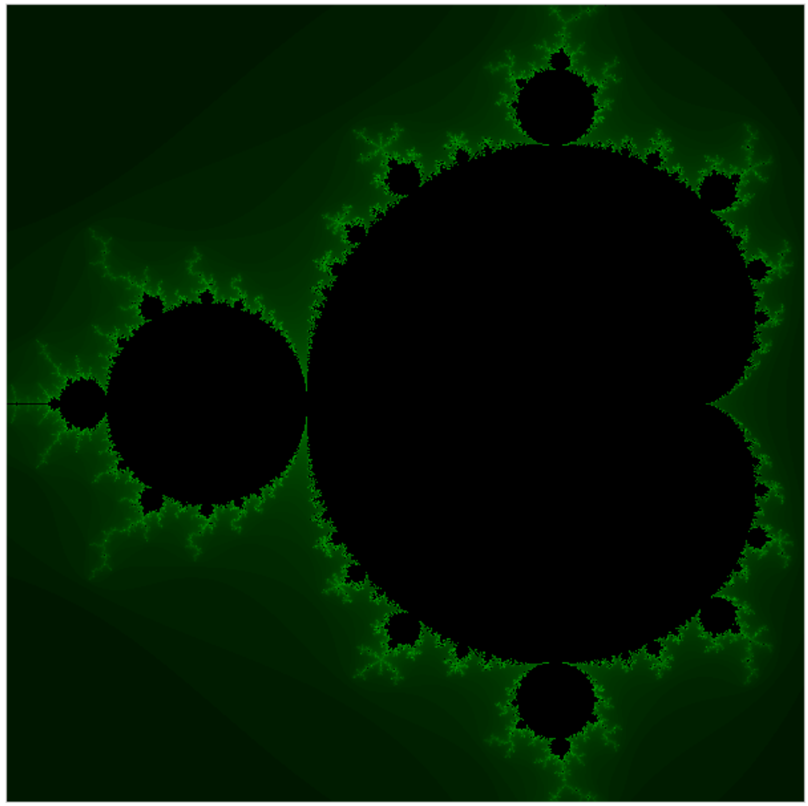
2IS50 – Software Development for Engineers – 2022-2023

Lecture 1.B (Sw. Eng.)

Lecturer: Tom Verhoeff



Review of Lecture 1.A (Python)

- `while` loop, unbounded repetition
  - invariant relation
  - termination, `break`, `continue`, `else`
  - nesting
- Defining functions: `def`
  - parameters, type hints
  - `return`, return type, void or fruitful
  - docstring (first sentence ends in a period)
  - local variables
  - default arguments

Preview of Lecture 1.B (Sw. Eng.)

- Software Engineering, looking beyond programming
  - Software development *process*

    - Dealing with errors
        - Python coding standard
        - `assert` statement
        - Pair programming
        - Systematic testing
- Version control: **Git**
- PyCharm: Python Integrated Development Environment (IDE)

Software Engineering

Engineering =

- Application of *scientific* principles to
- *design*, *construction*, *operation*, and *maintenance* of (technological) products

Scientific = *systematic*, *disciplined*, *quantifiable*, *supported by theory*

Software Engineering (SE) = Engineering of software products

Software Engineering versus Programming

Software Engineering goes beyond programming

Programming concerns (program) 'code':

- Syntax (form)
- Semantics (meaning)
- Pragmatics (practical aspects)

Software Engineering also includes:

- (Problem) Domain Analysis/Engineering
- Requirements Management/Engineering
- Project Management (staffing, cost, schedule, risks)
- Quality Assurance/Engineering
- Release Management
- Maintenance

- Version Control (who did/may change what when why)
    - **Change management**
    - **Issue tracking**
- Verification and Validation
    - **Code review**
    - **Testing**

2IS50 will address

- Version Control (`git`)
- Verification (pair programming, `doctest`, `pytest`)
- Documentation

Software development is a *process*

Don't expect to get everything right on the first try.

1. Write down/analyze the intended purpose of your program.
2. Write a small program, without functions, that does something minimal.
3. *Test it*; fix errors if necessary.
4. Add further features to the program, and *test again*.
5. Identify opportunities to encapsulate program fragments into a function.
6. Generalize functions by introducing extra parameters.

Multiple short cycles of

- problem analysis
- design (problem decomposition)
- coding, documenting
- reviewing, testing

Next cycles:

- fixing (repair defects)
- refactoring (improve structure)
- enhancing (add features)

Dealing with Errors

- People make mistakes (invariant fact)
- Engineers must take this into account
  - Deal with errors
  - Ignoring them is not an option

Terminology

- **Mistake**: made by people
  - slip of the mind or keyboard
  - causing a *fault*

Defect, **fault** (jargon: 'bug')
  - anomaly in a product
  - can cause a *failure*

- **Failure**: when product in use deviates from spec

- **Error**: difference between actual and expected result
  - assumes a predefined expectation

How to Deal with Defects

1. **Admit** that people make mistakes (don't ignore/punish)
2. **Prevent** mistakes as much as possible
3. **Minimize** consequences of mistakes
4. **Detect** presence of defects a.s.a.p.
5. **Localize** defects
6. **Repair** defects
7. **Trace** failures -> defects -> mistakes -> root causes
8. **Learn** to **improve** the process and tools

Prevention: Coding Standard

- Write 'clean' *readable* code
  - layout: indentation, spacing, empty lines

- Write *understandable* code
  - meaningful names, comments, docstrings, type hints
  - small functions, shallow nesting

Adhere to our [Python Coding Standard]

`assert` statement

Syntax of **`assert` statement**:

```
assert condition, message
```

Semantics of `assert` statement:

1. Evaluate condition
2. If condition evaluates to `True`,
   then do nothing,
   else *raise* `AssertionError` with given message (interrupts *flow of control*)

(See *Think Python*, Section 16.5.)

In [1]:
```python
def print_line_2(n: int) -> None:
    """Print a line of n (n >= 0) times letter "o".
    """
    assert n >= 0, "n must be >= 0 (n == {})".format(n)
    print(n * "o")
```

In [2]:
```python
print_line_2(-1)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last
)
/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_9589/2065839862
.py in <module>
----> 1 print_line_2(-1)

/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_9589/2723797084
.py in print_line_2(n)
      2     """Print a line of n (n >= 0) times letter "o".
      3     """
----> 4     assert n >= 0, "n must be >= 0 (n == {})".format(n)
      5     print(n * "o")

AssertionError: n must be >= 0 (n == -1)
```

In [3]:
```python
%xmode
```

```
Exception reporting mode: Verbose
```

`assert` helps to **minimize consequences of**, **detect**, and **localize** defects

- rather than think the defect is *inside* the function
- you now know it is *outside*
- `assert` is a **built-in** test case!

Code review

Reading code *with the purpose of finding defects*

- Does not require execution
  - Hence, does not require a complete program
- When you *detect* a defect this way, you have *localized* it

Pair programming

In pair programming there are two *roles*:

- **driver** (who controls the keyboard)
- **observer** (who *reviews code* on screen)
  or **navigator** (who advises/questions the driver)
- in real-time dialog

Can be done *online* via **screen sharing** or in PyCharm via **[Code with Me]**:

- Driver shares screen with navigator
- Audio connection

Important advice: **Regularly switch roles**

- In 2IS50: pair programming is **only applied in Homework Assignments 1 & 2**

Testing

Executing code *with the purpose of finding defects*

- Needs a *complete* program
- You *detect defects* through their *failures*
- Does not necessarily *localize* defects

Nested loops: inner loop hard to test in isolation:

In [4]:
```
N = 12  # table size
a = 1
# invariant: rows from a through N need printing

while a <= N:
    b = 1
    # invariant: in row a, columns from b through N need printing

    while b <= N:
        print(" {:3}".format(a * b), end='')
        b = b + 1

    print()
    a = a + 1
```

```
   1   2   3   4   5   6   7   8   9  10  11  12
   2   4   6   8  10  12  14  16  18  20  22  24
   3   6   9  12  15  18  21  24  27  30  33  36
   4   8  12  16  20  24  28  32  36  40  44  48
   5  10  15  20  25  30  35  40  45  50  55  60
   6  12  18  24  30  36  42  48  54  60  66  72
   7  14  21  28  35  42  49  56  63  70  77  84
   8  16  24  32  40  48  56  64  72  80  88  96
   9  18  27  36  45  54  63  72  81  90  99 108
  10  20  30  40  50  60  70  80  90 100 110 120
  11  22  33  44  55  66  77  88  99 110 121 132
  12  24  36  48  60  72  84  96 108 120 132 144
```

Put inner loop in separate function to improve testability:

In [5]:
```
N = 12  # table size
```

```python
def print_row(a: int) -> None:
    """Print row a for multiplication table of size N.
    """
    b = 1
    # invariant: columns from b through N need printing

    while b <= N:
        print(" {:3}".format(a * b), end='')
        b = b + 1

    print()


a = 1
# invariant: rows from a through 12 need printing

while a <= N:
    print_row(a)
    a = a + 1
```

```
   1   2   3   4   5   6   7   8   9  10  11  12
   2   4   6   8  10  12  14  16  18  20  22  24
   3   6   9  12  15  18  21  24  27  30  33  36
   4   8  12  16  20  24  28  32  36  40  44  48
   5  10  15  20  25  30  35  40  45  50  55  60
   6  12  18  24  30  36  42  48  54  60  66  72
   7  14  21  28  35  42  49  56  63  70  77  84
   8  16  24  32  40  48  56  64  72  80  88  96
   9  18  27  36  45  54  63  72  81  90  99 108
  10  20  30  40  50  60  70  80  90 100 110 120
  11  22  33  44  55  66  77  88  99 110 121 132
  12  24  36  48  60  72  84  96 108 120 132 144
```

In [6]:
```python
print_row(1)  # trhoug this function, inner loop is testable
```

```
   1   2   3   4   5   6   7   8   9  10  11  12
```

Introduce extra parameter to improve testability further:

In [7]:
```python
def print_row_2(a: int, n: int = 12) -> None:
    """Prints row a for multiplication table of size n.
    """
    b = 1
    # invariant: columns from b through 12 need printing

    while b <= n:
        print(" {:3}".format(a * b), end='')
        b = b + 1

    print()
```

In [8]:
```python
print_row_2(7, 3)
```

```
   7  14  21
```

Debugging

The process of *localizing* and *repairing* detected defects

- It can be hard to localize a defect based on a failure
- It can be unpredictable effort and frustrating

Techniques:

- (bad) Add `print` statements to observe intermediate results
  Must later be removed/suppressed (by commenting out)
- (better) Refactor code to make intermediate results accessible
  Add test cases (which you can keep and reuse later)

How to Test Systematically

For each **test case**:

1. Decide on **inputs**

    - Boundary/special cases
    - Typical cases
    - 'Large' cases (when performance matters)

2. Decide on which **outputs** to observe
3. Determine **expected** result
4. Execute test case
5. Compare **actual result** with **expected result**
6. Decide on **pass/fail**

Manual versus Automated Test Cases

- **Manual**: human executes code
    - types in input
    - looks at output
    - compares with expectation
    - decides about pass/fail

- **Automated**: write test code
    - to select inputs
    - to execute/run/call 'subject under test' (SUT)
    - to observe outputs
    - to compare actual and expected result
    - to decide and report on pass/fail

Some code that is not immediately undestandable:

In [9]:
```python
def root(n: int) -> int:
    """Return integer square root of n (n >= 0).
    """
    assert n >= 0, "n must be >= 0"
    r, s = 0, n + 1  # invariant: 0 <= r <= sqrt(n) < s

    while s - r != 1:
        m = (r + s) // 2  # r < m < s
        if m * m < n:  # m <= sqrt(n)
            r = m
        else:  # sqrt(n) < m
            s = m
        # s - r is now roughly halved

    #  s == r + 1, hence r <= sqrt(n) < r + 1
    return r
```

In [10]:
```python
# Manual systematic test cases
(
root(0),  # boundary case, expect 0
root(1),  # expect 1
root(2),  # expect 1
root(3),  # expect 1
root(4),  # expect 2
root(100),  # expect 10
root(1000),  # expect 31
)
```

Out[10]:
```
(0, 0, 1, 1, 1, 9, 31)
```

Conclusion: Failure! Hence, code contains defect(s)

Now test again with `<=` instead of `<` on line 9

Also test that safety net works:

In [11]:
```python
root(-1)  # invalid case, expect exception
```

```
---------------------------------------------------------------------------
-
AssertionError                            Traceback (most recent call last
)
/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_9589/3219956217
.py in <module>
----> 1 root(-1)  # invalid case, expect exception
        global root = <function root at 0x7fc7f06b68b0>

/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_9589/1171836655
.py in root(n=-1)
      2     """Return integer square root of n (n >= 0).
      3     """
----> 4     assert n >= 0, "n must be >= 0"
        n = -1
      5     r, s = 0, n + 1  # invariant: 0 <= r <= sqrt(n) < s
```

```
    6
```

```
AssertionError: n must be >= 0
```

What would happen to `root(-1)` without `assert`?

Systematic Testing Advice

- N.B. Keep your test cases in your notebook
- Don't throw them away (need ability to repeat)

- Testing itself is predictable effort
- Testing is challenging (find few good test cases)

- Later: What are good test cases? How to construct them?
- Later: How to automate testing?

