your distribution's package manager such as apt-get available on Debian/Ubuntu.

```
$ sudo apt-get install python
```

### 3.1.2 Windows

Open your favorite web browser and navigate to Python download page for a list of available options. There is an easy to use Python installer which you can and should take advantage of. To check if Python has installed correctly open Windows command prompt and type *python* (e.g. press `Windows + R`, then type *cmd* and confirm with `Enter`):

```
C:\> python
```

You should be presented with something similar to this:

```
Python 2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If that doesn't work it may be necessary to manually modify `Path` system variable in order to make `python` a recognizable command. On Windows 7 environment variables can be accessed by following these few steps:

1. Right click *My Computer* icon and select *Properties*.

2. Choose *Advanced system settings* from the menu on the left.
3. Click *Environment variables...* button.
4. Find system variable which says *Path* and double click it.
5. Append a semicolon followed by the path to the Python interpreter.

By default Python installs under `C:\Python27` directory. Unless otherwise you should simply append `;C:\Python27` to the variable.

## 3.2 External Modules

Microanalyst depends on some external modules, e.g. for processing Microsoft® Excel™ files, which are not part of the standard Python installation. They need to be installed manually, namely:

- `xlrd` for reading .xls files
- `xlwt` for writing .xls files
- `NumPy` for scientific computing

The easiest way of adding modules and packages to Python is with a package manager such as *easy_install* (which is a part of setuptools) or *pip*. Unfortunately Python comes with none of them by default so a package manager needs to be installed before.

### 3.2.1 Linux

On Linux certain popular Python modules may be available directly through system's package manager like apt-get, e.g.:

```
$ sudo apt-get install python-xlrd python-xlwt python-numpy
```

### 3.2.2 Windows

The recommended way of installing easy_install on Windows is by downloading and executing ez_setup.py script from the setuptools page. Afterwards, the `Path` system variable must be modified again. This time with a path to the `Scripts` subfolder:

```
;C:\Python27\Scripts
```

---

**Hint:** To improve maintainability one can create a custom system variable (e.g. called `PYTHON_PATH`) with all paths related to Python and refer to it in the `Path` variable instead of using literals, e.g.:

```
PYTHON_PATH=C:\Python27;C:\Python27\Scripts
Path=(...);%PYTHON_PATH%
```

---

Finally, to install the required modules:

```
C:\> easy_install xlrd xlwt numpy
```

Alternatively one can try an unofficial module installer provided by a 3rd party such as the one here. Some external Python modules are only distributed in source form and require a number of additional tools for compilation, which is a hassle particularly on Windows. Precompiled binary distributions of such modules are much easier to install on the other hand.

## 3.3 Installing microanalyst

Microanalyst is comprised of a Python module under the same name as well as a number of scripts which are built on top of it. To install microanalyst follow the standard procedure and type this command (works the same way on Windows and Linux, though root/sudo access may be needed for the latter):

```
C:\> cd microanalyst
C:\microanalyst> python setup.py install
```

## 3.4 Testing the Installation

To verify if the installation was successful you can exercise a comprehensive unit test suite. Change directory to `test/` subfolder and use the test discovery feature as shown below:

```
C:\microanalyst> cd test
C:\microanalyst\test> python -m unittest discover -v
```

## 3.5 Generating Documentation

The following document can be generated automatically with the Sphinx tool.

### 3.5.1 Linux

Ensure that `make` and `python-sphinx` are installed, then follow these steps to build the documentation:

```
$ cd microanalyst/doc
$ make html
```

### 3.5.2 Windows

Install Sphinx first:

```
C:\> easy_install sphinx
```

Use the `sphinx-build` command to generate documentation and place it under `build` folder:

```
C:\> cd microanalyst\doc
C:\microanalyst\doc> sphinx-build source build
```

CHAPTER 4

# Getting Started

Microanalyst serves a few different purposes which naturally translate into a sequential workflow:

1. Collecting data from various Tecan® spreadsheets comprising an experiment
2. Augmenting it with external metadata such as a map of genes and control wells
3. Value clustering (e.g. binarization)
4. Visualization
5. Analysis

Please read on to learn about those topics in more detail.

CHAPTER 5

# Data Model

Familiarity with data model used in microanalyst is central to understanding the usage of the tool. Virtually all information regarding an experiment is kept in a single JSON file acting as a container for hierarchical data. The reasons for choosing JSON over alternative file formats were the following:

- human readability
- portability
- compactness
- simplicity
- easily serieslized / deserieslized
- plain text rather than binary

The downside of using a text file instead of binary one is redundancy which makes the file grow rapidly in size. This can be mitigated by the choice of a more compact format such as YAML. However, benchmarks have shown that despite being about half the size of a similar JSON file the serialization of YAML takes an order of magnitude more time.

## 5.1 Schema

The structure of a document with experiment data as well as value constraints are formally defined by the follwing JSON schema. Scroll down for examples.

```
{
    "$schema": "http://json-schema.org/schema#",
    "description": "Experiment data collected with microanalyst.",
    "definitions": {
        "control": {
            "description": "Control wells (defined statically per whole iteration or a particular spr
            "type": "object",
            "additionalProperties": {
                "description": "Microplate names, e.g. 001 or B002.",
                "type": "array",
                "maxItems": 96,
                "items": {
                    "description": "Well addresses between A1 and H12.",
                    "type": "string",
                    "pattern": "^[A-H]([1-9]|10|11|12)$"
                }
            }
```

```
        }
    },
    "type": "object",
    "required": ["iterations"],
    "properties": {
        "genes": {
            "description": "Genes used in the experiment.",
            "type": "object",
            "additionalProperties": {
                "description": "Microplate names, e.g. 001 or B002.",
                "type": "object",
                "additionalProperties": false,
                "patternProperties": {
                    "^[A-H]([1-9]|10|11|12)$": {
                        "description": "Well addresses between A1 and H12 mapped to gene names, e.g.
                        "type": "string"
                    }
                }
            }
        },
        "iterations": {
            "description": "An ordered list of consecutive experiment iterations.",
            "type": "array",
            "items": {
                "description": "A single iteration comprising multiple spreadsheets, e.g. taken at 1
                "type": "object",
                "required": ["spreadsheets"],
                "properties": {
                    "control": {
                        "description": "Static control wells defined for all spreadsheets of this ite
                        "$ref": "#/definitions/control"
                    },
                    "spreadsheets": {
                        "description": "An ordered list of metadata objects describing consecutive sp
                        "type": "array",
                        "items": {
                            "description": "Metadata object with information about the original sprea
                            "type": "object",
                            "required": ["filename", "microplates"],
                            "properties": {
                                "control": {
                                    "description": "Additional control wells developed at a certain p
                                    "$ref": "#/definitions/control"
                                },
                                "filename": {
                                    "description": "Full name of the original Microsoft(R) Excel(TM)
                                    "type": "string"
                                },
                                "microplates": {
                                    "description": "A set of microplates in this spreadsheet.",
                                    "type": "object",
                                    "additionalProperties": {
                                        "description": "Microplate names, e.g. 001 or B002.",
                                        "type": "object",
                                        "required": ["timestamp", "values"],
                                        "properties": {
                                            "temperature": {
                                                "description": "Temperature at the time of the readou
```

