Tick-Tock

TIME & DOCUMENTATION

Moritz Nipshagen mnipshagen@uos.de
Antonia Hain ahain@uos.de

Structure

- Week 1: Introduction
- Week 2: Syntax, Variables & Functions
- Week 3: Control Structures
- Week 4: Lists & Collections
- Week 5: RegEx & Strings
- Week 6: Sorting & I/O
- Week 7:Debugging, Errors & Strategies
- Week 8: 4P: Packages, Practices and Patterns
- Week 9: Object Oriented Programming
- **Week 10: Time, Space & Documentation**
- Week 11: Numpy & Matplotlib
- Week 12&13: Neural Nets & Psychopy
- Week 14: Honorable Mentions & Wrap Up

Last Week's Homework

Updates Soon

- We are not far enough into corrections that we can draw conclusions on common mistakes
- We will update you via stud.ip once we have

Time

Dates

- How do you write down a date? For a journal? A presentation?
- There are many ways to write down a date
  - Wednesday June 13, 2018
  - 13 June 2018
  - 2018-06-13
  - 06/13/2018
  - 6/13/18
  - 2018-06-13T14:18:37+02:00
  - 1528892317
  - 2018164
- Can you read all of those?

Date Ambiguity

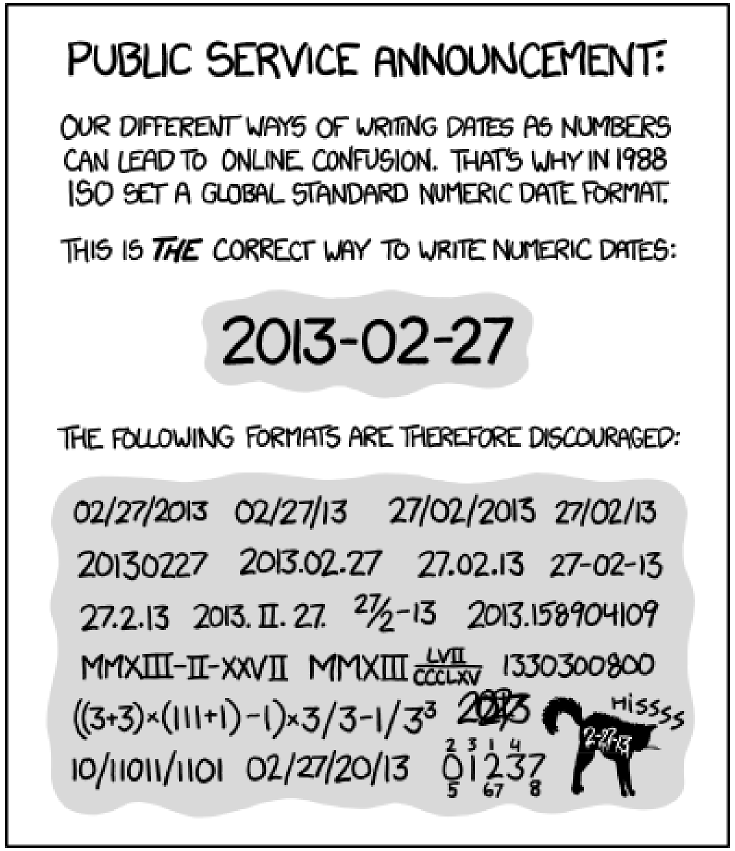
- 08-07-06
  - 08th of July? 06th of July? 07th of August? 07th of June?
- Endianness defines the order
  - Little Endian -> Day – Month – Year
    - Germany primarily uses this: e.g. 13 Juni 2018
  - Middle Endian -> Month – Day – Year
    - US primarily uses this: e.g. 06/13/2018
  - Big Endian -> Year – Month – Day
    - Japan primarily uses this: e.g. 2018-06-13

Date Ambiguity

- To solve this problem there exist many formats and standards
- The important ones:
  - ISO 8601
  - UNIX Timestamp
  - RFC 3339
  - RFC 5322
- Today, we will focus on ISO 8601 & the UNIX timestamp

ISO 8601

- Iso uses the big endian for dates
- It generally sorts the datetime from biggest to smallest
- Numbers have a fixed length and are padded by leading 0s
- The general format is YYYY-MM-DD hh:mm:ss.sss
  - 4 digit year (Y2k was scary)
  - 2 digit month and day
  - 2 digit hour and minute in the 24-hour system
  - 2 digit seconds followed by a dot for fraction of a second
  - Date is separated by a dash, time with colons
    - Both can be omitted in the basic format