Version Control

- Configuration management
  - Know what (code, etc.) you have
  - Store it safely
- Version management
  - Know which versions are used where
- **Change management**
  - Know who did/can change what when why
  - Ability to go back to earlier version
  - Don't lose changes: concurrent overwrite problem
- **Issue tracking**
  - Record known defects, feature requests

Git: Distributed Version Control System

VCS = Version Control System

- Install from https://git-scm.com/downloads
  - There are multiple options
  - Options depend on your OS

- Can be used through **Command-Line Interface** (CLI)
  - Separate app for **Graphical User Interface** (GUI)
  - E.g. PyCharm (next slide)

PyCharm IDE

IDE = Integrated Development Environment

- Install **PyCharm** IDE, *Professional Edition*
  - (Or: IntelliJ IDEA Ultimate, if you are a power user)
  - Can also install these from JetBrains ToolBox (recommended)

- Register for free *academic* license with your TU/e email address

- It should find your Python and Git installations

Clone the 2IS50 Study Material GitLab Repository

- Browse to GitLab repo
  - Click the blue **Clone** button
  - Click the copy icon for **Clone with HTTPS**

- In PyCharm menu: `VCS > Get from Version Control ...`
  - Version control: `Git`
  - URL: paste the copied URL
  - Directory: navigate to an emtpy directory (create if necessary)
    - New directory name (e.g.): `study-material-2is50-2021-2022`
  - Click **Clone**, and wait for data transfer
  - If asked, open project in a new window
  - The `README.md` file opens

Advice on using clone of GitLab repo

- Do not 'work' (read: edit) in the clone
  - Copy files to separate working directory

- Get all updates from master repository:
  - In PyCharm menu: `VCS > Update Project ...`
    - Merge incoming changes into the current branch
  - Also: keyboard shortcut and speed button

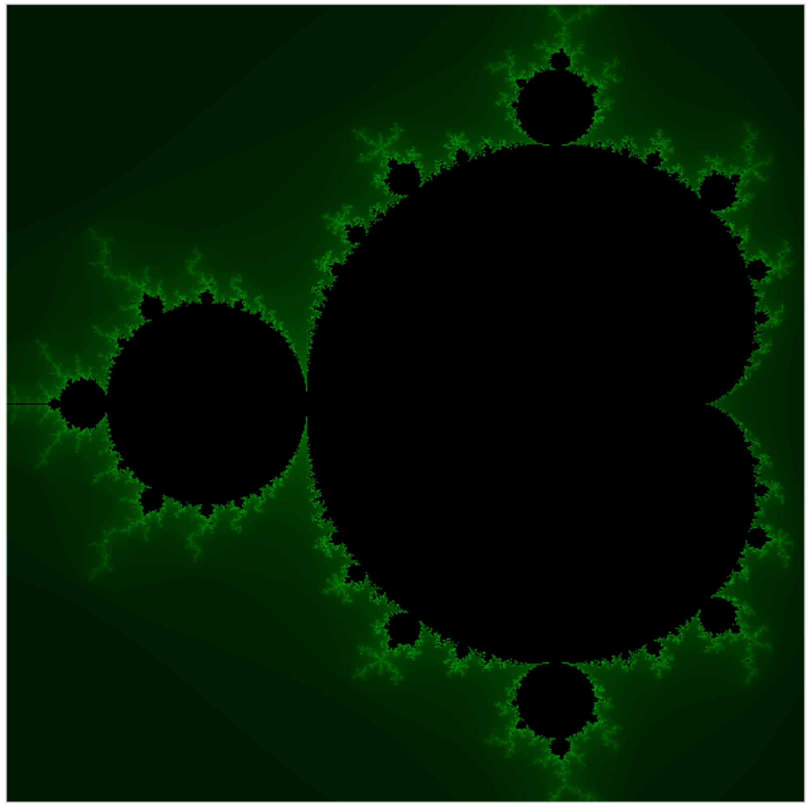
- If you did edit and get a **conflict**
  - If needed, copy edited file to outside the clone
  - Accept all incoming changes (overwrites your changes)

(End of Notebook)

2IS50 – Software Development for Engineers – 2022-2023

Lecture 2.B (Sw. Eng.)

Lecturer: Tom Verhoeff



Review of Lecture 1.B

- Software Engineering, more than programming
- Dealing with errors
  - Python coding standard
  - `assert` statement
  - Pair programming
  - Systematic testing: manually
- Version control: **Git**
- **PyCharm**: Python Integrated Development Environment (IDE)

Preview of Lecture 2.B

- Coding Standard: naming, docstring conventions
- Automated testing
  - `doctest` examples
  - `pytest`
- Test case design

        - Code coverage
- Version control
    - change sets
    - pull/commit/push/conflict
    - Trunk-Based Development (TBD)

Python Coding Standard

- Adhering to a standard does not make a program *work* better
- Not adhering does make a program worse, in ...
    - getting it to work
    - understanding it
    - modifying it
- Standards are there to help (saves time, in the longer run)
    - Reduces risk of making mistakes

Comments

Comments should not just tell what a statement is doing.

- Rather, they should focus on *why* it is done.
- You may assume that the reader of the comments knows Python.

```
n = 0  # set n to zero (USELESS COMMENT)

c, i = 0, 0  # c == word[:i].count('a')  (BETTER)
```

That last comment is an **invariant** (a *relationship* that holds between `c`, `i`, and `word`).

Names

- Naming conventions:
    - **Constants**: noun phrase in UPPER case with underscores
        - `CONFIG_FILE_NAME`
    - **Variables**: noun phrase in lower case with underscores
        - `passenger_names`
    - **Functions**: verb phrase in lower case with underscores
        - `load_passenger_data`
    - **Classes**: singular noun phrase in CamelCasing
        - `RandomPlayer`

- Cannot use Python **keywords** as name
- Avoid names that are also used for built-in functions:
    - `sum`, `min`, `max`, `str`, `list`
    - Using them makes built-in function inaccessible

In [1]:
```python
from pprint import pprint
import keyword, builtins

pprint(keyword.kwlist, compact=True)
pprint(dir(builtins), compact=True)
```

```
['False', 'None', 'True', '__peg_parser__', 'and', 'as', 'assert', 'async'
,
 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'exc
ept',
 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda',
 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with
',
 'yield']
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarn
ing',
 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationE
rror',
 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError',
 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError',
 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWar
ning',
 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceErro
r',
 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration'
,
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemEx
it',
 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',
 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError',
 'Warning', 'ZeroDivisionError', '__IPYTHON__', '__build_class__', '__debu
g__',
 '__doc__', '__import__', '__loader__', '__name__', '__package__', '__spec
__',
 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray',
 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyrig
ht',
 'credits', 'delattr', 'dict', 'dir', 'display', 'divmod', 'enumerate', 'e
val',
 'exec', 'execfile', 'filter', 'float', 'format', 'frozenset', 'get_ipytho
n',
 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'i
nt',
 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', '
map',
 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow'
,
 'print', 'property', 'range', 'repr', 'reversed', 'round', 'runfile', 'se
t',
```

```
 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tup
le',
 'type', 'vars', 'zip']
```

- Trade-off between short and long names
  - The wider the *scope* of a name, the more helpful a (longer) self-documenting name is.
    - Scope = lines where name has same meaning
  - **Local names** can be short(er), but do use a **comment** to explain their purpose.
  - **Invariants** make good comments.

Type Hints

- `list` versus `List[int]`
- `tuple` versus `Tuple[int, int]` versus `Tuple[int, ...]`
- Void functions have return type `None`: `def f() -> None`
  - Technically speaking, `None` is not a type
  - `None` indicates the absence of a return type

N.B. Python >= 3.9 supports `list[int]`, etc.

Docstring Conventions

- Always use a pair of **triple double-quotes**, placed on *separate lines*.

  ```
  """First sentence must be a complete summary
    (without details) in _imperative_ mood,
    terminated by a period.

    Details (e.g. assumptions) follow after an empty line.
    """
  ```

  Note the indentation of the lines w.r.t. the quotes.

- Start with a single short *summary sentence*, ending in a period.
  - Use *imperative mood*: 'Print report', and *not* 'Prints report.'
  - Strive for completeness, without details.
- Separate the summary from following lines (with details) by an *empty line*.

Reason: Tools use this format to extract the summary

- Don't repeat type information as such.
- Refer to parameters by their name.
- Explicitly mention important *assumptions* and *side-effects* in the details part.

In [2]:
```
# examples ... see below
```

Functions and Side-effects

- In *mathematics,* functions are only interesting because of the value they 'return' for given arguments.
  - Applying a math function to the same argument always gives the same result
- In *programming,* functions can also have **side-effects**:
  - on each call, they can return a *different* value for the same argument (`random.choice`)
  - or, they don't deliver any result (**void functions** like `print` or `list.sort`)

- Side-effects make *reasoning* about functions harder.
  - So, use with care (see 1st example below).
  - Document side effects in the docstring
- *Mutability* can cause surprises (see 2nd example below).

In [3]:
```python
from random import randint

([2 * randint(1, 6)   # always even!
  for _ in range(10)],

 [randint(1, 6) + randint(1, 6)   # can be odd
  for _ in range(10)]
)
```

Out[3]: ([8, 10, 4, 6, 4, 6, 6, 8, 8, 10], [7, 9, 6, 10, 9, 10, 6, 7, 7, 8])

In [4]:
```python
def reverse_list(lst: list) -> list:
    """Reverse lst.  (Warning: DOCSTRING NOT GOOD ENOUGH)
    """
    i, j = 0, len(lst) - 1  # lst[i:j+1] must still be reversed
    # invariant: 0 <= i and j < len(lst)

    while i < j:
        lst[i], lst[j] = lst[j], lst[i]
        i, j = i + 1, j - 1

    return lst
```

In [5]:
```python
a = [1, 0, 2, 4]
b = reverse_list(a)

a, b  # aliasing!
```

Out[5]: ([4, 2, 0, 1], [4, 2, 0, 1])

- In `reverse_list`, parameter `lst` is bound to a `list` object, which is *mutable*.
- Operations on `lst` modify the object that `lst` is bound to.

It is highly recommended to document this in the docstring:

In [6]:
```python
def reverse_list(lst: list) -> list:
    """Reverse lst.
```

```
    Modifies lst IN PLACE, and returns the reverse as well.
    """
    i, j = 0, len(lst) - 1  # lst[i:j+1] must still be reversed
    # invariant: 0 <= i and j < len(lst)

    while i < j:
        lst[i], lst[j] = lst[j], lst[i]
        i, j = i + 1, j - 1

    return lst
```

- It might be even clearer, if `reverse_list` were a *void function*.
  - Alternatively: create a fresh result list with the reverse
- But sometimes it is useful to return the modified list, so that it can be used inside an experession.

In [7]:
```
def reverse_list(lst: list) -> None:
    """Reverse lst.

    Modifies lst IN PLACE.
    """
    i, j = 0, len(lst) - 1  # lst[i:j+1] must still be reversed
    # invariant: 0 <= i and j < len(lst)

    while i < j:
        lst[i], lst[j] = lst[j], lst[i]
        i, j = i + 1, j - 1
```

In [8]:
```
a = [1, 0, 2, 4]
b = reverse_list(a)

a, b  # aliasing!
```

Out[8]:
```
([4, 2, 0, 1], None)
```

In [9]:
```
def reverse_list(lst: list) -> list:
    """Return a reversed copy op lst.

    Modifies lst IN PLACE.
    """
    return [item for item in lst[::-1]]
```

In [10]:
```
a = [1, 0, 2, 4]
b = reverse_list(a)

a, b  # aliasing!
```

Out[10]:
```
([1, 0, 2, 4], [4, 2, 0, 1])
```

Systematic Testing

Recap

Testing: Execute program *with intention to detect defects*

Test case

Goal: Try to *break* the program

1. Decide on **inputs**
   - Boundary/special cases
   - Typical cases
   - 'Large' cases (when performance matters)
2. Decide on which **outputs** to observe
   - Function result
   - Modified parameters
   - Modified global variables
   - Modified files, including printed output
3. Determine **expected** result
4. Execute test case
5. Compare **actual result** with **expected result**
6. Decide on **pass/fail**

Function Testing

- Testing a function by *one* call is hardly ever enough.
- Pick a *few important* arguments, for which you can check the corresponding result
- Strive for **problem coverage**:
  - **boundary cases**
  - **special cases**
  - **typical case**
- Strive for **code coverage**
  - Code that isn't executed during the call, isn't tested
  - Cover all branches of `if-elif-else`
- You don't need to check the result *directly*
  - Could test it *indirectly* (see next example)

In [11]:
```python
import random
from typing import List
```

In [12]:
```python
def roll_dice(n: int) -> List[int]:
    """Roll n regular dice and return outcomes in a list.

    Assumption: n >= 0
    """
    return [random.randint(1, 6) for _ in range(n)]
```

In [13]:
```python
# Manual test cases
```

```
# boundary case
print(roll_dice(0))
# Expected: []

# test length
print(roll_dice(2))
# Expected: list of length 2

# test range of values
print(roll_dice(10))
# Expected: list with values in range(1, 6+1)
```

```
[]
[2, 2]
[6, 5, 3, 2, 1, 1, 2, 4, 4, 4]
```

Automated Testing

With *one* click

- Execute each test case
  - Call function with selected arguments
  - Capture result
  - Check result
    - Either: compare *directly* against *expected result*
    - Or: check *indirectly* for a *property*
- Report on all test outcomes

`doctest` examples in docstring

- You can put **usage examples** in a docstring
- You can do it in such a format that these examples are **automatically executable and checkable**

Format of **doctest** examples/test cases in docstring:

```
>>> expression with function call
expected result
...
...
>>> expression with function call
expected result
```

In [14]:

```
def roll_dice(n: int) -> List[int]:
    """Roll n regular dice and return outcomes in a list.

    Assumption: n >= 0.

    Examples and test cases:
```

```
    >>> roll_dice(0)  # boundary case
    []
    >>> len(roll_dice(2))  # test length
    2
    >>> all(roll in range(1, 6+1) for roll in roll_dice(10))  # test value
s
    True
    """
    return [random.randint(1, 12) for _ in range(n)]
```

Note that:

- Boundary test case is a *direct* test
- Other test cases are *indirect*
  - They check for a desirable property, by applying a function to the result, and inspecting that

How to run `doctest` examples in Jupyter notebook

- `import doctest`
- `doctest.run_docstring_examples(func, globals(), verbose=True, name='...')`
  - Runs all test cases of `func`, reporting *all details*
- `doctest.run_docstring_examples(func, globals(), verbose=False)`
  - Runs all test cases, *only reporting failures*

In [15]:

```
import doctest
```

In [16]:

```
doctest.run_docstring_examples(roll_dice, globals(), verbose=True, name='r
oll_dice')
```

```
Finding tests in roll_dice
Trying:
    roll_dice(0)  # boundary case
Expecting:
    []
ok
Trying:
    len(roll_dice(2))  # test length
Expecting:
    2
ok
Trying:
    all(roll in range(1, 6+1) for roll in roll_dice(10))  # test values
Expecting:
    True
**********************************************************************
File "__main__", line 12, in roll_dice
Failed example:
    all(roll in range(1, 6+1) for roll in roll_dice(10))  # test values
Expected:
```

```
    True
Got:
    False
```

Did you spot the defect in the code for `roll_dice`?