```
                                                "type": "number"
                                            },
                                            "timestamp": {
                                                "description": "Date and time of the readout.",
                                                "type": "string",
                                                "pattern": "^\\d{4}-\\d{2}-\\d{2}T\\d{2}:\\d{2}:\\d{
                                            },
                                            "values": {
                                                "description": "An array of microplate values in row-
                                                "type": "array",
                                                "minItems": 96,
                                                "maxItems": 96,
                                                "items": {
                                                    "type": "number"
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

### 5.1.1 Root

The root element is an object `{}` with `iterations` being the only mandatory child, while `genes` remain optional. Example:

```
{
   "iterations": [],
   "genes": {},
}
```

### 5.1.2 Genes

This is a global map of genes (proteins) scattered over microplates and their wells. It is assumed that this mapping remains consistent throughout the entire experiment and does not change in any of the iterations. The `genes` object maps microplates' names to wells' addresses (from A1 to H12). Then each well address is mapped to a gene name which can be an arbitrary text string. A microplate typically has only a subset of its wells assigned to genes. Example:

```
"genes": {
   "001": {
       "F5": "carotene"
       "A1": "collagen",
   },
   "007": {
       "G4": "myosin"
   }
}
```

### 5.1.3 Control

There are specially designated wells on some microplates (determined by the manufacturer) that remain empty. This is to allow for validating optical density values of regular wells against noise such as infections. Control wells can be defined per iteration but also per a particular spreadsheet within an iteration. In both cases they are optional. Example:

```
"control": {
   "001" [
       "A1", "A2", "A3"
   ],
   "002": [
       "A1", "A2"
   ]
}
```

### 5.1.4 Iterations

This is an array `[]` of objects representing subsequent experiment series with Tecan® files and additional metadata described later. Note that the order of iterations must correspond to their actual sequence in time. Example:

```
"iterations": []
```

### 5.1.5 Iteration

An iteration is an anonymous object which must define `spreadsheets` array and may also define `control` wells property. Example:

```
{
   "spreadsheets": [],
   "control": {}
}
```

### 5.1.6 Spreadsheets

This is an array `[]` of objects encapsulating key data from Tecan® files in chronological order. Example:

```
"spreadsheets": []
```

### 5.1.7 Spreadsheet

A spreadsheet is an anonymous object which must define `filename` (absolute path) corresponding to a Microsoft® Excel™ file and `microplates` object. It can also define additional `control` wells which will only become available in this particular spreadsheet. Example:

```
{
   "filename": "C:\\experiment\\series2\\GAL_s02_21days.xls",
   "microplates": {},
   "control": {}
}
```

### 5.1.8 Microplates

Microplates is an object `{}` whose keys are microplates' names. Example:

```
"microplates": {
   "001": {},
   "002": {}
}
```

### 5.1.9 Microplate

A microplate is an instance of a given microplate identified by its unique name and scanned at a particular point in time. It belongs to a spreadsheet within experiment series/iteration. It defines ISO 8601 `timestamp`, Celsius degrees `temperature` and 96 floating point `values`. Example:

```
"B002": {
   "timestamp": "2014-01-13T12:49:03",
   "temperature": 24.0,
   "values": [
       0.7184000015258789,
       0.6804999709129333,
       0.6837000250816345,
       (...)
   ]
}
```

CHAPTER 6

# Command Line Interface

## 6.1 Overview

Generating data model in JSON file format can be done with the help of a few Python scripts. If installed correctly the scripts will be recognized and available in the command line without any prefixing. For a graphical user intreface (GUI) wrapper please refer to the next chapter.

By default scripts expect data from standard input and print to the standard output, which allows for combining them in a pipeline. Alternatively intermediate JSON structure can be passed between the scripts through file redirection.

### 6.1.1 Windows

```
C:\> type input.json | foo.py > output.json
```

### 6.1.2 Linux / Mac OS

```
$ cat input.json | foo.py > output.json
```

## 6.2 Grouping Tecan® Spreadsheets

When starting from scratch with data collection the first step is to group the original Excel™ file names into a stub JSON structure which represents consecutive experiment iterations. Each iteration is comprised of files and additional metadata. This can be done either manually or using the `group.py` script. For convenience you may put spreadsheet files under different folders to take advantage of pipelining, e.g.:

```
$ group.py series1/*.xls | group.py series2/*.xls | group.py series3/*.xls
```

The order of file groups is determined from left to right when calling the script multiple times with a pipe `|`. Since this will be the entry point of another script you may redirect standard output to a file instead.

This would print the following JSON to the standard output provided that each of the folders contains relevant files:

```
[
   {
       "files": [
           "series1/series1_14days.xls",
           "series1/series1_28days.xls",
```

```
            "series1/series1_42days.xls",
            "series1/series1_56days.xls"
        ]
    },
    {
        "files": [
            "series2/series2_14days.xls",
            "series2/series2_28days.xls",
            "series2/series2_42days.xls",
            "series2/series2_56days.xls"
        ]
    },
    {
        "files": [
            "series3/series3_14days.xls",
            "series3/series3_28days.xls",
            "series3/series3_42days.xls",
            "series3/series3_56days.xls"
        ]
    }
]
```

Note this is an intermediate structure which is different from the final JSON data model described *earlier*. For example file paths can be relative at this point - they will be resolved into absolute paths at a later stage. Unicode characters and path separators of either type are accepted to achieve portability but non-ASCII characters must be escaped with Unicode NFC escape sequences while backslash with a double \\.

Custom properties can be defined to annotate groups of files. While not recognized explicitly they are retained and can be accessed later. Example:

```
C:\> group.py tecan\*.xls --species "Saccharomyces cerevisiae" --version 3
```

```
[
    {
        "files": [
            "tecan\\starvation_collection_7days.xls",
            (...)
        ],
        "species": "Saccharomyces cerevisiae",
        "version": 3
    }
]
```

It is worth noting that the result of calling `group.py` as well as many other scripts distributed with microanalyst is simply plain text in JSON format.