PUBLIC SERVICE ANNOUNCEMENT:
OUR DIFFERENT WAYS OF WRITING DATES AS NUMBERS CAN LEAD TO ONLINE CONFUSION. THAT'S WHY IN 1988 ISO SET A GLOBAL STANDARD NUMERIC DATE FORMAT.
THIS IS THE CORRECT WAY TO WRITE NUMERIC DATES:
2013-02-27
THE FOLLOWING FORMATS ARE THEREFORE DISCOURAGED:
02/27/2013 02/27/13 27/02/2013 27/02/13
20130227 2013.02.27 27.02.13 27-02-13
27.2.13 2013. II. 27. 27⁄2-13 2013.158904109
MMXIII-II-XXVII MMXIII LVII/CCCLXV 1330300800
((3+3)×(111+1)-1)×3/3-1/3³ 2013 HISSSS
2-27-13
10/11011/1101 02/27/20/13 01237

ISO 8601 was published on 06/05/88 and most recently amended on 12/01/04. (Munroe 2013)

Why Use Dates?

- So far we had no need to use dates
- But there are quite a few applications:
  - Calendars
  - Transaction management
  - Timeseries data
  - Events
  - Identification
  - Logging

Python Dates

- The **datetime** module offers a lot of functionality to deal with date & time
- It offers several classes, including
  - `date`
    - Can represent a date up to the day
    - It does not include time information
  - `datetime`
    - Can represent a date *including time*, up to microseconds
- You can see that the first three entries are shared, after that datetime includes the time information that date is missing
- Docs: https://docs.python.org/3.5/library/datetime.html#module-datetime

```python
import datetime

today = datetime.date.today()
print(today)
print(repr(today))

now = datetime.datetime.now()
print(now)
print(repr(now))
```

Output:

```
2018-06-04
datetime.date(2018, 6, 4)
2018-06-04 11:13:53.058980
datetime.datetime(2018, 6, 4, 11, 13, 53, 58980)
```

Have A Date

- You can easily create new `date` instances
  - E.g. for appointments, schedules, etc.
- There are more constructors
  - Like `today()` from earlier
  - `fromtimestamp()`
  - etc.

```python
from datetime import date


# class datetime.date(year, month, day)
bday = date(1993, 11, 24)
print(bday)
```

Output:

```
1993-11-24
```

More Dates

- The same day. Two different days of the week.
- `weekday()` is “programmer friendly”
    - It starts with Monday as 0
- `isoweekday()` is the normed way
    - It starts with Monday as 1
- So this was a Wednesday

```
from datetime import date


bday = date(1993, 11, 24)
print(bday.weekday())
print(bday.isoweekday())
```

Output:

```
2
3
```

Formatting Dates

- You often want to output dates in a very specific format, or maybe only need to display parts of it
- **`strftime`** exists for this purpose
  - it stands for **st**ring-**f**ormat-**time**
- We can match the formatting operators to the output
  - %a Wed, abbreviated day in locale
  - %d Day of the month in digits
  - %b abbreviated month in locale
  - %Y the four digit year

```python
from datetime import datetime


now = datetime.now()
print(now)
print(now.strftime('%a, %d. %b %Y'))
print(now.strftime('%c'))
print(now.strftime('%Z %X %f %j’))
```

Output:

```
2018-06-13 14:27:34.053416
Wed, 13. Jun 2018
Wed Jun 13 14:27:34 2018
 14:27:34 053416 164
```

Formatting Dates

- The last two lines might seem a little weird
  - %c full date according to locale
  - %Z the time zone. None is present here
  - %X the date’s time
  - %f the date’s milliseconds
  - %j the current day of the year
- For all the formatting rules see the docs [docs.python.org/3.6/library/datetime.html#strftime-and-strptime-behavior]

```python
from datetime import datetime


now = datetime.now()
print(now)
print(now.strftime('%a, %d. %b %Y'))
print(now.strftime('%c'))
print(now.strftime('%Z %X %f %j’))
```

Output:

```
2018-06-13 14:27:34.053416
Wed, 13. Jun 2018
Wed Jun 13 14:27:34 2018
 14:27:34 053416 164
```