You can also run the test cases more quietly, only showing test cases that failed:

In [17]:
```python
doctest.run_docstring_examples(roll_dice, globals(), verbose=False, name='
roll_dice')
```

```
**********************************************************************
File "__main__", line 12, in roll_dice
Failed example:
    all(roll in range(1, 6+1) for roll in roll_dice(10))  # test values
Expected:
    True
Got:
    False
```

Two more examples of tests in docstring:

In [18]:
```python
OPTIONS = {0: "Rock", 1: "Paper", 2: "Scissors"}
RPS = ''.join(name[0].lower() for name in OPTIONS.values())

def rps_choice(letter: str) -> int:
    """Return choice integer corresponding to given letter.

    The letter is first converted to lower case.

    Assumptions:
    * len(letter) == 1
    * letter.lower() in RPS

    >>> rps_choice('r')
    0
    >>> rps_choice('P')
    1
    >>> rps_choice('s')
    2
    >>> rps_choice('X')
    Traceback (most recent call last):
        ...
    AssertionError: letter.lower() must be in RPS
    """
    assert letter.lower() in RPS, "letter.lower() must be in RPS"

    return RPS.index(letter.lower())
```

In [19]:
```python
doctest.run_docstring_examples(rps_choice, globals(), verbose=False, name=
'rps_choice')
```

Note that we also tested for **expected exceptions**.

In [20]:
```python
def beats(choice_1: int, choice_2: int) -> bool:
    """Return whether choice_1 beats choice_2.

    Assumption: choice_1 in OPTIONS and choice_2 in OPTIONS

    :param choice_1: choice of first player
    :param choice_2: choice of second player
    :return: whether choice_1 beats choice_2

    :examples:

    >>> beats(0, 0)
    False
    >>> beats(0, 1)
    False
    >>> beats(1, 0)
    True
    >>> beats(0, 2)
    True
    """
    return choice_1 > choice_2  # (choice_1 - choice_2) % 3 == 1
```

In [21]:
```python
doctest.run_docstring_examples(beats, globals(), verbose=False, name='beat
s')
```

```
**********************************************************************
File "__main__", line 18, in beats
Failed example:
    beats(0, 2)
Expected:
    True
Got:
    False
```

So, there is a defect ... somewhere

Exhaustive testing usually not possible/desirable.

Separate test cases (not in docstring; *note the indentation*):

In [22]:
```python
beats_test = """
    >>> beats(2, 2)   # was missed above!
    False
"""
```

In [23]:
```python
doctest.run_docstring_examples(beats_test, globals(), verbose=True, name='
beats_test')
```

```
Finding tests in beats_test
Trying:
    beats(2, 2)   # was missed above!
Expecting:
    False
```

```
ok
```

Test calls must produce *predictable* output:

- Sets and dictionaries do *not* print in a specific reproducible order
- Instead:
  - turn them into a sorted list, or
  - compare them to expected result

In [24]:
```
set_test = """
    >>> set("asdf")
    {'a', 's', 'd', 'f'}
"""
```

In [25]:
```
doctest.run_docstring_examples(set_test, globals(), verbose=True, name='se
t_test')
```

```
Finding tests in set_test
Trying:
    set("asdf")
Expecting:
    {'a', 's', 'd', 'f'}
**********************************************************************
Line 2, in set_test
Failed example:
    set("asdf")
Expected:
    {'a', 's', 'd', 'f'}
Got:
    {'s', 'a', 'd', 'f'}
```

In [26]:
```
set_test = """
    >>> set("asdf") == {'a', 's', 'd', 'f'}
    True
"""
```

In [27]:
```
doctest.run_docstring_examples(set_test, globals(), verbose=True, name='se
t_test')
```

```
Finding tests in set_test
Trying:
    set("asdf") == {'a', 's', 'd', 'f'}
Expecting:
    True
ok
```

Disadvantage: you won't see the actual set value in case of a failure

In [28]:
```
set_test = """
    >>> sorted(set("asdf"))
    ['a', 'd', 'f', 's']
"""
```

In [29]:
```
doctest.run_docstring_examples(set_test, globals(), verbose=True, name='se
t_test')
```

```
Finding tests in set_test
Trying:
    sorted(set("asdf"))
Expecting:
    ['a', 'd', 'f', 's']
ok
```

How to run `doctest` examples for all functions

- `doctest.testmod(verbose=True)` runs all test cases, reporting details
- `doctest.testmod(verbose=False)` runs all test cases, showing a summary

In [30]:
```
doctest.testmod(verbose=True)   # with details
```

```
Trying:
    beats(0, 0)
Expecting:
    False
ok
Trying:
    beats(0, 1)
Expecting:
    False
ok
Trying:
    beats(1, 0)
Expecting:
    True
ok
Trying:
    beats(0, 2)
Expecting:
    True
**********************************************************************
File "__main__", line 18, in __main__.beats
Failed example:
    beats(0, 2)
Expected:
    True
Got:
    False
Trying:
    roll_dice(0)   # boundary case
Expecting:
    []
ok
Trying:
    len(roll_dice(2))   # test length
Expecting:
    2
ok
Trying:
```

```
    all(roll in range(1, 6+1) for roll in roll_dice(10))  # test values
Expecting:
    True
**********************************************************************
File "__main__", line 12, in __main__.roll_dice
Failed example:
    all(roll in range(1, 6+1) for roll in roll_dice(10))  # test values
Expected:
    True
Got:
    False
Trying:
    rps_choice('r')
Expecting:
    0
ok
Trying:
    rps_choice('P')
Expecting:
    1
ok
Trying:
    rps_choice('s')
Expecting:
    2
ok
Trying:
    rps_choice('X')
Expecting:
    Traceback (most recent call last):
        ...
    AssertionError: letter.lower() must be in RPS
ok
2 items had no tests:
    __main__
    __main__.reverse_list
1 items passed all tests:
   4 tests in __main__.rps_choice
**********************************************************************
2 items had failures:
   1 of   4 in __main__.beats
   1 of   3 in __main__.roll_dice
11 tests in 5 items.
9 passed and 2 failed.
***Test Failed*** 2 failures.
```

Out[30]: TestResults(failed=2, attempted=11)

In [31]:

```python
doctest.testmod(verbose=False)  # without details
```

```
**********************************************************************
File "__main__", line 18, in __main__.beats
Failed example:
    beats(0, 2)
Expected:
    True
Got:
```

```
    False
**********************************************************************
File "__main__", line 12, in __main__.roll_dice
Failed example:
    all(roll in range(1, 6+1) for roll in roll_dice(10))  # test values
Expected:
    True
Got:
    False
**********************************************************************
2 items had failures:
   1 of   4 in __main__.beats
   1 of   3 in __main__.roll_dice
***Test Failed*** 2 failures.
```

Out[31]:
```
TestResults(failed=2, attempted=11)
```

PyCharm & `doctest`

PyCharm has built-in facilities

- to find and execute `doctest` examples
- to report details about failures

See Homework Assignment 0.

`pytest` test framework

Pytest is a complete **testing framework**

- It uses `assert` statement to check expectations
- Each test case is written as a function
  - Name must start with `test_...`
- Framework has extensive failure reporting
  - Shows details of differences

Testing advice for `pytest`

- Each `test_...` function should contain only *one* test case
  - Do not combine multiple test cases
  - Reason: `test_...` function stops at first failure
- Keep `test_...` functions *independent* of each other
  - Reason: `test_...` functions are executed *in arbitrary order*

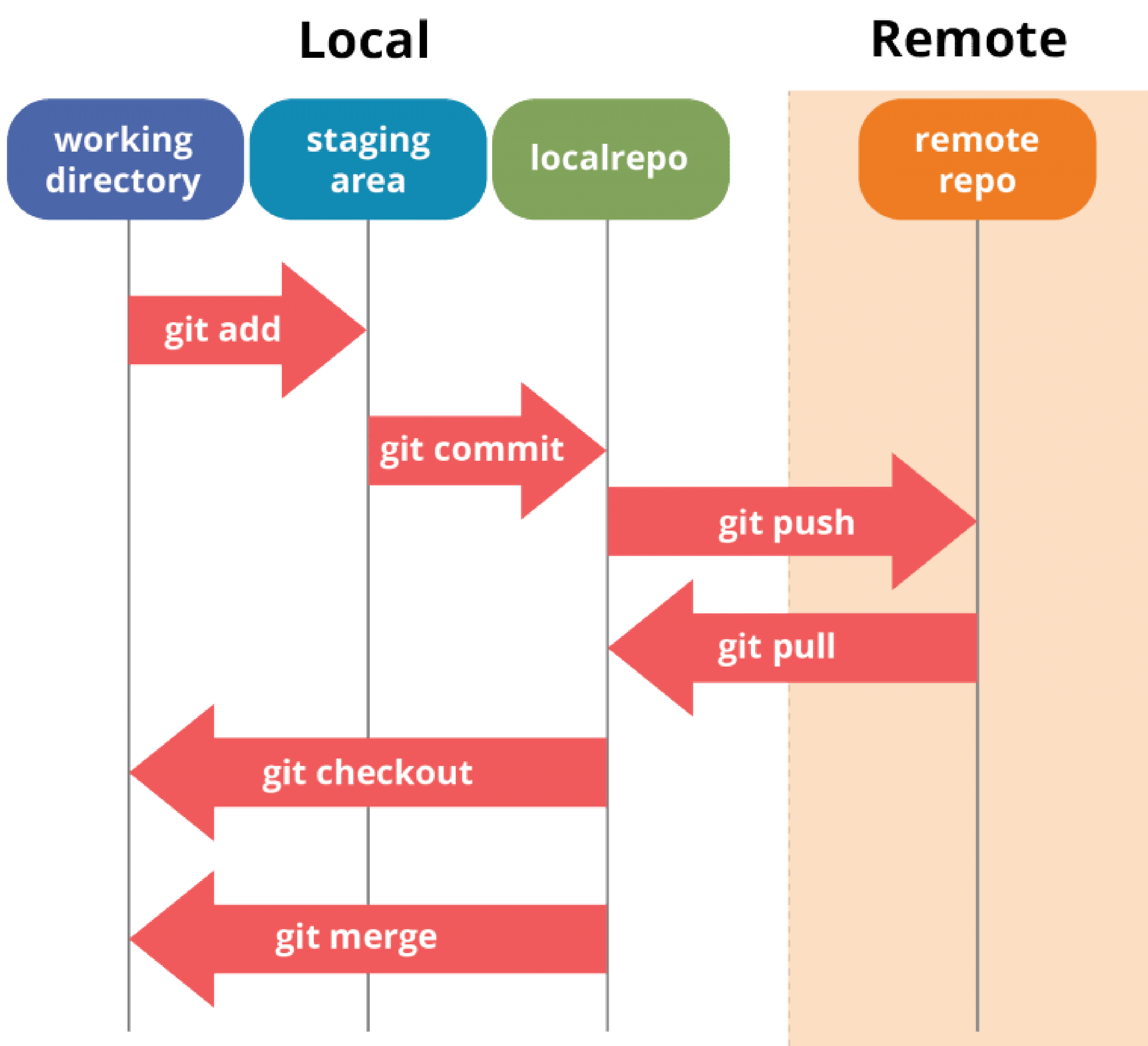
Version Control with Git

- Git can be used on command line (via CLI)
  - or via separate GUI

- We will use Git through PyCharm
- Usage scenarios:
    - Single user
    - Multiple users

Git concepts

- Remote & local *repository* (*repo* in jargon)
    - Repo has: complete history, as sequence of *commits*
    - Each commit has: *change set* & *commit message*
    - Each commit is identified by a *commit hash*
- Working copy, staging area
- Commands
    - Clone (PyCharm: VCS > Get from Version Control...)
    - Add, commit, push
    - Pull (only for multi-user)