## 6.3 Adding Control Wells per Iteration

**Note:** This step is optional.

Control wells can be added to all or some iterations by appending a `control` property to the intermediate JSON structure. Its contents should conform to the JSON schema outlined before. Example:

```
[
    {
```

```
        "files": [
            "series1/series1_14days.xls",
            "series1/series1_28days.xls",
            "series1/series1_42days.xls",
            "series1/series1_56days.xls"
        ],
        "control": {
            "002": [
                "A1"
            ],
            "006": [
                "A4", "D5", "E7"
            ]
        }
    },
    (...)
]
```

To automate the process of adding control wells and for greater flexibility control wells can be kept in separate JSON files for each iteration, such as this one:

```
{
    "002": ["A4"],
    "006": ["A4", "D5", "E7", "F3", "G3", "G8", "H12"],
    "B001": ["H1", "H7"],
    "B002": ["G2", "H5", "H6", "H7", "H8", "H9", "H12"],
    (...)
}
```

Then the contents of each of those external JSON files can be quickly pulled in and put under its corresponding iteration by calling the `control.py` script:

```
C:\> group.py series1/*.xls | group.py series2/*.xls | control.py series1/ctrl.json series2/ctrl.jso
```

The order of this script's parameters determines which iteration to put the control wells into. Note that the number of iterations on the left of the pipeline must be the same as the number of control well files on the right. Otherwise a missing control well file must be indicated with a dash sign to explicitly omit certain iterations, e.g.:

```
C:\> (...) | control.py series1/ctrl.json - series3/ctrl.json
```

## 6.4 Assembling Experiment Data

Up until now the JSON produced by previous scripts or perhaps edited manually merely contained file references and optional metadata. To translate this information into a standalone data model with all samples from Tecan® spreadsheets, e.g. for further processing and analysis, it needs to be assembled. Assembly phase is about reading the original files (without altering them) and putting their contents in a single JSON object according to the schema, while retaining any additional metadata defined inside the iterations of the intermediate JSON. The data is parsed, made uniform and unambiguous (e.g. relative file paths are turned into absolute ones) and optimized for performance.

```
C:\> (...) | assemble.py > experiment.json
```

```
{
    "iterations": [
        {
            "control": {
                "B001": ["H1", "H7"],
```

```
                (...)
            },
            "spreadsheets": [
                {
                    "control": {
                        "002": ["A4"],
                        (...)
                    },
                    "filename": "ex2\\series1\\GAL_s01_21days.xls",
                    "microplates": {
                        "001": {
                            "temperature": 23.6,
                            "timestamp": "2014-01-13T12:43:19",
                            "values": [
                                0.7384999990463257,
                                0.7184000015258789,
                                (...)
                            ]
                        },
                        (...)
                    }
                },
                (...)
            ]
        },
        (...)
    ]
}
```

The resultant JSON file is the basis for later experiment evaluation.

## 6.5 Adding a Map of Genes

Genes are fixed for the entire experiment so they do not belong to any iteration. As a result of that adding genes only makes sense after the assembly phase when the final JSON has a root element `{}`. Example:

```
{
    "genes": {
        "001": {
            "A1": "Q0085",
            "A2": "YDR034C-A",
            "A3": "tORF13",
            (...)
        },
        (...)
    },
    "iterations": [
        (...)
    ]
}
```

There is a counterpart script to `control.py` for including genes in the JSON data model from an external file such as this one:

```
{
    "001": {
        "A1": "Q0085",
```

```
        "A2": "YDR034C-A",
        "A3": "tORF13",
        (...)
    },
    "002": {
        (...)
    },
    (...)
}
```

It is called `genes.py` and can be used in the following way:

```
C:\> type experiment.json | genes.py yeast.json
```

Note there can be at most one external file with genes since the script overwrites the `genes` JSON property with the last file specified.

## 6.6 Quantizing Data Samples

Sometimes it is useful to apply a filtering function on a continuous range of values (such as real numbers) so that they are mapped to a discrete set of values. This can dramatically reduce the complexity of a problem domain. Microanalyst comes with a `quantize.py` script which can be used for clustering of data samples measured by a microplate scanner. It recognizes a few types of data samples, based on the metadata provided in the model as well as their actual values, which can be mapped to arbitrary integer numbers. Here they are with the corresponding default values to map to (if not otherwise specified by the user):

| Type | Meaning | Threshold | Argument | Default |
|---|---|---|---|---|
| control | Indicates a possible infection if not empty | N/A | `--control` | 2 |
| neutral | Living microorganisms (including infection) | N/A | `--other` | 1 |
| empty | Dead microorganisms | x <= 0.2 | `--starved` | 0 |

In spite of being technically possible there is no facility as of now to declaratively override default thresholds for recognizing starved, infected and violated control wells which were mentioned in the introduction.

Custom properties are retained in the result.

If one is only interested in starvation then control and other types of data samples can be disregarded by choosing a common value such as zero for the mapping, while emphasizing empty wells with a different value, e.g. one. This produces a binary view of the experiment, which is very convenient for pattern discovery.

```
C:\> type experiment.json | quantize.py --control 0 --other 0 --starved 1
```

```
{
    "iterations": [
        {
            "spreadsheets": [
                {
                    "filename": "/experiment/series1/series1_14days.xls",
                    "microplates": {
                        "001": {
                            "temperature": 23.3,
                            "timestamp": "2013-04-02T12:47:14",
                            "values": [
                                0,
                                0,
                                1,
                                0,
```

```
                                    0,
                                    0,
                                    0,
                                    (...)
                                ]
                            },
                (...)
    ]
}
```

To invert neutral and empty data sample values but keep the information about control wells:

```
C:\> type experiment.json | quantize.py --other 0 --starved 1
```

## 6.7 Exporting to Microsoft® Excel™

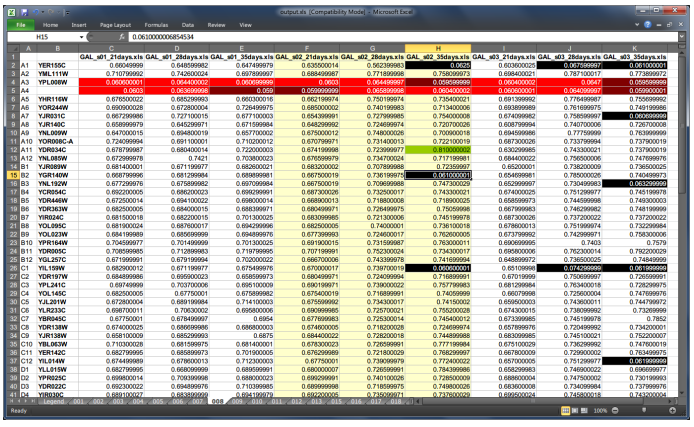
Assembled JSON can be visualized in the form of a spreadsheet with color-coded cells to give you a big picture of experiment data at a glance as well as for quick lookup.

### 6.7.1 Templates

There are two templates to choose from, i.e. **horizontal** with each microplate kept in a separate tab and a **vertical** one where all microplates are stored in a single tab. The former can be helpful in individual microplates comparison, whereas the latter in looking for global patterns. The scripts for representing the model with horizontal and vertical layouts are `xlsh.py` and `xlsv.py` respectively.

Generating visual representation with the horizontal layout from a JSON model could look like this:

```
C:\> type experiment.json | xlsh.py output.xls --colors
```



If the output file already exists it can be forced to be overwritten by specifying the `-f` flag.

### 6.7.2 Genes

Gene names are automatically rendered next to well addresses on pertinent microplates when available in the model.

### 6.7.3 Colors

Colors are disabled by default resulting in a less cluttered view. To enable them one must use `--colors` boolean flag, which also enables legend rendering (unless binary mode is enabled at the same time with `--binary` switch).