Formatting Dates

Format	Meaning	Example
%Y	4 digit year	1993, 2018
%y	2 digit year	93, 18
%m	2 digit month	11, 06
%b	Abbreviated month	Nov, Mar
%B	Month	November, March (you might see März)
%H	Hours (24h)	06, 11, 23
%M	Minutes	00, 12, 47
%S	Seconds	04, 43, 59
%a	Abbreviated weekday	Tue, Wed, Thu
%c	Locale default	Mon Jun 4 14:59:23 2018

- The formatting rules follow the standards of the programming language C
- This list is not exhaustive, it is just some of the more important ones
- *Locale* can be seen as the language and location settings of your pc

Locale: A Formatting Example

- The output of `strftime` of %c depends on your locale
- We can use the `strftime` to emulate that output
- If you try that for yourself, you might get a different output

```
from datetime import datetime


now = datetime.now()
print(now.strftime('%c'))
print(now.strftime('%a %b %d %H:%M:%S %Y’))


Output:
Wed Jun 13 14:56:13 2018
Wed Jun 13 14:56:13 2018
```

Changing Locale

- We can use the locale module to change our current locale (as far as Python is concerned)
- This also changes the output of %c
- This can be practical sometimes
    - E.g. letting a user choose the format of the output they are getting
- Depending on your system the string in `setlocale` can be different
    - For windows you can find a list here: docs.microsoft.com/en-us/cpp/c-runtime-library/language-strings
    - For linux (and probably mac) you can use locale -a for a list of available locales

```
from datetime import datetime
import locale


# locale.setlocale(category, locale=None)
locale.setlocale(locale.LC_ALL, 'de-DE')
now = datetime.now()
print(now.strftime('%c'))
print(now.strftime('%a %b %d %H:%M:%S %Y’))


Output:
13.06.2018 14:57:49
Mi Jun 13 14:57:49 2018
```

ISO Time

- ISO time looks like this:
  2018–06–13T15:07:34
- It is great for making internationally compatible output
- How can we create a format like this?
- datetime offers a function which does it format you
  - Yay for no typos!

```
from datetime import datetime


someday = datetime(2016,11,28,18,29,37)
print(someday.strftime('%Y-%m-%dT%H:%M:%S'))
print(someday.isoformat())


Output:
2016-11-28T18:29:37
2016-11-28T18:29:37
```

Date Parsing

- In contrast to strftime, there is **`strptime`**
  - Which stands for **str**ing **p**arse **time**
- You give it a *date string*, and in which *format* it is, and strptime creates a datetime object
- This can be used to e.g. parse user inputted information from a form

```
from datetime import datetime


parsed = datetime.strptime(
    'Wed Jun 13 14:47:12 2018’,
    '%a %b %d %H:%M:%S %Y’
)

print(parsed.isoformat())
```

Output:

```
2018-06-13T14:47:12
```

Date Calculations

- Often we need to know how much time is between two dates
  - How many minutes/hours until the alarm rings
  - How many days are between 2018-02-28 and 2018-03-01
  - How many weeks are between 2018-04-03 and 2018-07-07?
    - E.g. how many lectures do we have
- Python allows to calculate with dates only if they have a time component
  - Meaning we cannot calculate with date
  - But with datetime (but also not with time)

```python
from datetime import datetime


a = datetime(2018, 6, 13, 14, 35)
b = datetime(2018, 6, 13, 17, 22)

print(b - a)
```

Output:
2:47:00

Date Calculations: Example

- Detecting a leap year
- Just giving a date defaults to 00:00 time
- Using the days property we can access how many days there are in-between
  - Other attributes are seconds and microseconds
  - The function total_seconds() adds up the days, seconds and microseconds into a single number

```
from datetime import datetime


a = datetime(2000, 2, 28, 23, 59)
b = datetime(2000, 3, 1)
c = datetime(2100, 2, 28, 23, 59)
d = datetime(2100, 3, 1)

print((b - a).days) # leap year
print((d - c).days) # no leap year
```

Output:

```
1
0
```

Timedelta

- Calculating with datetimes results in a new class: **timedelta**
- Timedelta represents a time *difference*
  - At most times, when you see a delta (**Δ**, **δ**), it expresses a form of difference
- It has the properties
  - days
  - seconds
  - microseconds

```python
from datetime import datetime


a = datetime(2000, 2, 28, 23, 59)
b = datetime(2000, 3, 1)
print(type((b - a)))
```