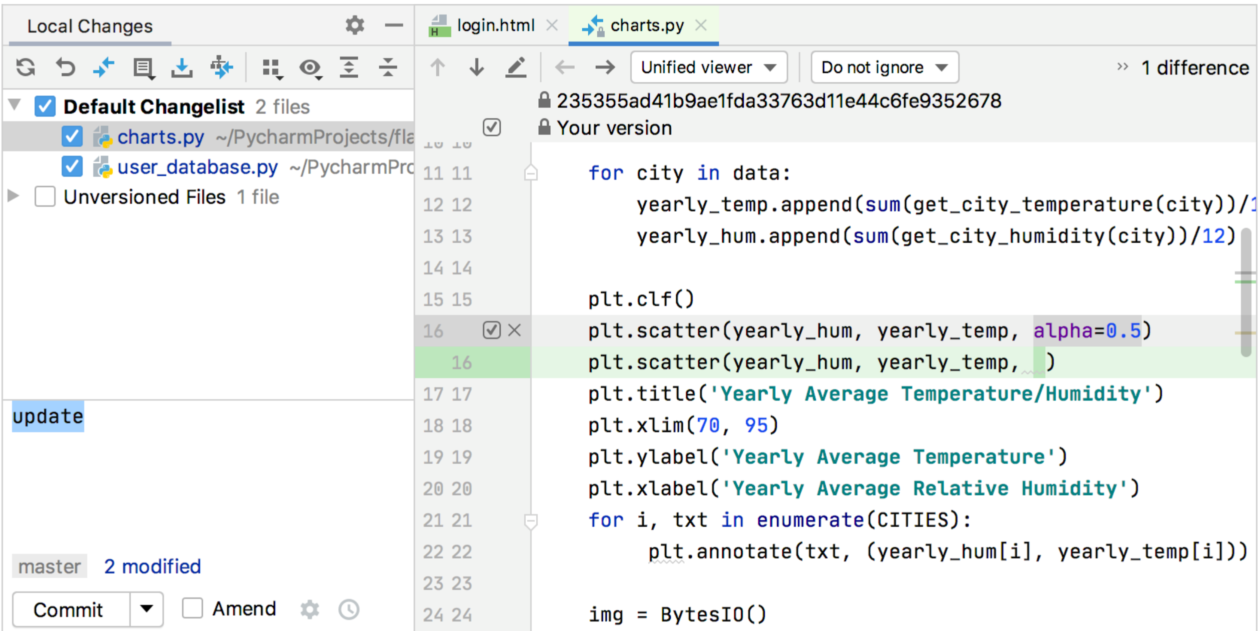


Source: https://www.edureka.co/blog/git-tutorial/

PyCharm manages the staging area

- **Add file to VCS** in PyCharm $\ne$ add to staging area

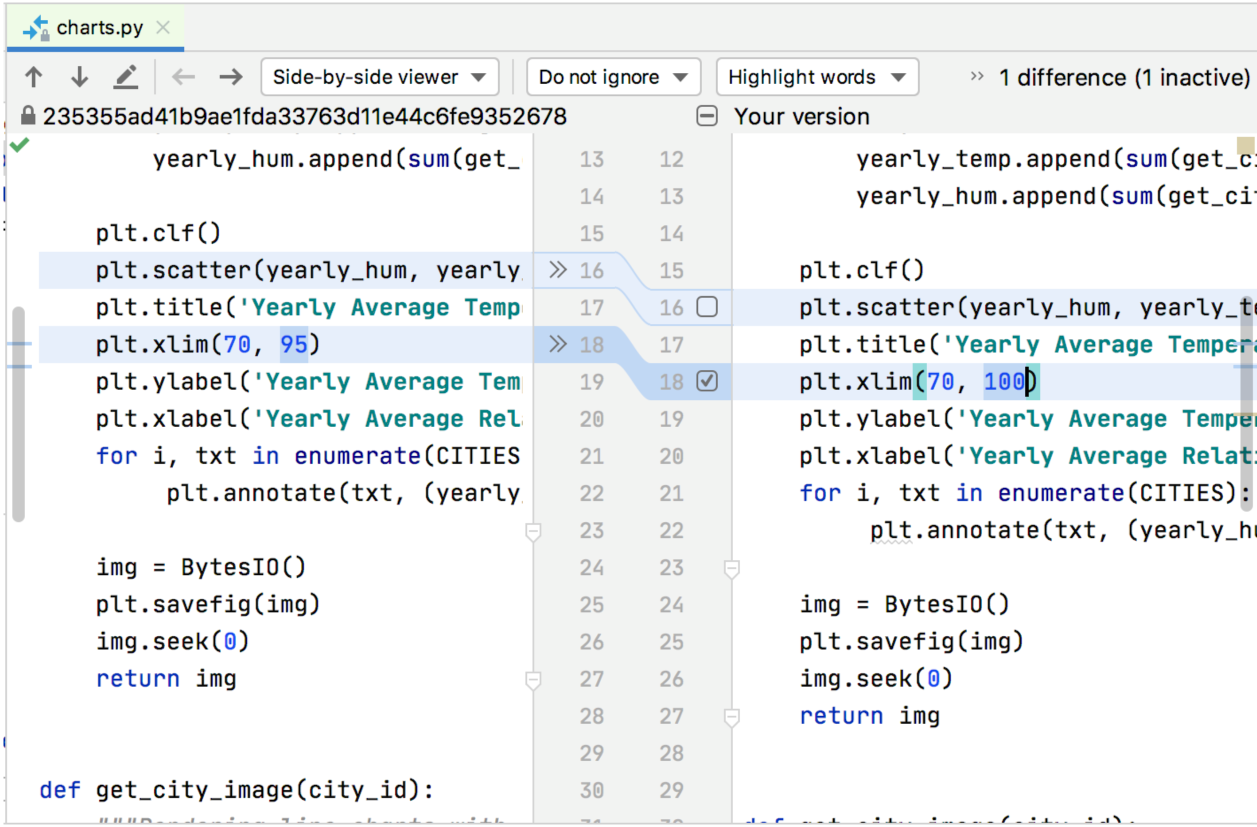
PyCharm: select files to commit in the *Git tool window*



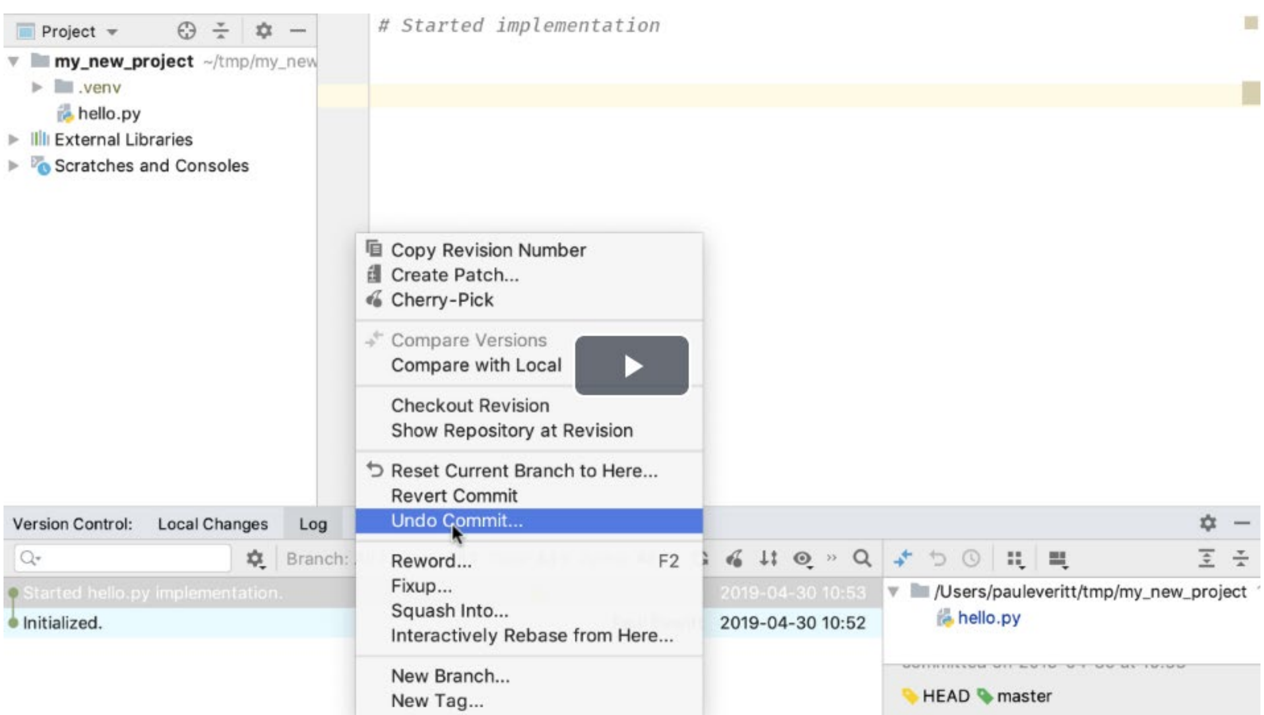
See: https://www.jetbrains.com/help/pycharm/commit-and-push-changes.html?section=Windows%20or%20Linux#

PyCharm: *Partial* commits

Select changes to commit per *chunk*



PyCharm: Undo last commit (if not yet pushed)



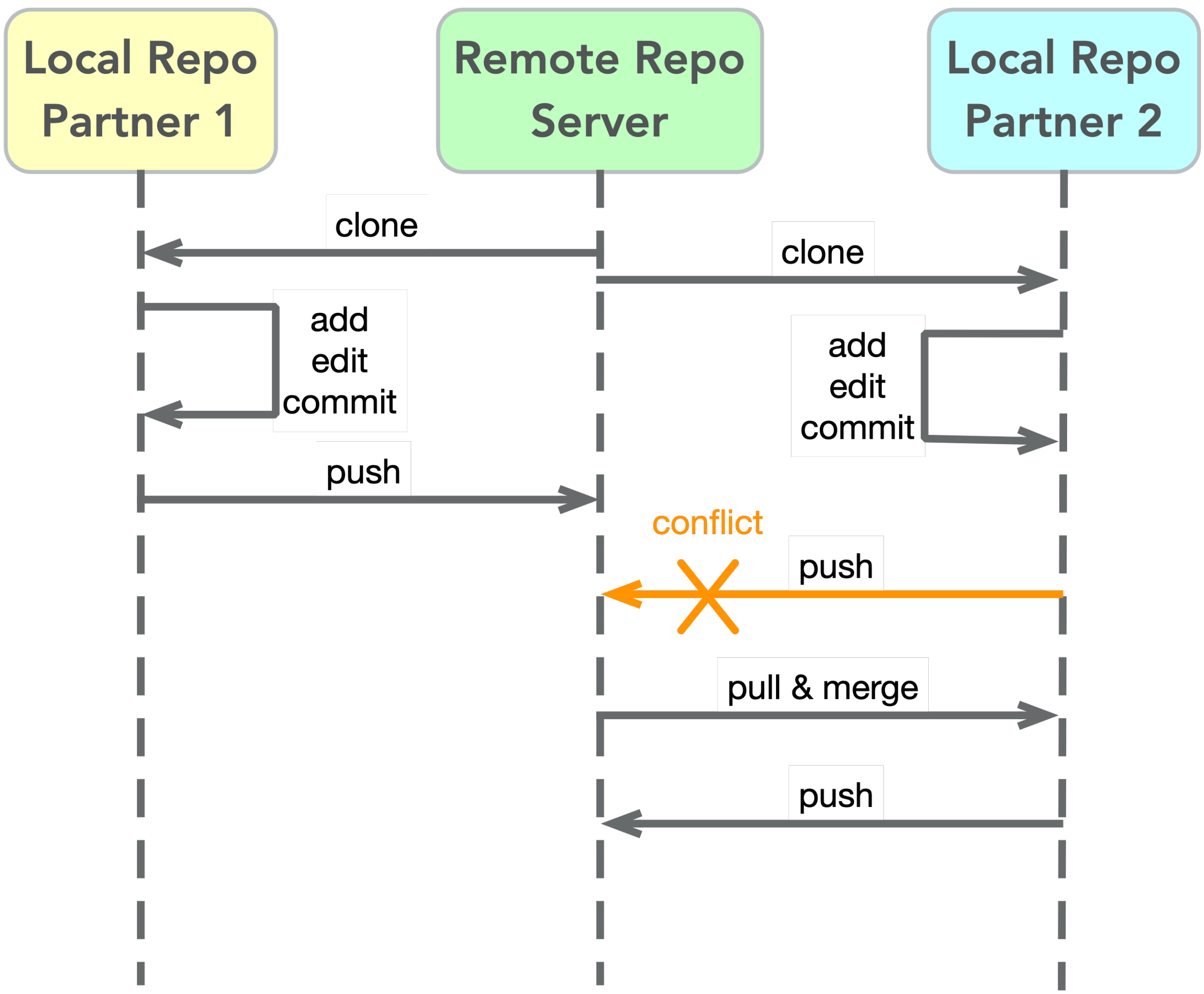
Video: https://www.jetbrains.com/pycharm/guide/tips/undo-last-commit/

Trunk-Based Development (TBD)

There are many Git *workflows*.

We will use *Trunk-Based Development,* without creating new branches.