### 6.7.4 Binary mode

Binary mode enhances rendering of binary data samples, i.e. values belonging to `{0, 1}` set, by formatting the cells as integers rather than floating-point values. When colors are enabled in binary mode then usual thresholds for detecting different well types (starved, infected, etc.) are disregarded. Instead (a) ones, (b) zeros and (c) the rest of values are color coded.

To make the most out of binary mode one should utilize `quantize.py` script in the pipeline:

```
C:\> type experiment.json | quantize.py | xlsh.py output.xls --binary --colors
```

### 6.7.5 CSS Style

The default color-coding scheme (shown below) emphasizes zeros in binary mode since these typically manifest some interesting properties. However, if this is not what you want a custom style can be defined with a CSS-like syntax to override part or all of the default stylesheet. A custom CSS file may be loaded with the `--style` parameter (ignored if `--colors` flag is missing):

```
C:\> type experiment.json | xlsh.py output.xls --colors --style custom.css
```

To invert the rendering of ones and zeros in binary mode:

```
.zero {
}

.one {
    background-color: lime;
    color : dark_green;
}
```

```
C:\> type experiment.json |
     quantize.py --control 0 --other 0 --starved 1 |
     xlsh.py output.xls --binary --colors --style invert.css
```

### 6.7.6 Default Style

```
* {
    border-color: gray25;
}

.default, .one {
}

.alt {
    background-color: ivory;
}

.header {
    font-weight: bold;
```

```
}

.centered, .binary {
    text-align: center;
}

.binary {
    number-format: 0;
}

.starved, .control, .violated, .other {
    color: white;
}

.starved {
    background-color: black;
}

.control {
    background-color: dark_red;
}

.violated {
    background-color: red;
}

.other {
    background-color: red;
}

.infected, .zero {
    background-color: lime;
    color : dark_green;
}
```

CHAPTER 7

# Graphical User Interface

**TODO**

Uses Tkinter under the hood.

```
$ manalyst.pyw
```

CHAPTER 8

# Data Crunching

## 8.1 Load Experiment

### 8.1.1 Standalone Mode

To interactively inspect and manipulate experiment data in standalone mode:

```
C:\> python -m microanalyst experiment.json
Type "model", "help(model)" for more information.
>>>
```

This opens Python console with preloaded instance of the model to run queries on. Type `help(model)` for details or take a look at the unit tests which go into much greater detail with respect to possible use cases compared to the examples below.

### 8.1.2 Import

Using microanalyst as a module, e.g. in a custom script or from IPython Notebook which is a popular tool among the scientific community:

```
>>> import microanalyst.model
>>> model = microanalyst.model.from_file(r'C:\data\experiment.json')
```

## 8.2 Query Examples

### 8.2.1 Well Names

A static list of 96 well names on an 8x12 microplate in row-major order (returns a generator object):

```
>>> for i, name in enumerate(model.well_names()):
>>>     print '[%d] = %s' % (i, name)
[0] = A1
[1] = A2
[2] = A3
(...)
```

### 8.2.2 Microplate Names

Microplates used in the whole experiment:

```
>>> model.microplate_names()
[u'001', u'002', u'003', (...), u'B001', u'B002']
```

To restrict the lookup domain use one or both of the indices: `spreadsheet`, `iteration`:

```
>>> model.microplate_names(spreadsheet=0)
[u'001', u'002', u'003', (...), u'B001', u'B002']
```

### 8.2.3 Genes/Proteins

Return a sorted list of unique genes:

```
>>> model.genes()
['ART2', 'HMLALPHA2', 'Q0017', 'Q0080', 'Q0085', (...)]
```

To obtain a list of genes actually used in the experiment, i.e. the ones that are mapped to existing microplates call `genes_used()`:

```
>>> a = set(model.genes())
>>> b = set(model.genes_used())
>>> c = a.difference(b)
>>> len(a), len(b), len(c)
(5869, 5590, 279)
```

Genes can be restricted to a given microplate, well or both. The `microplate` argument is the name of a microplate such as "B002" while `well` can be either a 0-based index or a string such as "H12". For example to get all genes on a certain microplate:

```
>>> model.genes(microplate='B002')
['YBL097W', 'YBR004C', 'YBR021W', 'YBR174C', (...)]
```

Getting a single gene rather than an array is done with a `gene_at()` method which expects both arguments:

```
>>> model.gene_at(microplate='B002', well='E8')
'YBL097W'
```

Alternatively to obtain a gene by its name (case insensitive):

```
>>> model.gene('ybl097w')
'YBL097W'
```

Genes shown in previous examples are not just textual names but fully-fledged objects encapsulating useful information:

```
>>> for gene in model.genes():
>>>     info = (gene.name, gene.microplate_name, gene.well_name)
>>>     print 'Gene "%s" is located on microplate "%s" at well "%s".' % info
>>>
Gene "ART2" is located on microplate "025" at well "D2".
Gene "HMLALPHA2" is located on microplate "053" at well "E9".
Gene "Q0017" is located on microplate "001" at well "D6".
(...)
```

**Note:** It is assumed that genes are bijectively mapped to (microplate, well) pairs. If a gene occurs more than once, i.e. duplicates are found on one or more microplates, this may lead to undefined and faulty results. Textual warnings are issued for duplicate instances of genes.

Instances of genes are also callable function objects:

```
>>> for gene in model.genes():
>>>     print gene, gene()
ART2 (u'025', u'D2')
HMLALPHA2 (u'053', u'E9')
Q0017 (u'001', u'D6')
(...)
```

Finally, they can be used to quickly obtain corresponding data samples with a convenience method which is a wrapper over model's `values()`. By default all experiment iterations and spreadsheets are taken into account but this can be restricted with two optional parameters, i.e. `iteration` and `spreadsheet` (both are 0-based indices). Example:

```
>>> for gene in model.genes_used():
>>>     print gene.values().ravel()
[0.6780999898910522, 0.6870999932289124, 0.6870999932289124, (...)]
[0.633899986743927, 0.7077000141143799, 0.679099977016449, (...)]
(...)
```

### 8.2.4 Well Values

Due to large amounts of numerical data in the experiment NumPy is a natural choice for storage and computation. Data samples are kept in a non-jagged 4-dimensional floating-point array where consecutive dimentions are: `iteration` x `spreadsheet` x `microplate` x `well`. To take full advantage of speed improvements over Python's lists each dimension is ensured to contain subarrays of the same size. This is achieved by padding missing spreadsheets if necessary (the remaining dimensions do not matter, e.g. a microplate is always assumed to have 96 wells).