Output:

```
<class 'datetime.timedelta'>
```

Timedelta

- But what if we know the difference and need to calculate the date?
- You can create timedeltas with the parameters
  - Weeks
  - Hours
  - Minutes
  - Seconds
  - Milliseconds
  - Microseconds
- They will all get converted into days, seconds, microseconds

```python
from datetime import datetime, timedelta


now = datetime.now()
days137 = timedelta(days=137)

print(now + days137)
```

Output:

```
2018-10-28 15:12:37
```

Timedelta

- You can even divide by the time to get a certain time difference
  - E.g. how many weeks are in a semester (so how many lectures do we have)
- We ceil the result to get full weeks instead of a fraction

```
import math
from datetime import datetime, timedelta


begin = datetime(2018, 4, 3)
end = datetime(2018, 7, 7)
print(math.ceil((end - begin) /
    timedelta(weeks=1)))
print(math.ceil((end - begin) /
    timedelta(weeks=2)))


Output:
14
7
```

Human Time Calculations

- As humans though, we often use contextual time differences
  - Tomorrow, 5 minutes ago, Saturday
- Those are harder to parse
- There exists a neat library called parsedatetime for this purpose
- time_struct holds the changed time after parsing
- parse_status holds whether the parsing was successful
  - 1 for success, 0 for no success

```python
import parsedatetime as pdt


cal = pdt.Calendar()
time_struct, parse_status =
    cal.parse("tomorrow")
print(time_struct)
print(parse_status)
```

Output:

```
time.struct_time(tm_year=2018, tm_mon=6,
tm_mday=7, tm_hour=9, tm_min=0, tm_sec=0,
tm_wday=3, tm_yday=158, tm_isdst=-1)
1
```

Human Time Calculations

- As humans though, we often use contextual time differences
  - Tomorrow, 5 minutes ago, Saturday
- Those are harder to parse
- There exists a neat library called parsedatetime for this purpose
- time_struct holds the changed time after parsing
- parse_status holds whether the parsing was successful
  - 1 for success, 0 for no success
- If there was no success, time_struct is now

```
import parsedatetime as pdt


cal = pdt.Calendar()
time_struct, parse_status =
    cal.parse(“hello")
print(time_struct)
print(parse_status)


Output:
time.struct_time(tm_year=2018, tm_mon=6,
tm_mday=6, tm_hour=7, tm_min=57, tm_sec=28,
tm_wday=2, tm_yday=157, tm_isdst=1)
0
```

Complexity

Complexity

- We call the *resources* our program needs its **complexity**
  - This means mainly *time* and *space (aka memory)*
- It is a measure of the efficiency of the program
- We can compare algorithms that perform the same task by their complexity
- Most often we are concerned with *time complexity*
- This does not necessarily mean the seconds an algorithm needs to complete, but rather how many elementary operations need to be performed
  - The actual time taken is very dependent on the system it is executed on, but can still be a significant measure to compare algorithms when used on the same machine

Complexity

- It is often impossible to count how many operations an algorithm performs
- Since complexity generally grows with the input size, it is often expressed as a *function* of input size *n*
  - Since complexity may vary greatly for different input types, we often need to express complexity with many different functions
- Two relevant measures then are the **worst-case complexity** and the **average-case complexity**
  - Where the worst-case is the highest complexity for all input types and the average-case is the average complexity over all input types
- The exact numbers for this are near impossible to determine
  - And again vary from machine to machine

Landau (Big O) Notation

- Furthermore, complexity in general is rather low for small input sizes
- As such we mostly consider complexity for very large *n,* meaning input sizes that tend towards infinity, and the limits that complexity reaches there
    - Instead of giving an exact function like $n^2 + 3n - 1$ we giver upper bounds like $c * n^2$
        - Where c is some constant to multiply n^2 with
    - Those limits are called the **asymptotic behaviour** of complexity
- Since this terminology is lend from mathematics, we continue to use math stuff
- The big-O (or Landau) notation is used for asymptotic behaviour: $O(complexity\_function)$
    - Note that this usually represents the worst case, for the average case we use $\Omega(function)$