- One remote repository (on server at GitHub Classroom)
    - With one *branch,* called *trunk* (often called *master* branch)
- Multiple local repositories
    - Repositories: possibly out of sync
    - Push to remote: can fail if out-of-sync
    - Pull from remote: can cause *merge conflicts*



Source: Tom Verhoeff

If push fails

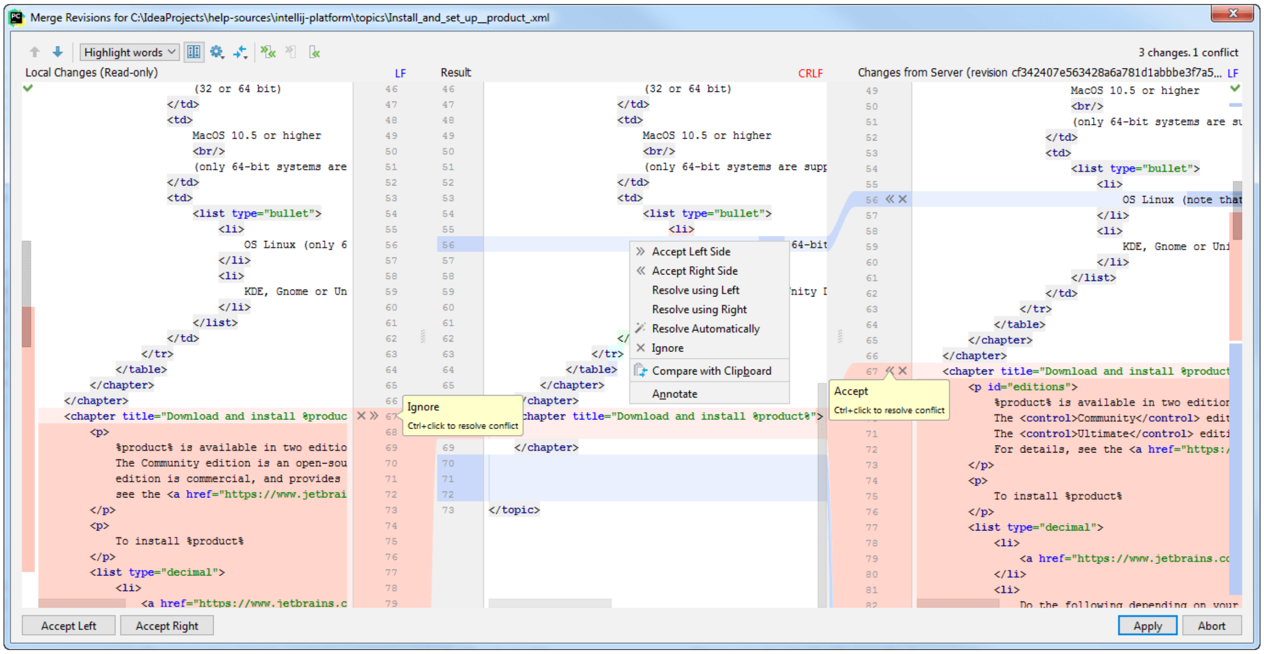
- Reason: Your local repo is out-of-sync with remote repo
- Someone else already pushed new changes that you miss

How to handle:

- **First pull** and resolve merge conflicts
- **Then push** again (fingers crossed)

Advice: Communicate with your co-developers, to avoid this situation

PyCharm: Resolve merge conflicts



See: https://www.jetbrains.com/help/pycharm/resolving-conflicts.html

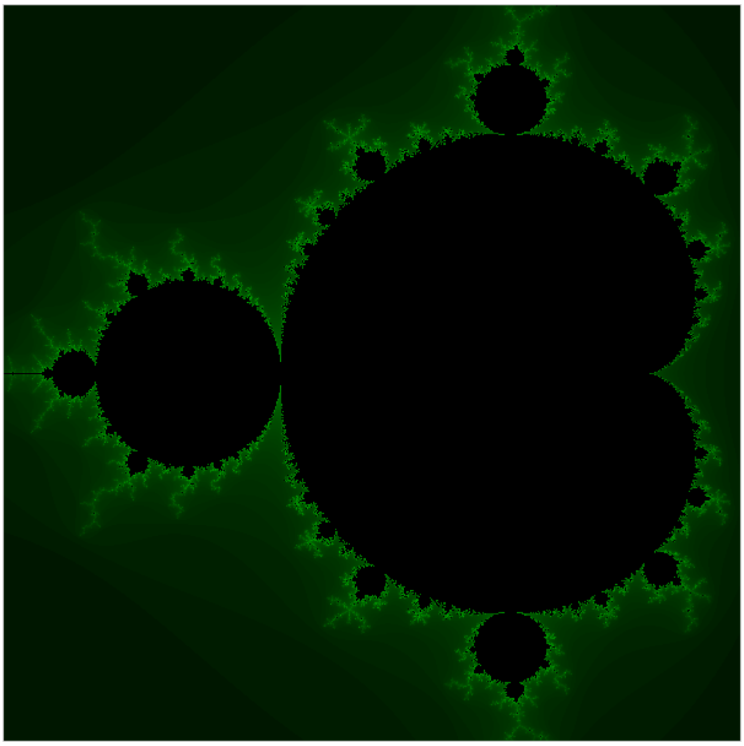
---

(End of Notebook)

In [ ]:

```python
# enable mypy type checking
try:
    %load_ext nb_mypy
except ModuleNotFoundError:
    print("Type checking facility (Nb Mypy) is not installed.")
    print("To use this facility, install Nb Mypy by executing (in a cell):")
    print("  !python3 -m pip install nb_mypy")
```

2IS50 – Software Development for Engineers – 2022-2023



Lecture 3.B (Sw. Eng.)

Lecturer: Lars van den Haak

Review of Lecture 2.B

- Coding Standard:
  - Naming conventions, docstring conventions
- Automated testing
  - `doctest` examples
  - `pytest`
- Test case design
  - Problem coverage, code coverage
- Version control
  - change sets
  - pull/commit/push/conflict
  - Trunk-Based Development (TBD)

"Talented Programmers Don't Tolerate Chaos"

- "There are several defining traits of top programmers, and
  one of the most important of these is that
  they know how to structure things via code."
- "Talented coders want that structure to be as close to perfection as possible."
- "Mediocre coders simply care that the program works."
- "At first, any working program seems like results,
  but poorly coded software is a poor result. And

but poorly coded software is a poor result. And
the goal is not just any results, but instead, top-notch results."

Credits: BLOG@ACM, by Yegor Bugayenko

Clean Code

> *Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.*
> — [John F. Woods]

Preview of Lecture 3.B

- Code duplication and recomputation
  - Don't Repeat Yourself (DRY)
- Object-Oriented-Programming (OOP)
  - An introduction, more details in lecture 4A & 5A
- Graphical User Interface (GUI)
  - An introduction, more details in lecture 5B
- Test-Driven Development (TDD)
- Testing sets and dictionaries (iteration order can vary)
- Issue descriptions & commit messages
  - mentioning issue number & commit hash

In [ ]:

```
from collections import defaultdict, Counter
from typing import Tuple, List, Dict, Set, DefaultDict, Counter, Callable
from typing import Any, Sequence, Iterable, Generator
from math import sqrt
from PyQt5 import QtWidgets
import sys
import doctest
```

Code duplication and recomputation

- Motto: Don't Repeat Yourself (DRY)

Quadratic equation

- Given $a, b, c \in \mathbb{R}$ with $a \neq 0$
- Find all x such that $ax^2 + bx + c = 0$

In [ ]:

```
def solve(a: float, b: float, c: float) -> Set[float]:
    """Compute approximate solutions of a * x ** 2 + b * x + c == 0.

    Assumption: a != 0
    """
    if b ** 2 - 4 * a * c >= 0:
        return {
            (-b + sqrt(b ** 2 - 4 * a * c)) / (2 * a),
            (-b - sqrt(b ** 2 - 4 * a * c)) / (2 * a),
        }
    else:
        return set()
```

In [ ]:

```
%%timeit -c

solve(1, -8, 12)
```

Warning about abc-formula: numerically not reliable

- **What code is (nearly) duplicated?**
- **What is recomputed?**

How to avoid recomputation

- **Store earlier computed result for later reuse**

Trade-offs for recomputation

- **Cost of recomputation: extra time**
- **Cost of avoidance: extra memory**

In [ ]:

```
def solve_1(a: float, b: float, c: float) -> Set[float]:
    """Compute approximate solutions of a * x ** 2 + b * x + c == 0.

    Assumption: a != 0
    """
    discriminant = b ** 2 - 4 * a * c

    if discriminant >= 0:
        return {
            (-b + sqrt(discriminant)) / (2 * a),
            (-b - sqrt(discriminant)) / (2 * a),
        }
    else:
        return set()
```

In [ ]:

```
%%timeit -c

solve_1(1, -8, 12)
```

In [ ]:

```
def solve_2(a: float, b: float, c: float) -> Set[float]:
    """Compute approximate solutions of a * x ** 2 + b * x + c == 0.

    Assumption: a != 0
    """
    discriminant = b ** 2 - 4 * a * c

    if discriminant >= 0:
        s = sqrt(discriminant)
        a2 = 2 * a
        b_neg = -b
        return {
            (b_neg + s) / a2,
            (b_neg - s) / a2,
        }
    else:
        return set()
```

In [ ]:

```
%%timeit -c
```

```
solve_2(1, -8, 12)
```

- $ax^2 + bx + c = 0$
- $x^2 - 2px + q = 0$ with $p = \frac{-b}{2a}$ and $q = \frac{c}{a}$

Have the same solutions (if $a \neq 0$)

In [ ]:

```python
def solve_3(a: float, b: float, c: float) -> Set[float]:
    """Compute approximate solutions of a * x ** 2 + b * x + c == 0.

    Assumption: a != 0
    """
    p, q = -b / (2 * a), c / a
    # a * x ** 2 + b * x + c == 0  <==>  x ** 2 - 2 * p * x + q == 0

    discriminant = p ** 2 - q

    if discriminant >= 0:
        s = sqrt(discriminant)
        return {
            p + s,
            p - s,
        }
    else:
        return set()
```

In [ ]:

```python
%%timeit -c

solve_3(1, -8, 12)
```

Reduced recomputation

- **Cost: 4 more local variables**

- **Still some recomputation:**
  - **if `s == 0`, then `p + s == p - s`**
- **Could avoid this:**

```python
if s:
    return {p + s, p - s}
else:
    return {p}
```

- **Cost: extra condition check (`s != 0`)**
- **Saving: `p+s`, `p-s`, check `p+s == p-s`**

Still some (near) code duplication

- **`p + s` and `p - s` are near duplicates**
- **`return ...` (some set) occurs twice**

How to avoid (near) code duplication

- Auxiliary variable (also avoids recomputation)
- Loop (control variable varies)
- Function (with parameters, to vary)

All add *overhead* (cost time and/or memory)

Trade-offs for (nearly) duplicated code

- Easy to write: copy-paste(-edit)
- Faster (less overhead)
- Can harm *understandability*
  - Is it really (almost) the same?
- Harder to *test*
- Harder to *modify*

  Reasons for future modification:
  - To fix a defect
  - To improve performance
  - To enhance functionality

  Need to modify *all* duplicates, *consistently*

In [ ]:

```
def solve_4(a: float, b: float, c: float) -> Set[float]:
    """Compute approximate solutions of a * x ** 2 + b * x + c == 0.

    Assumption: a != 0
    """
    p, q = -b / (2 * a), c / a
    # a * x ** 2 + b * x + c == 0  <==>  x ** 2 - 2 * p * x + q == 0

    discriminant = p ** 2 - q

    if discriminant >= 0:
        s = sqrt(discriminant)
        result = {p + y for y in (-s, s)}
    else:
        result = set()

    return result
```

In [ ]:

```
%%timeit -c

solve_4(1, -8, 12)
```

- Now, `result = ...` occurs twice
  - Harder to avoid
  - The following is not an improvement (why?)

In [ ]:

```
def solve_5(a: float, b: float, c: float) -> Set[float]:
    """Compute approximate solutions of a * x ** 2 + b * x + c == 0.

    Assumption: a != 0
    """
    p, q = -b / (2 * a), c / a
    # a * x ** 2 + b * x + c == 0  <==>  x ** 2 - 2 * p * x + q == 0

    discriminant = p ** 2 - q
    result = set()  # anticipated

    if discriminant >= 0:
```

```
        s = sqrt(discriminant)
        result = {p + y for y in (-s, s)}

    return result
```

In [ ]:

```
%%timeit -c

solve_5(1, -8, 12)
```

In [ ]:

```
def solve_6(a: float, b: float, c: float) -> Set[float]:
    """Compute approximate solutions of a * x ** 2 + b * x + c == 0.

    Assumption: a != 0
    """
    p, q = -b / (2 * a), c / a
    # a * x ** 2 + b * x + c == 0  <==>  x ** 2 - 2 * p * x + q == 0

    discriminant = p ** 2 - q

    return (
        {p + y for s in [sqrt(discriminant)] for y in (-s, s)}
        if discriminant >= 0
        else set()
    )
```

In [ ]:

```
%%timeit -c

solve_6(1, -8, 12)
```

Maximum Segment Problem

Source: https://www.lotuswritings.nl/wanneer-mag-vlag-uit

- A sequence of houses
- Many have a waving flag, but possibly not all
- What is (the length of) a *longest* slice of houses
    - that all wave the flag?