The array can be manipulated directly leveraging NumPy features by accessing `array4d` property, e.g.:

```
>>> model.array4d.shape
(3, 4, 65, 96)
```

However, a pivotal way for slicing the array is through the `values()` method which uses explicitly named arguments (all are optional). Additionally `microplate` and `well` can be either 0-based indices or names. Missing values are indicated with `None`. The rows in this case correspond to iterations, whereas the columns to Excel™ spreadsheets:

```
>>> model.values(microplate='001', well='A1')
array([[ 0.7385    ,  0.66869998,  0.66420001],
       [ 0.74629998,  0.70660001,  0.63870001],
       [ 0.71689999,  0.78380001,  0.72259998]])
```

### 8.2.5 Control Wells

To explicitly check if a particular well is a control well (either names or 0-based indices can be used for both `microplate` and `well` arguments):

```
>>> model.is_control(iteration=0, spreadsheet=0, microplate='008', well='A4')
True
```

There is also a mask for quick retrieval of control wells which can be used to eliminate them from the whole experiment at once. For instance clamping starved samples can be done like that:

```
>>> model.array4d[(model.array4d <= 0.2) & ~model.control_mask.values] = 0.0
```

**Note:** Gaps in data samples may cause discrepancies in the total number of control wells reported. Missing microplates are not accounted for when using `control_mask` or when iterating over array dimensions, e.g.:

```
>>> len(model.array4d[model.control_mask.values])
2056
>>>
>>> max_i, max_s, max_m, max_w = model.array4d.shape
>>> num_control = 0
>>> for i in xrange(max_i):
>>>     for s in xrange(max_s):
>>>         for m in xrange(max_m):
>>>             for w in xrange(max_w):
>>>                 if model.is_control(i, s, m, w):
>>>                     num_control += 1
>>> num_control
2056
```

To obtain an actual number of control wells, i.e. without missing data samples, iterate over raw JSON:

```
>>> from microanalyst.model import welladdr
>>>
>>> num_control = 0
>>> for i, iteration in enumerate(model.json_data['iterations']):
>>>     for s, spreadsheet in enumerate(iteration['spreadsheets']):
>>>         for m in spreadsheet['microplates']:
>>>             for w in welladdr.names():
>>>                 if model.is_control(i, s, m, w):
>>>                     num_control += 1
>>> num_control
2042
```

### 8.2.6 File names

Return a flat list of file names used throughout the experiment:

```
>>> from pprint import pprint
>>> pprint(model.filenames())
[u'/home/microanalyst/experiment/series1/series1_14days.xls',
 u'/home/microanalyst/experiment/series1/series1_28days.xls',
 u'/home/microanalyst/experiment/series1/series1_42days.xls',
 u'/home/microanalyst/experiment/series1/series1_56days.xls',
 u'/home/microanalyst/experiment/series2/series2_14days.xls',
 u'/home/microanalyst/experiment/series2/series2_28days.xls',
 u'/home/microanalyst/experiment/series2/series2_42days.xls',
 u'/home/microanalyst/experiment/series2/series2_56days.xls',
 u'/home/microanalyst/experiment/series3/series3_14days.xls',
 u'/home/microanalyst/experiment/series3/series3_28days.xls',
 u'/home/microanalyst/experiment/series3/series3_42days.xls',
 u'/home/microanalyst/experiment/series3/series3_56days.xls']
```

Restrict to only the second iteration and hide paths:

```
>>> pprint(model.filenames(False, iteration=1))
[u'series2_14days.xls',
 u'series2_28days.xls',
 u'series2_42days.xls',
 u'series2_56days.xls']
```

### 8.2.7 Number of Experiment Iterations

```
>>> model.num_iter
3
```

### 8.2.8 Parsed JSON Data

If the underlying model does not live up to your needs you can retrieve a Python dictionary built from raw JSON and process it in any way you can possibly imagine. This may be handy for accessing ignored metadata such as custom annotations (e.g. introduced with `group.py` script). Example:

```
>>> temperatures = []
>>> for iteration in model.json_data['iterations']:
>>>     for spreadsheet in iteration['spreadsheets']:
>>>         for microplate in spreadsheet['microplates'].values():
>>>             temperatures.append(microplate['temperature'])
>>>
>>> print 'Average temperature was %.1f Celsius' % (sum(temperatures) / float(len(temperatures)))
Average temperature was 27.5 Celsius
```

A deep copy of the original JSON is created to avoid side effects, e.g. when missing spreadsheets are substituted with empty stubs. If that hapens an appropriate warning message is printed to the console.

CHAPTER 9

# Case Study

## 9.1 Overview

Let us identify genes which consistently reduce the lifespan of sampled cells. In the first example we will take a closer look at strictly repeatable patterns, but later that constraint will be relaxed with the use of Hamming distance metric.

## 9.2 Basic Example

First, the experiment needs to be loaded from a JSON file with microanalyst. We will also need NumPy later on, thus the additional import statement.

```
import microanalyst.model
import numpy as np

model = microanalyst.model.from_file(r'C:\data\experiment.json')
```

### 9.2.1 Clustering

In order to facilitate pattern recognition well values must be translated to a simpler form. This can be achieved twofold, either via `quantize.py` script described in previous chapters or by directly manipulating the model, which is more flexible.

The most suitable representation of data samples in this case would be with sequences of binary digits. Designating "ones" for starved wells instead of "zeros" allows for capturing gene's values with a meaningful decimal number directly expressing the mortality of that gene. Due to positional nature of the binary system the earlier the moment of starvation the greater that number will be, which can serve as a simple metric. This is briefly demonstrated by the following table.

| iteration | 1st | | | 2nd | | |
|---|---|---|---|---|---|---|
| days | 14 | 28 | 42 | 14 | 28 | 42 |
| values | 0.6915 | 0.0605 | 0.0603 | 0.6979 | 0.6187 | 0.0653 |
| binary | 0 | 1 | 1 | 0 | 0 | 1 |
| decimal | 3 | | | 1 | | |

Binarization is straightforward given a known threshold (e.g. values below or equal to 0.2):

```
starved = model.array4d <= 0.2
model.array4d[starved]  = 1
model.array4d[~starved] = 0
```

However, this does not account for missing data samples (indicated with `None`), which would be incorrectly regarded as zeros, nor for control wells. To reject them we need to isolate those elements with a mask before doing any value assignment and then mark invalid data samples with a special value after the binarization. The final code for value clustering could be encapsulated in a function similar to this one (note array type casting at the end to allow bitwise operations later):

```
def cluster(model):
    """Assign values: { starved=1, missing/control=2, other=0 }."""

    starved = model.array4d <= 0.2
    special = model.control_mask.values | np.equal(model.array4d, None)

    model.array4d[starved] = 1
    model.array4d[~starved] = 0
    model.array4d[special] = 2

    model.array4d = model.array4d.astype(np.int8, copy=False)
```

After calling `cluster(model)` the only values left should be 0, 1 and 2:

```
>>> set(model.values().ravel())
set([0, 1, 2])
```

### 9.2.2 Conversion

One way of converting a sequence of binary digits, e.g. a tuple or a list of integers, to a decimal number is with bit shifting (from right to left):

```
def decimal(binary_digits):
    """Convert a sequence of binary digits to decimal number."""

    number = 0
    for i, digit in enumerate(reversed(binary_digits)):
        number |= digit << i

    return number
```