Landau Notation: Example

- The usual algorithm for integer multiplication has a complexity of $O(n^2)$
  - Where both numbers have at most n digits
- This means that the operations necessary to multiply two integers are less or equal to some constant c multiplied by the function n^2
- So the actual amount of operations – let's call it $f(n)$ – is always smaller than $n^2 * c$, for some fixed c and from a n_0 onwards
  - Since the operations will always be lower, $c * n^2$ is an upper bound
- Don't be too intimidated by this. We just wanted you to have seen this once.
- For the most time, you will probably not be concerned with complexity analysis

Measuring Complexity

The time Module

- Time is – in a way – a more basic module than datetime, as it does not offer the many conversions of time available in datetime
  - Though there are some overlaps
  - Especially function names. A good idea: do not use from time/datetime import *
- The probably most relevant functions of this modules are time() and sleep(s)
  - time() returns the seconds since the *epoch*[1]
    - Also called the UNIX timestamp
    - The precision is dependent on the system, but it is always precise to the second
  - sleep(s) pauses the execution for s seconds

```python
import time


print(time.time())
time.sleep(2)
print("I was delayed by 2 seconds!")


Output:
1528273434.4472227
I was delayed by 2 seconds!
```

[1] Nearly. There are “leap seconds”, which throws this off a little: see here for an entertaining explanation https://youtu.be/-5wpm-gesOY

The UNIX Timestamp

- The UNIX timestamp is the time that has passed since 1970-01-01T00:00:00
    - This is called the *epoch*
- It is an arbitrary point in time that is used as 0 for many computer purposes
- The accuracy of this timestamp depends on the hardware and operating system
- It is used throughout the world's computing system and builds the base for many processes
    - Stock markets, server protocols, network connections, etc.

Measuring Time

- We can use time() to measure the time it took for a function to complete
- This can be used for example to
  - Compare functions
  - Log time (taken)
  - Make nice loading bars
  - How long the program has been running
- I get different results though, if I execute it several times
- This might lead to inaccuracies when comparing times

```
import random
import time


size = 10000
seq = random.sample(range(10000000000), k=size)
start = time.time()
seq.sort()
end = time.time()
print(f"It took {end - start: .3f} seconds to sort {size} numbers.")
```

Output:

```
It took  0.004 seconds to sort 10000 numbers.
```

Measuring Time

- Better: run a function many times and take some statistical value
  - The mean, the maximum, only the top quintile, etc.
- This loop iterates 100 times and stores the results, and in the end can calculate different values from the collected results
  - In this example only the mean, for space reasons
- We could even abstract this into a check_performance() function to test other functions

```
import random
import time


size = 10000
trials = 100
results = []
for i in range(trials):
    seq = random.sample(range(10000000000), k=size)
    start = time.time()
    seq.sort()
    end = time.time()
    results.append(end-start)

avg = sum(results) / trials
print(f"Average of: {avg:.3f}s over {trials} trials.")


Output:
Average of: 0.005s over 100 trials.
```

timeit

- And of course this already exists.
- The timeit module is there to measure performance of functions
- It runs a function multiple times, and returns the total time taken in seconds
- I reduced the size immensely, since it defaults to 1’000’000 runs, and that would take a while

```
import timeit


size = 100
seq = random.sample(range(10000000000), k=size)
print(timeit.timeit(seq.sort))


Output:
1.9772904142155385
```

timeit

- You can run timeit from the commandline as well, and this is actually the preferred method
- Using the cli, timeit actually automatically determines a reasonable amount of repetitions for a function
- It then returns the average of the best 3 runs
- I used a fairly simple example here, but you can test your own functions as well

```
> python -m timeit "123 + 456”


Output:
100000000 loops, best of 3:
0.011 usec per loop
```

timeit

- To test your own functions, a little setup is required
  - Inside python you can use the setup parameter
- In the cli use the -s switch to include some setup
- That will be executed once before executing the actual test

code.py:

```
def func():
    "~".join(str(n) for n in range(100))
```

cli:

```
> python -m testit -s “import code”
‘code.func’
```

Output:

```
100000000 loops, best of 3:
0.00685 usec per loop
```

Calendar Module

- Just a short addition
- Python also offers the calendar module, which uses the time module
- It implements functionality to output a calendar either in pure text or html
    - It also offers functions that can be further used
        - E.g. to get a month in the form of a list of lists, showing the days of the month and which day of the week they are
- It is similar to the UNIX cal program

```
import calendar


calendar.prmonth(2018, 6)
print("\n", calendar.monthcalendar(2018, 6))


Output:
     Juni 2018
Mo Di Mi Do Fr Sa So
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30

[[0, 0, 0, 0, 1, 2, 3], [4, 5, 6, 7, 8, 9, 10],
[11, 12, 13, 14, 15, 16, 17], [18, 19, 20, 21,
22, 23, 24], [25, 26, 27, 28, 29, 30, 0]]
```