Modeling the problem

- Sequence of boolean values: `flags: Sequence[bool]`
- E.g. `[True, False, True, True, True, False, False, True]`
- Find int `i` and `j` with `0 <= i <= j <= len(flags)` such that
    - `all(flags[i:j])` and
    - `len(flags[i:j])` (which equals `j - i`) is maximal

In [ ]:

```
FLAGS = [True, False, True, True, True, False, False, True]
```

Naive solution

In [ ]:

```
def max_slice(flags: Sequence[bool]) -> int:
    """Determine length of longest slice of True values in flags."""
```

```
    return max(
        (
            j - i
            for i in range(len(flags))
            for j in range(i, len(flags) + 1)
            if all(flags[i:j])
        ),
        default=0,
    )
```

In [ ]:

```
max_slice(FLAGS)
```

How efficient is this solution?

In [ ]:

```
%%timeit -c  flags = 100 * [True]

max_slice(flags)
```

In [ ]:

```
%%timeit -c  flags = 200 * [True]

max_slice(flags)
```

- **Input k times longer**
- **Runtime $\approx k^3$ times longer: cubic *runtime complexity***
  - **There are three nested loops:** `for i`, `for j`, `all`
- **Each flag is inspected multiple times**
  - **Most** `all` **computations redo a lot of work**
- **How to avoid recomputation?**

Better solution

In [ ]:

```
def max_slice_1(flags: Sequence[bool]) -> int:
    """Determine length of longest slice of True values in flags."""
    k, m, tail = 0, 0, 0
    # invariants:
    #    0 <= k <= len(flags)
    #    m == max_slice(flags[:k])
    #    tail == max(k - i for i in range(k) if all(flags[i:k]))
    #    tail is length of longest True tail in flags[:k]

    while k != len(flags):
        # consider flags[k]
        if flags[k]:
            tail += 1
            if tail > m:
                m = tail
        else:
            tail = 0
        k += 1

    # k == len(flags)
    # hence, m == max_slice(flags)
    return m
```

In [ ]:

```
max_slice_1(FLAGS)
```

In [ ]:

```
%%timeit -c  flags = 100 * [True]

max_slice_1(flags)
```

In [ ]:

```
%%timeit -c  flags = 200 * [True]

max_slice_1(flags)
```

- **Input k times longer**
- **Runtime $\approx k$ times longer: linear *runtime complexity***
  - **There is only one loop**
- **Each flag is inspected once**

In [ ]:

```
def max_slice_2(flags: Sequence[bool]) -> int:
    """Determine length of longest slice of True values in flags."""
    m, tail = 0, 0  # max so far, longest True tail

    for flag in flags:
        if flag:
            tail += 1
            if tail > m:
                m = tail
        else:
            tail = 0

    return m
```

In [ ]:

```
max_slice_2(FLAGS)
```

In [ ]:

```
%%timeit -c  flags = 100 * [True]

max_slice_2(flags)
```

In [ ]:

```
%%timeit -c  flags = 200 * [True]

max_slice_2(flags)
```

Lessons:

- **Measuring can help**
- **Avoiding recomputation can help**

A (small) introduction to:

Object-Oriented Programming (OOP)

A Preview of lecture 4A & 5B

- **In Python, everything (data, code) is manipulated via objects**
- **Every object has a type, which determines**
  - **the kind of values (states) the object can have, and**
  - **the operations it supports**

You have already seen some examples of objects!

- `Counter`, the GUI application in HA_0

Creating and using objects

- An object of type `T` is **created** by calling the **constructor:** `t = T(...)`
  - **E.g.** `bag = Counter('aabc')`
- Objects can have **attributes**, accessed as `t.attribute`
  - **E.g.** `t.__doc__` is the docstring of object `t`
- Function attributes of an object are named **methods:** `t.method(...)`
  - **E.g.** `bag.most_common()`
  - They implicitly take the object itself as first argument

In [ ]:

```
bag: Counter = Counter("Mississippi")
```

In [ ]:

```
print(bag.__doc__)
```

In [ ]:

```
bag.most_common
```

In [ ]:

```
bag.most_common()
```

Creating your own type (`class`)

In [ ]:

```
class MyCounter:
    def __init__(self, items: str) -> None:
        # Make an empty dictionary
        self.counter: Dict[str, int] = dict()
        # Add the items in the update method
        self.update(items)

    def update(self, items: str) -> None:
        # For each item, we add + 1, if no count was known it starts at 0
        for i in items:
            self.counter[i] = self.counter.get(i, 0) + 1
```

- `__init__`: Initialize an object (automatically called after creation/calling the **constructor**)
- Each method needs `self` as (implicit) first argument
  - `self` reflects to the created object
- `self.counter` creates (& refers) to an attribute, where we store data
- `self.update` refers to the method `update` from the `MyCounter` class

In [ ]:

```
my_bag = MyCounter("Mississippi")
my_bag.counter
```

Inheritance (class composition)

When we want to reuse functionality in our own class, we use **inheritance**

- Class composition (see lecture 5A)
- a "is-a" relation
  - A cat is an animal
  - An integer is a number
  - A `MyAwesomeCounter` is a `MyCounter`
- Reuses all attributes and methods
  - But we can **override** them
  - When overriding, can call the **super** class

In [ ]:

```
class MyAwesomeCounter(MyCounter):
    def __init__(self, items: str) -> None:
        # Reuse the init method of the super class
        super().__init__(items)
        # also store the number of items
        self.n: int = len(self.counter.keys())
        # And print some cool message
        print(f"We've created an awesome counter with {self.n} distinct items")
```

In [ ]:

```
my_awesome_bag = MyAwesomeCounter("Mississippi")
print(my_awesome_bag.counter)
my_awesome_bag.n
```

A (small) introduction to:

Graphical User Interface (GUI)

A preview of lecture 5B.

- The interface contains **widgets**
- A widget is an **interactive** object
  - The user can invoke operations using them
  - *Examples*: (radio) buttons, input forms
- In this course we use PyQt5 as GUI framework

GUIs are naturally suited for OOP:

- Many widgets act similar (**inheritance**)
  - E.g. a radio button is similar to a normal button
- After a user interacted, we can store information in an **attribute**
  - E.g. the name the user put into a form

Example GUI (HA_0)

In [ ]:

```
OPTIONS = {0: "Rock", 1: "Paper", 2: "Scissors"}
app = QtWidgets.QApplication(sys.argv)
```

In [ ]:

```
class Application(QtWidgets.QMainWindow):
    def __init__(self) -> None:
        super().__init__()
        self.setWindowTitle("Rock - Paper - Scissors")
        self.resize(320, 320)

        self.main = QtWidgets.QWidget()
```

```
        self.setCentralWidget(self.main)


window = Application()
window.show()
result = app.exec_()
```

In [ ]:

```
class Application(QtWidgets.QMainWindow):
    def __init__(self) -> None:
        super().__init__()
        self.setWindowTitle("Rock - Paper - Scissors")
        self.resize(320, 320)

        self.main = QtWidgets.QWidget()
        self.setCentralWidget(self.main)
        self.main_layout: QtWidgets.QHBoxLayout = QtWidgets.QHBoxLayout()
        self.main.setLayout(self.main_layout)
        self.create_widgets()

    def create_widgets(self) -> None:
        self.rock_button = QtWidgets.QPushButton(OPTIONS[0])
        self.main_layout.addWidget(self.rock_button)
        self.rock_button.clicked.connect(lambda: print("You chose", OPTIONS[0]))  # type
: ignore

        self.paper_button = QtWidgets.QPushButton(OPTIONS[1])
        self.main_layout.addWidget(self.paper_button)
        self.paper_button.clicked.connect(lambda: print("You chose", OPTIONS[1]))  # typ
e: ignore

        self.scissors_button = QtWidgets.QPushButton(OPTIONS[2])
        self.main_layout.addWidget(self.scissors_button)
        self.scissors_button.clicked.connect(lambda: print("You chose", OPTIONS[2]))  #
type: ignore


window = Application()
window.show()
result = app.exec_()
```

Loop to avoid code duplication

In [ ]:

```
class Application(QtWidgets.QMainWindow):
    def __init__(self) -> None:
        super().__init__()
        self.setWindowTitle("Rock - Paper - Scissors")
        self.resize(320, 320)

        self.main = QtWidgets.QWidget()
        self.setCentralWidget(self.main)
        self.main_layout: QtWidgets.QHBoxLayout = QtWidgets.QHBoxLayout()
        self.main.setLayout(self.main_layout)
        self.create_widgets()

    def create_widgets(self) -> None:
        self.buttons = []
        for option, name in OPTIONS.items():
            button = QtWidgets.QPushButton(name)
            button.clicked.connect(lambda: print(f"You chose {name}"))  # type: ignore
            self.main_layout.addWidget(button)
            self.buttons.append(button)


window = Application()
window.show()
result = app.exec_()
```

- **Does not work as expected (test it)**

- `lambda` **binds to** `name` **(3x)**
- **When** `lambda` **is executed, it finds last value of** `name`
- **Solution: bind value of** `name` **to fresh parameter**

In [ ]:

```python
class Application(QtWidgets.QMainWindow):
    def __init__(self) -> None:
        super().__init__()
        self.setWindowTitle("Rock - Paper - Scissors")
        self.resize(320, 320)

        self.main = QtWidgets.QWidget()
        self.main_layout: QtWidgets.QHBoxLayout = QtWidgets.QHBoxLayout()
        self.main.setLayout(self.main_layout)
        self.setCentralWidget(self.main)
        self.create_widgets()

    def create_widgets(self) -> None:
        self.buttons = []
        for option, name in OPTIONS.items():

            def choose(name: str = name) -> Callable[[], None]:
                return lambda: print(f"You chose {name}")

            button = QtWidgets.QPushButton(name)
            button.clicked.connect(choose())  # type: ignore
            self.main_layout.addWidget(button)
            self.buttons.append(button)


window = Application()
window.show()
result = app.exec_()
```

- **Can omit** `self.buttons = []` **and** `self.buttons.append(button)`
  - **Unless the buttons need to be manipulated later**

More information on PyQt5

- ***DelftStack*: Tutorials**

Test-Driven Development

- **Testing is unavoidable**
- **Does it matter *when* you write test code?**
  - ***Before* or *after* writing product code?**

Benefits of thinking about test first

1. **It forces you to consider the *interface***

   - **Which parameters of what types are needed?**
   - **Which results of what types are needed?**
   - **Otherwise, you cannot write down test cases**

1. **It forces you to analyze the problem in advance**

1. It forces you to analyze the problem in advance
    - Delays coding; impatience is a bad guide
    - Test cases with good problem coverage

1. If test cases are ready before writing product code
    - then you can immediately test
    - and fix detected defects
    - without interruption
    - while you are still focused on product code

TDD Steps for Function Definition

1. Understand the problem to be solved
2. Write the docstring
    - Summary sentence
    - Assumptions
    - Details
3. Choose (design) the interface
    - Try to write some `doctest` examples
    - Parameter with type hints
    - Result with type hint
    - Update docstring
4. Analyze the problem further
    - Write more test cases (`doctest` or `pytest`)
5. Write code for function body
6. Run test cases
7. Fix defects

Dealing with later defects

If later you discover another defect, then

1. Add a test case to detect it
2. Fix the code
3. Rerun *all* test cases: called *regression testing*

    - Checks the fix
    - and that it did not break other things

Example: Quadratic equation

- Need test case with *two* solutions: `solve(1, -8, 12)`
- Need test case with *one* solution: `solve(1, 2, 1)`
- Need test case with *no* solutions: `solve(1, 0, -1)`

In [ ]:

```
solve_examples = """
    >>> solve(1, -8, 12)
    {2.0, 6.0}
    >>> solve(1, 2, 1)
    {-1.0}
    >>> solve(1, 0, 1)
    set()
"""
```

In [ ]:

```
doctest.run_docstring_examples(
    solve_examples, globs=globals(), verbose=True, name="solve"
)
```

Complications with `dict` and `set`

- **Printing a `set` need not give reproducible results**
  - **items in `set` have no particular order**
  - **`dict` order is fixed (since Python 3.6)**
- **Order can depend on implementation details of your function**

In [ ]:

```
# Solution 1: compare to expected set

solve_examples_1 = """
    >>> solve(1, -8, 12) == {2.0, 6.0}
    True
    >>> solve(1, 2, 1)
    {-1.0}
    >>> solve(1, 0, 1)
    set()
"""
```

In [ ]:

```
doctest.run_docstring_examples(
    solve_examples_1, globs=globals(), verbose=True, name="solve"
)
```

In [ ]:

```
# Solution 2: sort the set into a list

solve_examples_3 = """
    >>> sorted(solve(1, -8, 12))
    [2.0, 6.0]
    >>> solve(1, 2, 1)
    {-1.0}
    >>> solve(1, 0, 1)
    set()
"""
```

In [ ]:

```
doctest.run_docstring_examples(
    solve_examples_3, globs=globals(), verbose=True, name="solve"
)
```

Example: Maximum Segment Problem

- `flags` **is empty:** `[]`
- `flags` **has length 1:** `[False]`, `[True]`
- `flags` **has maximum at left edge:** `[True, True, False]`
- `flags` **has maximum at right edge:** `[False, True, True]`
- `flags` **has maximum in the middle:** `[False, True, True, False]`
- `flags` **has only** `True` **values:** `[True, True, True]`
- `flags` **has only** `False` **values:** `[False, False, False]`