Example:

```
>>> decimal([1, 0, 1, 1])
11
```

### 9.2.3 Smoothing

Naturally cells which once died can no longer become alive. For this reason it is expected to observe continuous sequences of zeros followed by continuous sequences of ones but never the other way around. If for some reason that rule does not hold all initial "ones" must be discarded as false positives. Consider this:

| | binary | | | | | | | | decimal |
|---|---|---|---|---|---|---|---|---|---|
| input | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 183 |
| output | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 7 |

A smoothing function:

```
def smooth(number):
    """Discard bits before the last continuous block of ones."""
```

```
    result = 0
    i = 0

    while number & 1:
        result |= 1 << i
        i += 1
        number >>= 1

    return result
```

Example:

```
>>> x = decimal([1, 0, 1, 1, 0, 1, 1, 1])
>>> y = smooth(x)
>>> print 'before: %d = %s, after: %d = %s' % (x, bin(x)[2:], y, bin(y)[2:])
before: 183 = 10110111, after: 7 = 111
```

### 9.2.4 Filtering

Now that we have all the building blocks in place we can proceed to filtering genes by eliminating those which are of no interest. Specifically we want to skip genes containing control wells or with missing data samples or those which caused no cell starvation whatsoever. This can be done by examining values of a particular well measured at different points in time.

If you recall gene's `values()` method returns a complete picture of a particular microplate well (associated with a gene). The result is a 2-D array of samples measured within iterations (rows) and spreadsheets (columns):

```
>>> gene.values()
array([[ 0.722     ,  0.6814    ,  0.70859998],
       [ 0.7245    ,  0.71319997,  0.73180002],
       [ 0.73210001,  0.7324    ,  0.77560002]])
```

Since our model was clustered the domain of `values()` becomes $\{0, 1, 2\}$:

```
>>> gene.values()
array([[0, 0, 0, 0],
       [0, 1, 1, 1],
       [2, 2, 2, 2]], dtype=int8)
```

Those clustered values can be used to evaluate genes:

```
for gene in model.genes_used():

     values = gene.values()

     # missing/control wells are marked with "2"
     if 2 in set(values.ravel()): continue

     # lack of starvation adds up to zero
     if values.sum() == 0: continue
```

The next step is compressing a series of binary digits from each iteration into a smoothed decimal number using helper functions defined earlier:

```
values = [smooth(decimal(x)) for x in values]
```

At this point smoothing might have introduced useless "zeros" again if the original binary sequence comprised of false positives followed by zeros. To mitigate this we need to rewrite our filter so that lack of starvation is detected

afterwards. Additionally we add a condition for ignoring patterns which do not repeat exactly across iterations. Note that `values` becomes a list rather than numpy.ndarray due to the use of list comprehension:

```
for gene in model.genes_used():

    values = gene.values()

    # missing/control wells are marked with "2"
    if 2 in set(values.ravel()): continue

    # express patterns with numbers
    values = [smooth(decimal(x)) for x in values]

    # lack of starvation adds up to zero
    if sum(values) == 0: continue

    # different patterns in iterations
    if len(set(values)) > 1: continue
```

### 9.2.5 Collecting

If a gene makes its way through all the checks then it can be regarded as a legitimate candidate for further evaluation. We may save its name and starvation pattern in a dictionary. As intended the list of values contains identical patterns so we are free to pick any index, e.g. `values[0]`:

```
patterns = {}
for gene in model.genes_used():
    (...)
    patterns[gene] = values[0]
```

Then to obtain protein names ranked by their mortality level (best come first):

```
>>> for i, gene in enumerate(reversed(sorted(patterns, key=patterns.get))):
>>>     print '%d. %s' % (i + 1, gene)
1. LYE328J
2. LTE008P
3. LBY102P
4. LOE101P
5. LQE368J
6. LUY021P
7. LWE127P
8. LTY014J
(...)
```

Remember to take advantage of `xlsv.py` and `xlsh.py` scripts for visualizing experiment data, which can substantially simplify and augment the process of protein analysis.

A distribution of genes causing death after a certain number of days can be calculated using a counter. The percentages are non-cumulative and are only relative to a small subset of genes which have a pattern repeating exactly across all subsequent iterations.:

```
>>> from collections import Counter
>>>
>>> day_offsets = [14, 28, 42]
>>>
>>> counter = Counter(patterns.values())
>>> for pattern in reversed(sorted(counter.keys(), key=counter.get)):
>>>
```

```
>>>     percentage = counter.get(pattern) / float(len(patterns)) * 100.0
>>>     days = day_offsets[len(day_offsets) - len(bin(pattern)[2:])]
>>>
>>>     print '%d%% caused death after %d days' % (percentage, days)
55% caused death after 42 days
33% caused death after 14 days
10% caused death after 28 days
```

### 9.2.6 Complete code

Basic Example:

```
#!/usr/bin/env python

# The MIT License (MIT)
#
# Copyright (c) 2014 Bartosz Zaczynski
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE ANfD NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.

from collections import Counter

import microanalyst.model
import numpy as np


def cluster(model):
    """Assign values: { starved=1, missing/control=2, other=0 }."""

    starved = model.array4d <= 0.2
    special = model.control_mask.values | np.equal(model.array4d, None)

    model.array4d[starved] = 1
    model.array4d[~starved] = 0
    model.array4d[special] = 2

    model.array4d = model.array4d.astype(np.int8, copy=False)


def decimal(binary_digits):
    """Convert a sequence of binary digits to decimal number."""
```

```
    number = 0
    for i, digit in enumerate(reversed(binary_digits)):
        number |= digit << i

    return number


def smooth(number):
    """Discard bits before the last continuous block of ones."""

    result = 0
    i = 0

    while number & 1:
        result |= 1 << i
        i += 1
        number >>= 1

    return result


def show_ranked(patterns):
    """List protein names ranked by their mortality level."""
    for i, gene in enumerate(reversed(sorted(patterns, key=patterns.get))):
        print '%d. %s' % (i + 1, gene)


def show_distribution(patterns, day_offsets):
    """Show distribution of genes' mortality level."""

    counter = Counter(patterns.values())
    for pattern in reversed(sorted(counter.keys(), key=counter.get)):

        percentage = counter.get(pattern) / float(len(patterns)) * 100.0
        days = day_offsets[len(day_offsets) - len(bin(pattern)[2:])]

        print '%d%% caused death after %d days' % (percentage, days)


def main(filename):

    model = microanalyst.model.from_file(filename)

    cluster(model)

    patterns = {}
    for gene in model.genes_used():

        values = gene.values()

        # missing/control wells are marked with "2"
        if 2 in set(values.ravel()): continue

        # express patterns with numbers
        values = [smooth(decimal(x)) for x in values]

        # lack of starvation adds up to zero
        if sum(values) == 0: continue
```

```
        # different patterns in iterations
        if len(set(values)) > 1: continue

        patterns[gene] = values[0]

    day_offsets = [14*(i+1) for i in xrange(model.array4d.shape[1])]

    show_ranked(patterns)
    show_distribution(patterns, day_offsets)


if __name__ == '__main__':
    main(r'C:\data\experiment.json')
```

## 9.3 Advanced Example

In this advanced example we will remove the constraint on strictly repeating patterns and replace it with a more sophisticated metric comprised of:

- the moment of starvation
- repeatability of patterns
- similarity of patterns.