Documentation 2.0

Documentation So Far

- We have used docstrings to inform about our code
- They handled parameters, general information and usage instructions
- They did not however handle introductory words to and the structure and general information of a project
- Neither are they accessible outside of the package
  - Meaning you need to install and use a package before you can get information on it
- That is suboptimal

Prelude: Structuring a Project

- If you are working on a bigger project that needs this kind of documentation, it is a good idea to properly structure it
- On the right the general base for such a structure is given
- Project Folder should have the name of your project
- The docs folder holds all the documentation
  - More on that in a second
- The src folder holds all your code pieces

```
Project Folder
|-- docs
|-- src
  |-- modules and packages
```

Introducing: Sphinx

- Sphinx is the documentation software that was developed for the Python docs
- You can install Sphinx with conda or pip
  - conda install sphinx
  - pip install sphinx
- From the cli change into the docs folder and run sphinx-quickstart to setup the directory
  - This will set up the important files, and let you choose certain configurations
  - Inside the conf.py file is all the information about your project

```
Project Folder
|-- docs
  |-- conf.py
  |-- index.rst
  |-- _build
  |-- _templates
  |-- _static
  |-- Makefile
  |-- make.bat
|-- src
  |-- modules and packages
```

Introducing: Sphinx

- You have now the basic information for your project set up!
- You can run make html from the cli in the docs folder to create your documentation
- If you want to have a look at it
  - Change into docs/_build/html
  - Type python -m http.server 8080
  - Open your browser at http://localhost:8080

```
Project Folder
|-- docs
  |-- conf.py
  |-- index.rst
  |-- _build
  |-- _templates
  |-- _static
  |-- Makefile
  |-- make.bat
|-- src
  |-- modules and packages
```