In [ ]:

```
max_slice_examples = """
    >>> max_slice{variant}([])
```

```
    0
    >>> max_slice{variant}([False])
    0
    >>> max_slice{variant}([True])
    1
    >>> max_slice{variant}([True, True, False])
    2
    >>> max_slice{variant}([False, True, True])
    2
    >>> max_slice{variant}([False, True, True, False])
    2
    >>> max_slice{variant}([True, True, True])
    3
    >>> max_slice{variant}([False, False, False])
    0
"""
```

In [ ]:

```
doctest.run_docstring_examples(
    max_slice_examples.format(variant=""),
    globs=globals(),
    verbose=True,
    name="max_slice",
)
```

In [ ]:

```
doctest.run_docstring_examples(
    max_slice_examples.format(variant="_2"),
    globs=globals(),
    verbose=False,
    name="max_slice_2",
)
```

Example: Sorting a list

- **list is empty:** `[]`
- **list has length 1:** `[1]`
- **list was already sorted:** `[1,3,6,8]`
- **list was sorted in reverse:** `[4,3,2,1]`
- **list contains some duplicates:** `[2,1,1,2]`
- **list contains only duplicates:** `[7,7,7]`
- **list containing negative numbers:** `[-1,1,-5]`

In [ ]:

```
sorted_examples = """
    >>> {sort_function}([])
    []
    >>> {sort_function}([1])
    [1]
    >>> {sort_function}([1,3,6,8])
    [1, 3, 6, 8]
    >>> {sort_function}([4,3,2,1])
    [1, 2, 3, 4]
    >>> {sort_function}([2,1,1,2])
    [1, 1, 2, 2]
    >>> {sort_function}([7,7,7])
    [7, 7, 7]
    >>> {sort_function}([-1,1,-5])
    [-5, -1, 1]
"""
```

In [ ]:

```
doctest.run_docstring_examples(
```

```
doctest.run_docstring_examples(
    sorted_examples.format(sort_function="sorted"),
    globs=globals(),
    verbose=True,
    name="sorted",
)
```

- **How is code coverage?**
- **Is every statement executed under these test cases?**
- **Is every branch taken?**
- **This requires that you analyze the code (not just the problem)**

More information on testing in Python

On *Real Python*:

- [Getting Started With Testing in Python]()
- [Effective Python Testing With Pytest]()

Issues & Commits

1. Issue opened
    - To add a feature
    - To enhance a feature
    - To fix a defect
2. Issue selected to work on
3. Issue discussed with partner(s)
4. Issue assigned
5. Issue analyzed
6. Artifact changed, reviewed/tested, and committed
    - Source, test cases, docs
7. Issue closed

Issue identification

- Each issue has a unique number
- Refer to issue by prefixing its number with `#`
    - in issue descriptions
    - in commit messages
- Turned into a link to that issue

Commit identification

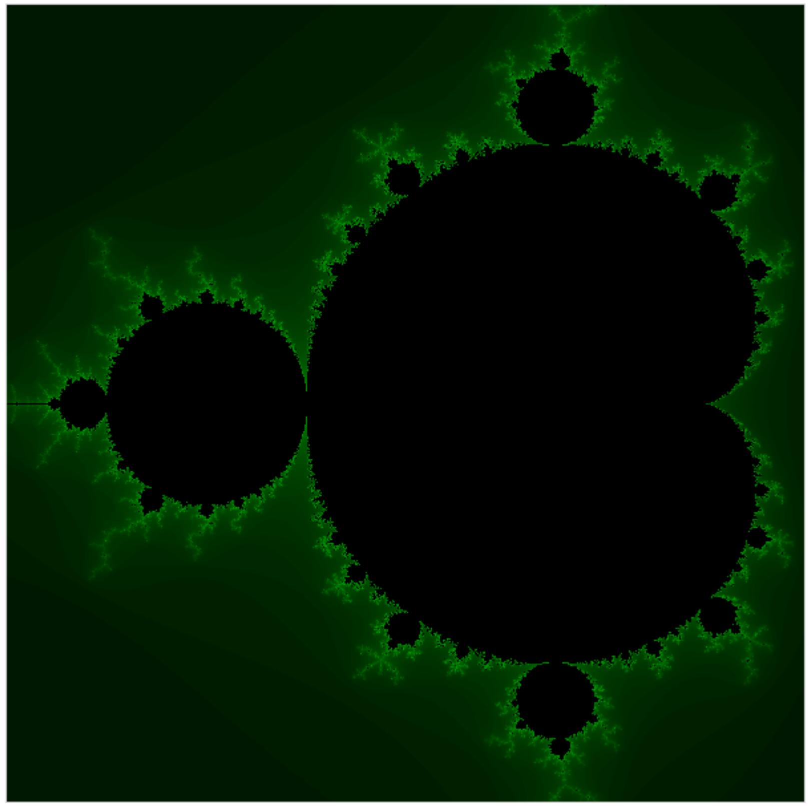
- Each commit has a 'unique' (SHA-1) *hash code*
    - 40 hexadecimal digits
    - often shortened to first 7 hex digits
- Refer to commit by its hash
    - in issue descriptions
    - in commit messages
- Turned into a link to that commit

(End of Notebook)

2IS50 – Software Development for Engineers – 2022-2023

Lecture 4.B (Sw. Eng.)

Lecturer: Tom Verhoeff



Review of Lecture 3.B

- Code duplication and recomputation
  - Don't Repeat Yourself (DRY)
- Test-Driven Development (TDD)
- Testing sets and dictionaries (iteration order can vary)
- Issue descriptions & commit messages
  - mentioning issue number & commit hash

Preview of Lecture 4.B

- Checking type hints
  - Extra type hint features
- Functional decomposition
  - Problem solving: Divide, Conquer & Rule
  - Single Responsibility Principle (SRP)
  - Jargon: *refactoring*
- `pytest` set-up and tear-down

In [1]:
```
%load_ext nb_mypy
# enable mypy type checking
if 'nb_mypy' in get_ipython().magics_manager.magics.get('line'):
    %nb_mypy On
    %nb_mypy
else:
    print("nb-mypy.py not installed")
```

```
Version 1.0.3
State: On DebugOff
```

In [2]:
```
from collections import defaultdict, Counter
from typing import Tuple, List, Dict, Set, DefaultDict, Counter
from typing import Any, Sequence, Mapping, Iterable
from typing import Callable, Generator
import random
import doctest
```

Python Type Hints

- See:
  - typing - Support for type hints
  - Type hints cheat sheet
- For variables
- For function parameters and return values
- Can use built-in types:
  - `str`, `int`, `float`, `bool`
  - `tuple`, `list`, `dict`, `set` (but not recommended)

In [3]:
```
n: int

def f(s: str, b: bool) -> str:
    """..."""
    return s if b else ''
```

For collections prefer capitalized type names, with argument

In [4]:
```
t: Tuple[str, Any] = ('a', True)
names: List[str] = []
d: Dict[str, float] = {}
v: Set[int] = set()
```

In assignments above, type cannot be inferred from expression

Can also use more *generic* type names

- `Sequence`: generalizes `List`, `Tuple`, and `str`
- `Iterable`: anything usable in `for`-loop

`Mapping`, `MutableMapping`: generalizes `Dict`, `DefaultDict`

Type hints in Python:

- Are *voluntary*
- Are *not* checked automatically
- Serve as **documentation**
- Can help **prevent mistakes**

Checking type hints

- Can use **`mypy`** (official type hint checker)
  - http://mypy-lang.org/
  - possibly via `nbQA` (on command line)
- PyCharm:
  - does type checking itself
  - can use Mypy Plugin (needs `mypy`)
- Jupyter Notebook
  - can use our `nb-mypy.py` script (experimental)
  - needs `mypy` and `astor`

In [5]:
```python
n = '42'
n
```

```
<cell>1: error: Incompatible types in assignment (expression has type "str
", variable has type "int")
```

Out[5]: `'42'`

In [6]:
```python
f(3.0, True) + 4
```

```
<cell>1: error: Argument 1 to "f" has incompatible type "float"; expected
"str"
<cell>1: error: Unsupported operand types for + ("str" and "int")
```

Out[6]: `7.0`

In [7]:
```python
def g(n: int) -> str:
    """Return n as string.
    """
    return n
```

```
<cell>4: error: Incompatible return value type (got "int", expected "str")
```

Extra type hint features

- **Type aliases**: different name for same type
- `NewType`: treat existing type as different type
- `TypeVar`: to express type constraints

- `reveal_type`: to find out about inferred types

In [8]:
```
from typing import TypeVar, NewType
```

In [9]:
```
# Type alias
Distribution = Sequence[float]

def sample(distr: Distribution, k: int = 1) -> Sequence[int]:
    """Return sample of size k according to distribution, with replacement
.

    Assumption: k >= 0
    """
    return random.choices(list(range(len(distr))), distr, k=k)

sample([0.3, 0.7], 20)
```

Out[9]:
```
[1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1]
```

In [10]:
```
# New type name (not just an alias!)
Distance = NewType('Distance', float)
Area = NewType('Area', float)

def scale(factor: float, dist: Distance) -> Distance:
    return factor * dist  # error with types
```

```
<cell>6: error: Incompatible return value type (got "float", expected "Dis
tance")
```

In [11]:
```
def scale(factor: float, dist: Distance) -> Distance:
    return Distance(factor * dist)  # error with types fixed
```

In [12]:
```
a = Area(100)

scale(10, a)
```

```
<cell>3: error: Argument 2 to "scale" has incompatible type "Area"; expect
ed "Distance"
```

Out[12]:
```
1000
```

In [13]:
```
# Type variable
T = TypeVar('T')

def mid(seq: Sequence[T]) -> T:
    """Return item from seq near the middle.

    Assumption: seq is not empty
    """
    return seq[len(seq) // 2]
```

This is more informative than

```
def mid(seq: Sequence[Any]) -> Any
```

In [14]:
```python
# reveal_type is not defined, but interpreted by mypy
reveal_type(mid([1, 2]))
reveal_type(mid(['a', 'b']))
```

```
<cell>2: note: Revealed type is "builtins.int"
<cell>3: note: Revealed type is "builtins.str"
```

Advanced type hints

- `Optional` : if value can also be `None`
- `Union` : if value can have multiple types

In [15]:
```python
from typing import Optional, Union
```

In [16]:
```python
result: Optional[int] = None

answer: Union[str, int, float, bool] = "Don't know"
```

Functional Decomposition

- **Monolith**: "a large single upright block of stone"
  - Greek: *monos* ('single') + *lithos* ('stone')
- **Monolithic**: "formed of a single block of stone"
  - (*subtractive* manufacturing)
- **Monolithic program**: one large block of code, without function definitions
- **Functional decomposition**: express computation as *composition of functions*
  - (*additive* manufacturing)

We start from some monolithic code to illustrate functional decomposition

Research question

> Suppose you know that your opponent chooses Rock-Paper-Scissors with probabilities r, p, s respectively ($0 \le r, p, s \le 1$ and $r+p+s=1$).
>
> What would be your best strategy?

Rather than do some math, we approach this by simulation. Just try.

The following program starts with given probabilities `r` , `p` , `s` , and tries each option for you one thousand times (always choosing that same option), keeping track of win-lose statistics. Afterwards, the best option is determined. (My guess is that the best choice should beat the highest probability. So, this is also computed.)

To see the relevance of the order of the probabilities, we try all six arrangements (outer loop).

Monolithic program

In [17]:
```python
r, p, s = 0.1, 0.4, 0.5  # probabilities of random-playing opponent
swap_left = True  # which pair to swap next: left vs. right

for k in range(6):
    print(f"Opponent's probability distribution: {r:1.2f}, {p:1.2f}, {s:1.2f}")
    wins = 3 * [0]  # initialize win counts for all options
    # Try each of my choices

    for choice_me in range(3):  # rock, paper, scissors
        print(f"My choice: {choice_me}")
        # Play one thousand games, and gather statistics

        for i in range(1000):
            choice_opponent = choice_me  # to start the loop

            while choice_me == choice_opponent:
                choice_opponent = random.choices([0, 1, 2], weights=[r, p, s], k=1)[0]

            if (choice_me - choice_opponent) % 3 == 1:
                wins[choice_me] += 1

        print(f"  win - lose: {wins[choice_me]} - {1000 - wins[choice_me]}")

    # determine my best choice (argmax)
    best_choice = max(range(3), key=lambda x: wins[x])
    print(f"My best choice: {best_choice}")

    # determine what beats highest probability (argmax, again)
    guessed_choice = (max(range(3), key=lambda x: [r, p, s][x]) + 1) % 3
    print(f"Guessed choice: {guessed_choice}", end='\n\n')

    if swap_left:
        r, p = p, r
    else:
        p, s = s, p
    swap_left = not swap_left
```

```
Opponent's probability distribution: 0.10, 0.40, 0.50
My choice: 0
  win - lose: 561 - 439
My choice: 1
  win - lose: 169 - 831
My choice: 2
  win - lose: 810 - 190
My best choice: 2
Guessed choice: 0

Opponent's probability distribution: 0.40, 0.10, 0.50
My choice: 0
  win - lose: 819 - 181
```

```
My choice: 1
  win - lose: 446 - 554
My choice: 2
  win - lose: 197 - 803
My best choice: 0
Guessed choice: 0

Opponent's probability distribution: 0.40, 0.50, 0.10
My choice: 0
  win - lose: 180 - 820
My choice: 1
  win - lose: 803 - 197
My choice: 2
  win - lose: 576 - 424
My best choice: 1
Guessed choice: 2

Opponent's probability distribution: 0.50, 0.40, 0.10
My choice: 0
  win - lose: 202 - 798
My choice: 1
  win - lose: 820 - 180
My choice: 2
  win - lose: 441 - 559
My best choice: 1
Guessed choice: 1

Opponent's probability distribution: 0.50, 0.10, 0.40
My choice: 0
  win - lose: 820 - 180
My choice: 1
  win - lose: 538 - 462
My choice: 2
  win - lose: 177 - 823
My best choice: 0
Guessed choice: 1

Opponent's probability distribution: 0.10, 0.50, 0.40
My choice: 0
  win - lose: 429 - 571
My choice: 1
  win - lose: 199 - 801
My choice: 2
  win - lose: 846 - 154
My best choice: 2
Guessed choice: 2
```