### 9.3.1 Mortality

Mortality level will be calculated as before, i.e. by comparing numbers which reflect gene's binary patterns. A greater number indicates stronger effect on survival of the cells. Since the numbers from subsequent iterations might be different at this time the actual mortality is the sum of respective decimal numbers:

```
mortality = sum(values)
```

### 9.3.2 Repeatability

A repeatable pattern is the one which contains at least one starvation and appears unchanged most frequently. Note there be might cases where a pattern is comprised of zeros in one iteration but not in the others. Therefore, we want to count occurrences of non-zero patterns only and determine the most frequent one as long as it occurs more than once. By definition a pattern which has a count of one is not repeatable.

Example:

| pattern | | | occurrences | repeatability |
|---|---|---|---|---|
| 0 | 0 | 1 | { "1": 1 } | 0 |
| 1 | 0 | 3 | { "1": 1, "3": 1 } | 0 |
| 1 | 7 | 3 | { "1": 1, "3": 1, "7": 1 } | 0 |
| 0 | 3 | 3 | { "3": 2 } | 2 |
| 1 | 3 | 3 | { "1": 1, "3": 2 } | 2 |
| 1 | 1 | 1 | { "1": 3 } | 3 |

The code:

```
from collections import Counter

counter = Counter([x for x in values if x != 0])
max_count = max(counter.values())

repeatability = max_count if max_count > 1 else 0
```

### 9.3.3 Similarity

Similarity is the measure of differences between the patterns observed in consecutive iterations. An essential part of this measure will be the Hamming distance which counts the number of positions at which two equally long binary strings manifest different digits:

```
def hamming_distance(a, b):
    """Calculate the Hamming distance between two numbers."""

    result, c = 0, a ^ b
    while c:
        result += 1
        c &= c - 1

    return result
```

This needs to be extrapolated over all combinations of unordered pairs of patterns:

```
import itertools

def hamming_pairs(values):
    """Return the sum of Hamming distances between all pair combinations."""

    distance = 0
    for pair in itertools.combinations(values, 2):
        distance += hamming_distance(*pair)

    return distance
```

Unlike the factors defined earlier Hamming distance is inversely proportional to the measured quality, i.e. a short distance is considered to be better than a long one. In order to make it compatible with the remaining components of the metric it needs to be reversed. The actual distance must be subtracted from the maximum theoretical one.

$$H_{max} = k \cdot C_n^2 = k \cdot \binom{n}{2} = k \cdot \frac{n!}{2! \cdot (n-2)!} = \frac{k \cdot n \cdot (n-1)}{2}$$

$k -$ number of spreadsheets per iteration

$n -$ number of iterations

However, this formula as well as the Hamming distance itself cannot be applied directly on an unprocessed sequence of patterns. Just like with repeatability we need to exclude zeros from the computation to avoid comparing meaningless patterns such as (0, 0) and reject non-repeating patterns. Therefore:

```
nonzero = [x for x in values if x != 0]

n = len(nonzero)
k = model.array4d.shape[1]

hmax = k * n * (n - 1) / 2
h = hamming_pairs(nonzero)
```

```
similarity = hmax - h if n > 1 else 0
```

### 9.3.4 Final Metric

All components can be encapsulated in a class utilizing delegation for the computation of specific metrics:

```
from collection import namedtuple

class Metric(object):

   def __init__(self, model, values):

       n, k = model.array4d.shape[:2]

       Component = namedtuple('Component', 'value, max')

       self.components = [
           Component(mortality(values), n*(2**k-1)),
           Component(repeatability(values), n),
           Component(similarity(model, values), k*n*(n-1)/2)
       ]

       self.percents = [x.value / float(x.max) * 100.0 for x in self.components]
       self.total = sum([x.value for x in self.components])

   def __str__(self):
       return '(m=%d%%, r=%d%%, s=%d%%)' % tuple(self.percents)
```

Example:

```
>>> Metric(model, [3, 3, 1])
(m=15%, r=66%, s=83%)
```

- Mortality is measured at 15% because the sum of all patterns amounts to 7, whereas the maximum is 3 x 15 for this particular model (there were four spreadsheets per iteration).
- Repeatability scores 66% because a pattern appears twice during three iterations.
- Similarity:

$$S = \frac{H_{max} - (h(3,3) + h(3,1) + h(3,1))}{H_{max}} = \frac{12 - (0 + 1 + 1)}{12} = \frac{10}{12} \approx 85\%$$

where

$$H_{max} = \frac{k \cdot n \cdot (n-1)}{2} = \frac{4 \cdot 3 \cdot 2}{2} = 12$$

Of course the components of this metric are correlated. For instance a pattern repeating exactly, i.e. having r=100%, implies similarity of s=100% and vice versa. However, some correlations do not work both ways such as mortality and repeatability (m=100% then r=100%, but not the other way around).

### 9.3.5 Scores

Genes can be ranked by their total score:

```
>>> it = sorted(metrics.iteritems(), key=lambda x: x[1].total)
>>> for i, (gene, metric) in enumerate(reversed(it)):
>>>     print '%d. %s = %s' % (i + 1, gene, metric)
1. LVE011P = (m=100%, r=100%, s=100%)
2. LVE013P = (m=100%, r=100%, s=100%)
3. LUE026J = (m=82%, r=66%, s=83%)
4. LYE299J = (m=64%, r=66%, s=83%)
5. LUY031P = (m=68%, r=66%, s=50%)
6. LZE276J = (m=46%, r=100%, s=100%)
7. LZE179J = (m=66%, r=66%, s=33%)
8. LTE162J = (m=46%, r=100%, s=100%)
9. LPE088J = (m=46%, r=100%, s=100%)
(...)
```

If some quality needs to be emphasized use a weighted average. For example to favour repeatable patterns (`average` is defined in the next section):

```
>>> weights = [1, 2, 1]
>>> it = sorted(metrics.iteritems(), key=lambda x: x[1].average(weights))
>>> for i, (gene, metric) in enumerate(reversed(it)):
>>>     print '%d. %s = %s' % (i + 1, gene, metric)
1. LVE011P = (m=100%, r=100%, s=100%)
2. LVE013P = (m=100%, r=100%, s=100%)
3. LZE276J = (m=46%, r=100%, s=100%)
4. LUY021P = (m=46%, r=100%, s=100%)
5. LTE162J = (m=46%, r=100%, s=100%)
6. LPE088J = (m=46%, r=100%, s=100%)
7. LBE223J = (m=46%, r=100%, s=100%)
8. LRE123J = (m=46%, r=100%, s=100%)
9. LNY032P = (m=46%, r=100%, s=100%)
(...)
```