The Output

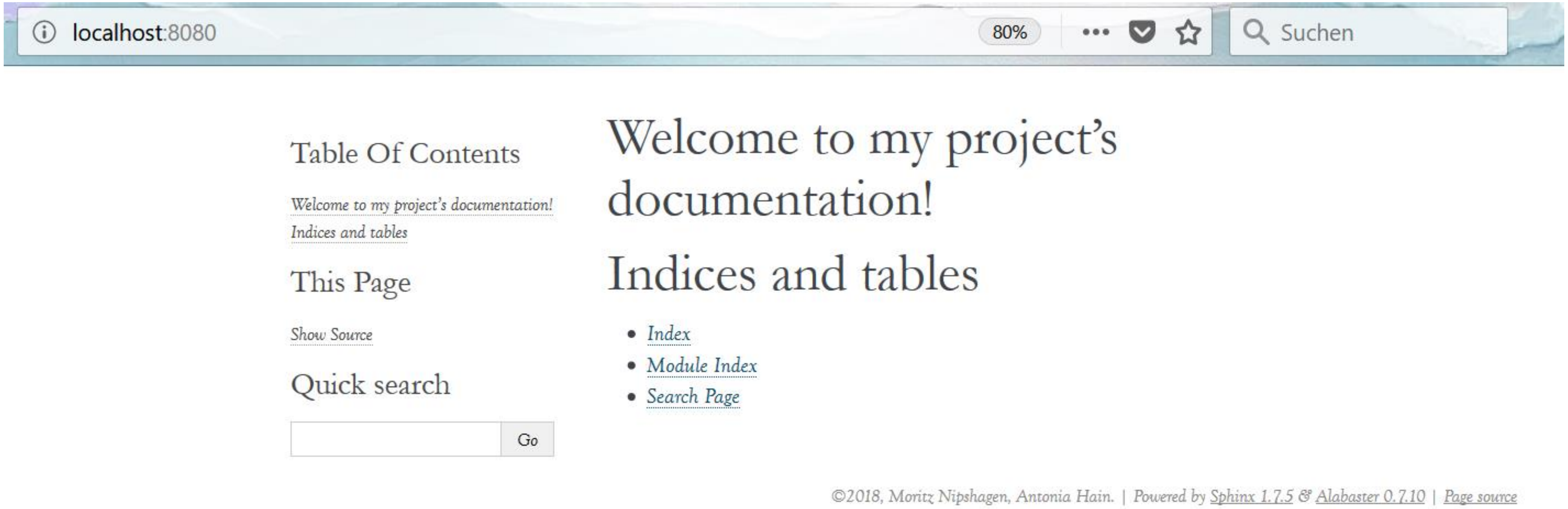


Table Of Contents

Welcome to my project's documentation!
Indices and tables

This Page

Show Source

Quick search

Welcome to my project's documentation!

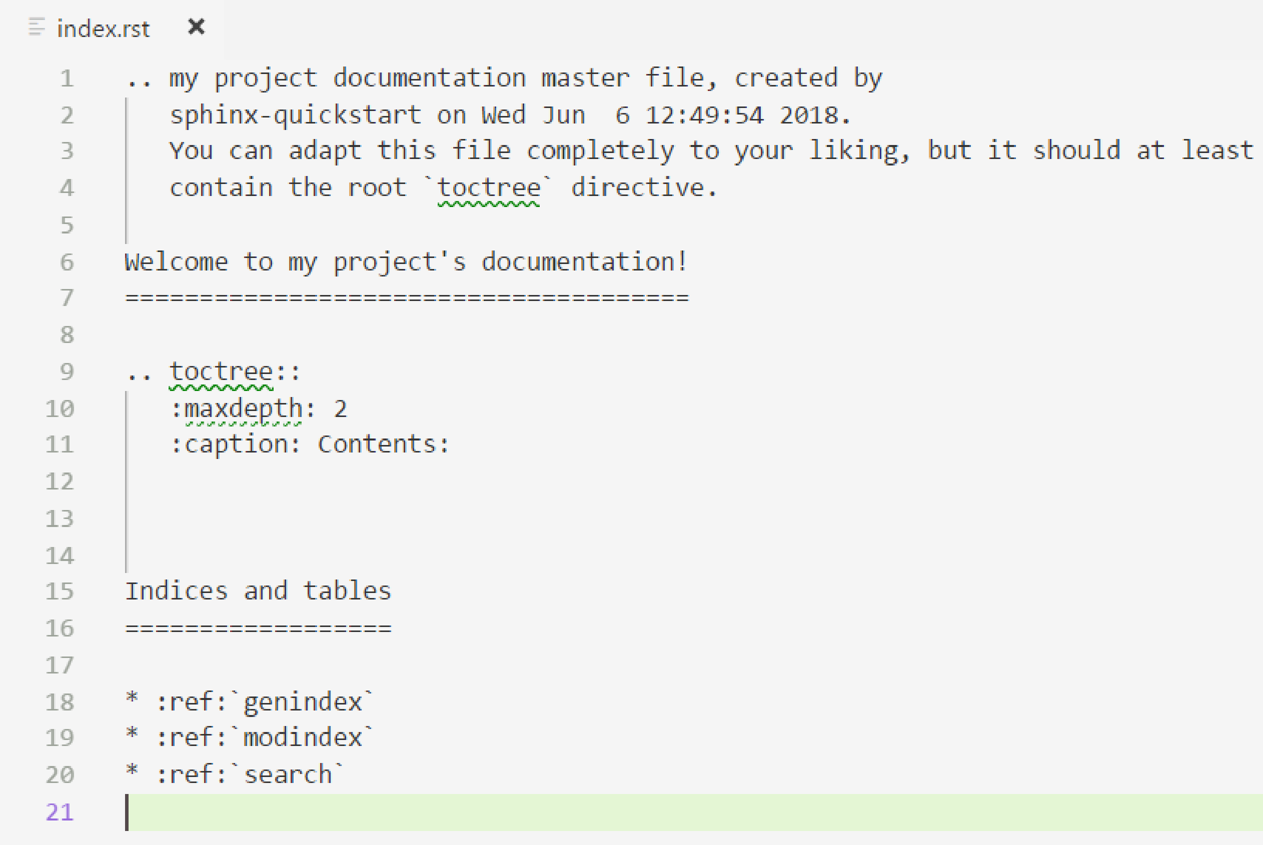
Indices and tables

- Index
- Module Index
- Search Page

©2018, Moritz Nipshagen, Antonia Hain. | Powered by Sphinx 1.7.5 & Alabaster 0.7.10 | Page source

Sphinx

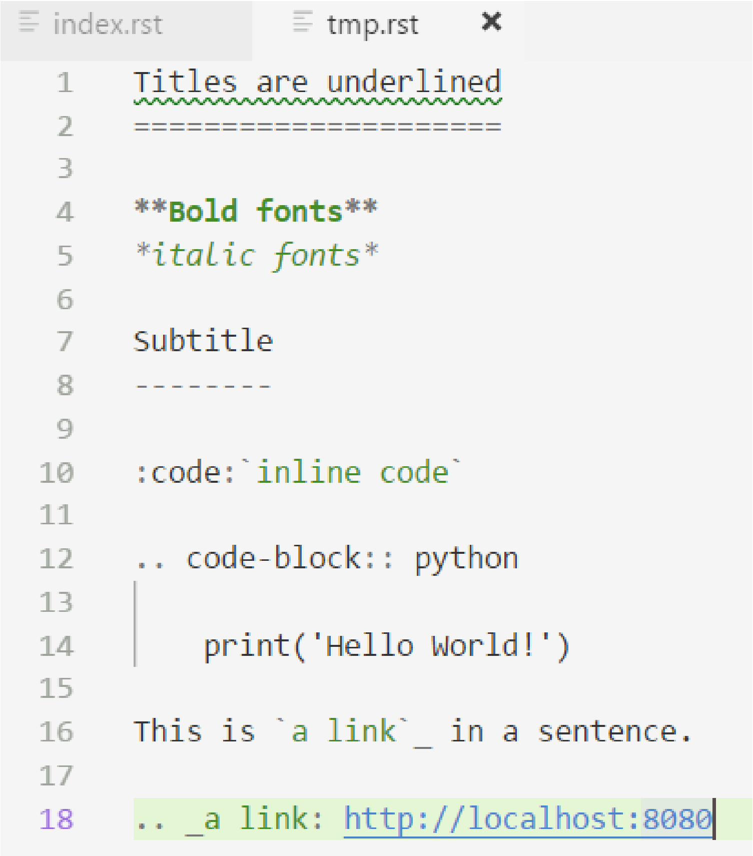
- To change what you see, you need to edit the index.rst file
- Then rebuild the project with make html



```
.. my project documentation master file, created by
   sphinx-quickstart on Wed Jun  6 12:49:54 2018.
   You can adapt this file completely to your liking, but it should at least
   contain the root `toctree` directive.

Welcome to my project's documentation!
======================================

.. toctree::
   :maxdepth: 2
   :caption: Contents:



Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`

```

Sphinx

- The format that you see used inside the file is a form of markup called restructured text (rst)
  - You can find more about rst and its rules here: www.sphinx-doc.org/en/master/usage/restructuredtext/index.html
  - You can try it out online at http://rst.ninjs.org/#



```
Titles are underlined
=====================

**Bold fonts**
*italic fonts*

Subtitle
--------

:code:`inline code`

.. code-block:: python

    print('Hello World!')

This is `a link`_ in a sentence.

.. _a link: http://localhost:8080
```

Sphinx Extensions

- Remember to use proper doc comments
- This can be extracted and used by Sphinx
- To this purpose Sphinx includes a couple of extensions
  - E.g. in the index file, this is used to include the modules folder inside the docs folder:

```
.. toctree::
    :maxdepth: 2
    :caption: Contents:

   modules/modules
```

```
class SampleClass(object):
    """Summary of class here.

    Longer class information....
    Longer class information....

    Attributes:
        likes_spam: A boolean indicating if we like SPAM
        or not.
        eggs: An integer count of the eggs we have laid.
    """

    def __init__(self, likes_spam=False):
        """Inits SampleClass with blah."""
        self.likes_spam = likes_spam
        self.eggs = 0

    def public_method(self):
        """Performs operation blah."""
```

From https://google.github.io/styleguide/pyguide.html#Comments

Homework

Next Homework

- Make a dramatic countdown!
- Log the time!
- Calculate how much time passed since your birthday!

See you all next week!