### 9.3.6 Complete code

Advanced Example:

```
#!/usr/bin/env python

# The MIT License (MIT)
#
# Copyright (c) 2014 Bartosz Zaczynski
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE ANfD NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
```

```
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.

from collections import Counter, namedtuple

import microanalyst.model
import numpy as np
import itertools


def cluster(model):
    """Assign values: { starved=1, missing/control=2, other=0 }."""

    starved = model.array4d <= 0.2
    special = model.control_mask.values | np.equal(model.array4d, None)

    model.array4d[starved] = 1
    model.array4d[~starved] = 0
    model.array4d[special] = 2

    model.array4d = model.array4d.astype(np.int8, copy=False)


def decimal(binary_digits):
    """Convert a sequence of binary digits to decimal number."""

    number = 0
    for i, digit in enumerate(reversed(binary_digits)):
        number |= digit << i

    return number


def smooth(number):
    """Discard bits before the last continuous block of ones."""

    result = 0
    i = 0

    while number & 1:
        result |= 1 << i
        i += 1
        number >>= 1

    return result


def hamming_distance(a, b):
    """Calculate the Hamming distance between two numbers."""

    result, c = 0, a ^ b
    while c:
        result += 1
        c &= c - 1

    return result
```

```
def hamming_pairs(values):
    """Return the sum of Hamming distances between all pair combinations."""

    distance = 0
    for pair in itertools.combinations(values, 2):
        distance += hamming_distance(*pair)

    return distance


def mortality(values):
    return sum(values)


def repeatability(values):
    counter = Counter([x for x in values if x != 0])
    max_count = max(counter.values())
    return max_count if max_count > 1 else 0


def similarity(model, values):

    nonzero = [x for x in values if x != 0]

    n = len(nonzero)
    k = model.array4d.shape[1]

    hmax = k * n * (n - 1) / 2
    h = hamming_pairs(nonzero)

    return hmax - h if n > 1 else 0


class Metric(object):

    def __init__(self, model, values):

        n, k = model.array4d.shape[:2]

        Component = namedtuple('Component', 'value, max')

        self.components = [
            Component(mortality(values), n*(2**k-1)),
            Component(repeatability(values), n),
            Component(similarity(model, values), k*n*(n-1)/2)
        ]

        self.percents = [x.value / float(x.max) * 100.0 for x in self.components]
        self.total = sum([x.value for x in self.components])

    def __str__(self):
        return '(m=%d%%, r=%d%%, s=%d%%)' % tuple(self.percents)

    def average(self, weights=None):

        if not weights:
            weights = [1] * 3
```

```
        return sum([weights[i] * self.percents[i] for i in xrange(3)]) / float(sum(weights))


def show_ranked_total(metrics):
    _show_ranked(metrics, lambda x: x[1].total)


def show_ranked_average(metrics, weights=None):
    _show_ranked(metrics, lambda x: x[1].average(weights))


def _show_ranked(metrics, key_func):
    it = sorted(metrics.iteritems(), key=key_func)
    for i, (gene, metric) in enumerate(reversed(it)):
        print '%d. %s = %s' % (i + 1, gene, metric)


def main(filename):

    model = microanalyst.model.from_file(filename)

    cluster(model)

    metrics = {}
    for gene in model.genes_used():

        values = gene.values()

        # missing/control wells are marked with "2"
        if 2 in set(values.ravel()): continue

        # express patterns with numbers
        values = [smooth(decimal(x)) for x in values]

        # lack of starvation adds up to zero
        if sum(values) == 0: continue

        metrics[gene] = Metric(model, values)

    #show_ranked_total(metrics)
    show_ranked_average(metrics, [1, 2, 1])


if __name__ == '__main__':
    main(r'C:\data\experiment.json')
```

CHAPTER 10

# Modules Overview

Individual modules that come with this project are not exposed directly. However, there are a few worth being discussed here due to their agnostic qualities allowing for an application outside of the scope of microanalyst as well as to better understand and utilize them.

## 10.1 expression.py

Defines a flexible wrapper around Python's lambda function which can be evaluated dynamically:

```
>>> from microanalyst.model.expression import Expression
>>> is_even = Expression('x % 2 == 0')
>>> print is_even
x % 2 == 0
>>> print is_even(2)
True
```

The optional `scope` allows to evaluate custom variables:

```
>>> scope = {'is_even': is_even}
>>> is_odd = Expression('not is_even(x)', scope)
>>> print is_odd(3)
True
```

Use `locals()` to refer to variables in the current scope:

```
>>> is_even = Expression('x % 2 == 0')
>>> is_odd = Expression('not is_even(x)', locals())
```

Use `globals()` and forward declarations for recursive calls:

```
>>> expr = '1 if n < 2 else fib(n - 2) + fib(n - 1)'
>>> global fib
>>> fib = Expression(expr, globals(), ('n',))
>>> print fib(11)
144
```

Enforce parameter order with explicit `params`:

```
>>> pp = Expression('(x/z, y/z)', params=('x', 'y', 'z'))
>>> x, y = pp(10, 25, 5)
```

## 10.2 stylesheet.py

Minimalist syntax for XF records (Microsoft® Excel™ cell formatting) inspired by Cascading Style Sheets (CSS).

### 10.2.1 Supported properties

- background-color
- border-color
- color
- font-weight
- text-align
- number-format (non-standard)

### 10.2.2 Supported selectors

- wildcard `*`
- class selector, e.g. `.header`
- combined selectors, e.g. `.header, .footer`

Styles can be commented out with multi-line comments enclosed between `/*` and `*/` which can be nested.

XF records are cached due to the upper limit of 4k in a single xls file.

Sample usage:

```
>>> from microanalyst.xls import stylesheet
>>> style = stylesheet.load('colorful.css')
>>> for i in range(5):
>>>     xf = style('.header.big', alt=i % 2)
```

## 10.3 tkwidgets.py

User friendly wrappers over new-style and legacy Tkinter widgets.

**TODO**

## 10.4 uniutils.py

Miscellaneous utilities for Unicode handling which strive to maintain portability across the three major platforms, i.e. Windows, Linux and Mac OS. Despite the 21st century character encoding still remains an issue justifying the need for such a module.

File system, shell and stdin/stdout/stderr may all have different character encodings even on a single platform. Furthermore, these encodings vary depending on the language version of the operating system.

Unfortunatley, the sole character encoding is not sufficient to retain portability either. For example both Linux and Mac OS can use the same UTF-8 encoding for file system but in different normal forms (NFC vs. NFD), which has to be taken into account and addressed accordingly.

CHAPTER 11

# Troubleshooting

1. When invoking a script on Linux there is a "`:No such file or directory`" error.

**Tip:** This indicates that some Python source files contain non-Unix newline characters such as `\r\n` (Windows) or `\r` (Mac OS) instead of simply `\n`. To identify the culprits use the included `check_newline.py` script and correct the issue with a text editor.

2. Error "`IOError: [Errno 9] Bad file descriptor`" or similar.

**Tip:** Are you using a shell wrapper on Windows? Use the standard `cmd.exe`.

3. Error "`The process tried to write to a nonexistent pipe`".

**Tip:** This is a bug on some versions of Windows. You can safely ignore it.

CHAPTER 12

# License

This project is licensed under:

CHAPTER 13

# Indices and tables

- genindex
- modindex
- search