Code analysis

- *Magic literal constants*: `3, 6, 1000, [0, 1, 2]`
  - Name them
- Separate variables for probabilities: `r, p, s`
  - Combine them into a list or dictionary
- Everything is *entangled*

    - Decompose (refactor), using functions
- Traversing all permutations
    - Use `itertools.permutations`

Note (not about code, but about this problem)

- The inner `while` loop is dangerous
    - could never end, depending on choice of `r, p, s`
- The inner while loop is not needed
    - record (and ignore) ties separately, and also losses

Refactored code

Introduce (problem-specific or general)

- Type names
- Named Constants
- Named Functions

In [18]:
```
#: Type for choice options
#: Constraint: only 0, 1, 2 used
Option = NewType('Option', int)

#: Constants
ROCK, PAPER, SCISSORS = Option(0), Option(1), Option(2)
OPTIONS: List[Option] = [ROCK, PAPER, SCISSORS]
```

In [19]:
```
#: Type for outcomes of an RPS encounter
#: 0 (tie), 1 (Player 1), or 2 (Player 2)
Outcome = NewType('Outcome', int)

#: Encoding of tie outcome
TIE = Outcome(0)

def judge_encounter(choice_1: Option, choice_2: Option) -> Outcome:
    """Judge an RPS encounter, returning who wins: Player 1 or 2 (TIE for tie).

    >>> judge_encounter(ROCK, ROCK)
    0
    >>> judge_encounter(PAPER, SCISSORS)
    2
    >>> judge_encounter(ROCK, SCISSORS)
    1
    """
    return Outcome((choice_1 - choice_2) % len(OPTIONS))
```

In [20]:
```
doctest.run_docstring_examples(judge_encounter, globs=globals(), name='judge_encounter')
```

In [21]:
```
#: Type for probability distribution on OPTIONS
#: Assumptions for distr: Distribution
#:
#:    * all(0 <= p <= 1 for p in distr)
#:    * sum(distr) == 1
#:    * len(distr) == len(OPTIONS)
Distribution = Sequence[float]
```

In [22]:
```
def choose_random(distr: Distribution) -> Option:
    """Make random choice according to given distribution.

    >>> choose_random([1, 0, 0])
    0
    >>> choose_random([0, 1, 0])
    1
    >>> choose_random([0, 0, 1])
    2
    >>> all(choose_random([1/3, 1/3, 1/3]) in OPTIONS for _ in range(100))
    True
    """
    return Option(random.choices(OPTIONS, weights=distr, k=1)[0])
```

In [23]:
```
doctest.run_docstring_examples(choose_random, globs=globals(), name='choos
e_random')
```

In [24]:
```
def play_my_game(choice_1: Option, distr_2: Distribution) -> Outcome:
    """Play an RPS game, returning who wins: Player 1 or 2.

    Player 1 always chooses choice_1.
    Player 2 chooses according to given distribution

    Note: This could lead to infinite loop!

    >>> play_my_game(0, [0, 1, 0])
    2
    >>> play_my_game(0, [0, 0, 1])
    1
    >>> all(play_my_game(0, [1/3, 1/3, 1/3]) in [1, 2] for _ in range(100)
)
    True
    """
    result = TIE  # prime the loop

    while result == TIE:
        result = judge_encounter(choice_1, choose_random(distr_2))

    # result != TIE
    return result
```

In [25]:
```
doctest.run_docstring_examples(play_my_game, globs=globals(), name='play_m
y_game')
```

Can be generalized:

- Provide two choice functions as arguments

- Better testable
- (Later: even better object-oriented solution)

In [26]:
```
#: Function without arguments that chooses among OPTIONS
ChoiceFunction = Callable[[], Option]

def play_game(choice_1: ChoiceFunction, choice_2: ChoiceFunction) -> Outco
me:
    """Play an RPS game, returning who wins: Player 1 or 2.

    Note: This could lead to infinite loop!

    >>> play_game(lambda: 0, lambda: 1)
    2
    >>> play_game(lambda: 0, lambda: 2)
    1
    """
    result = TIE  # prime the loop

    while result == TIE:
        result = judge_encounter(choice_1(), choice_2())

    # result != TIE
    return result
```

In [27]:
```
doctest.run_docstring_examples(play_game, globs=globals(), name='play_game
')
```

In [28]:
```
def play_my_game(choice_1: Option, distr_2: Distribution) -> int:
    """Play an RPS game, returning who wins: Player 1 or 2.

    Player 1 always chooses choice_1.
    Player 2 chooses according to given distribution

    Note: This could lead to infinite loop!

    >>> play_my_game(ROCK, [0, 1, 0])
    2
    >>> play_my_game(ROCK, [0, 0, 1])
    1
    >>> all(play_my_game(ROCK, [1/3, 1/3, 1/3]) in [1, 2] for _ in range(1
00))
    True
    """
    return play_game(lambda: choice_1, lambda: choose_random(distr_2))
```

In [29]:
```
doctest.run_docstring_examples(play_my_game, globs=globals(), name='play_m
y_game')
```

In [30]:
```
def play_games(n: int, choice_1: ChoiceFunction, choice_2: ChoiceFunction)
 -> int:
    """Play n games, returning number of wins for Player 1.

    Assumptions:
```

```
    * n >= 0

    >>> play_games(-1, lambda: ROCK, lambda: ROCK)
    Traceback (most recent call last):
        ...
    AssertionError: n must be >= 0
    >>> play_games(0, lambda: ROCK, lambda: ROCK)
    0
    >>> play_games(1, lambda: ROCK, lambda: PAPER)
    0
    >>> play_games(2, lambda: ROCK, lambda: SCISSORS)
    2
    """
    assert n >= 0, "n must be >= 0"
    result = 0  # number of wins for Player 1

    for _ in range(n):
        outcome = play_game(choice_1, choice_2)
        if outcome == 1:
            result += 1
        # alternative
        # result += outcome % 2

    return result
```

In [31]:
```
doctest.run_docstring_examples(play_games, globs=globals(), name='play_gam
es')
```

In [32]:
```
def play_all_my_games(n: int, distr_2: Distribution) -> None:
    """Play n games for each option and print summary statistics, and
    actual and guessed best choice.

    Assumptions:

    * n >= 0
    """
    print("Opponent's probability distribution: {:1.2f}, {:1.2f}, {:1.2f}"
.format(*distr_2))
    wins = len(OPTIONS) * [0]  # initialize win counts for all options
    # Try each of my choices

    for choice_me in OPTIONS:
        print(f"My choice: {choice_me}")
        wins[choice_me] = play_games(n, lambda: choice_me, lambda: choose_
random(distr_2))
        print(f"  win - lose: {wins[choice_me]} - {n - wins[choice_me]}")

    # determine my best choice (argmax)
    best_choice = max(OPTIONS, key=lambda x: wins[x])
    print(f"My best choice: {best_choice}")

    # determine what beats highest probability (argmax, again)
    guessed_choice = (max(OPTIONS, key=lambda x: distr_2[x]) + 1) % len(OP
TIONS)
    print(f"Guessed choice: {guessed_choice}")
```

Harder to test automatically!

Let's run a manual test case (*smoke test*)

In [33]:
```
play_all_my_games(10, [1/3, 1/3, 1/3])
```

```
Opponent's probability distribution: 0.33, 0.33, 0.33
My choice: 0
  win - lose: 3 - 7
My choice: 1
  win - lose: 7 - 3
My choice: 2
  win - lose: 5 - 5
My best choice: 1
Guessed choice: 1
```

Now go through all permutations

In [34]:
```
import itertools as it
```

In [35]:
```
for distr in it.permutations([0.1, 0.4, 0.5]):
    play_all_my_games(1000, distr)
    print()
```

```
Opponent's probability distribution: 0.10, 0.40, 0.50
My choice: 0
  win - lose: 556 - 444
My choice: 1
  win - lose: 181 - 819
My choice: 2
  win - lose: 815 - 185
My best choice: 2
Guessed choice: 0

Opponent's probability distribution: 0.10, 0.50, 0.40
My choice: 0
  win - lose: 471 - 529
My choice: 1
  win - lose: 195 - 805
My choice: 2
  win - lose: 838 - 162
My best choice: 2
Guessed choice: 2

Opponent's probability distribution: 0.40, 0.10, 0.50
My choice: 0
  win - lose: 829 - 171
My choice: 1
  win - lose: 430 - 570
My choice: 2
  win - lose: 224 - 776
My best choice: 0
Guessed choice: 0
```

```
Opponent's probability distribution: 0.40, 0.50, 0.10
My choice: 0
  win - lose: 150 - 850
My choice: 1
  win - lose: 802 - 198
My choice: 2
  win - lose: 574 - 426
My best choice: 1
Guessed choice: 2

Opponent's probability distribution: 0.50, 0.10, 0.40
My choice: 0
  win - lose: 793 - 207
My choice: 1
  win - lose: 551 - 449
My choice: 2
  win - lose: 167 - 833
My best choice: 0
Guessed choice: 1

Opponent's probability distribution: 0.50, 0.40, 0.10
My choice: 0
  win - lose: 184 - 816
My choice: 1
  win - lose: 856 - 144
My choice: 2
  win - lose: 419 - 581
My best choice: 1
Guessed choice: 1
```

Decomposition trade-offs

- How to decide on decomposition?
- How many functions are needed?
- How small/big should functions be?
- With what parameters and what result?

Disadvantages of using (many) functions:

- Functions bring *execution overhead*
- Functions need names; parameters need names and types
- Functions need *documentation*
- Functions need *testing*

Advantages of using functions:

- easier to understand and reason about
- easier to get to work
- easier to test (avoids most debugging)
- easier to modify (*locality of change*)

easier to (re-)use code in same or other programs
  - code completion, built-in documentation

Decomposition guidelines

- View *functional decomposition* as **problem solving** technique

  1. **Divide**: Subdivide big problem into smaller problems
  2. **Conquer**: Solve subproblems
  3. **Rule**: Combine solutions to subproblems into solution to big problem

- Each function should serve a *single purpose*
  - **Single Responsibility Principle**

- For each function, you should be able to provide
  - docstring
  - test cases

- Make functions *general*
  - through *parameters*
  - with *generic types* (e.g. prefer `Sequence` over `List`)
  - avoid using *global variables*

- Consider and compare alternative decompositions

Test case set-up and tear-down

- When multiple test cases need the same data:
  - In some data structure
  - In a file
  - On a web site
  - In a data base
- How to avoid code duplication?
- **Set-up** code: arranges access to the data
- **Tear-down** code: closes access properly

In [36]:
```
class Card:
    """A mutable card with an up and down side (non-empty strings).
    """

    def __init__(self, up: str, down: str):
        """Create card with given state.
        """
        assert up and down, "up and down must not be empty"
```

```python
        self.up = up
        self.down = down

    def __repr__(self) -> str:
        return f"Card({self.up!r}, {self.down!r})"

    def __str__(self) -> str:
        return f"{self.up} ({self.down})"

    def flip(self) -> None:
        """Flip over this card.

        Modifies: self
        """
        self.up, self.down = self.down, self.up
```

Use `pytest` in Jupyter Notebook (NOT NEEDED FOR FINAL TEST)

- Install `ipytest`:

```
$ pip3 install ipytest
```

- Import and configure `ipytest` (see below)
- Use *cell magic* `%%ipytest` to run test cases
  - you can pass `pytest` command-line options

In [37]:

```python
import ipytest
ipytest.autoconfig()
```

In [38]:

```python
%%ipytest -vv
# -vv: extra verbose mode

import pytest

def test_constructor_assert():
    with pytest.raises(AssertionError):
        card = Card('', 'O')

def test_constructor_attributes():
    card = Card('#', 'O')
    assert card.up == '#'
    assert card.down == 'O'

def test_repr():
    card = Card('#', 'O')
    assert repr(card) == "Card('#', 'O')"

def test_str():
    card = Card('#', 'O')
    assert str(card) == "# (O)"

def test_flip():
    card = Card('#', 'O')
```

```
    card.flip()
    assert card.up == 'O'
    assert card.down == '#'
```

```
======================================= test session starts ==============
==========================
platform darwin -- Python 3.9.7, pytest-6.2.4, py-1.10.0, pluggy-0.13.1 --
 /Users/wstomv/opt/anaconda3/bin/python
cachedir: .pytest_cache
rootdir: /Users/wstomv/Documents/Education/2IS50 Software Development for
Engineers/Year 2022-2023/study-material-2is50-2022-2023/lectures/year 2022
-2023
plugins: anyio-2.2.0
collecting ... collected 5 items

tmp04q8vzlw.py::test_constructor_assert PASSED
                   [ 20%]
tmp04q8vzlw.py::test_constructor_attributes PASSED
                   [ 40%]
tmp04q8vzlw.py::test_repr PASSED
                   [ 60%]
tmp04q8vzlw.py::test_str PASSED
                   [ 80%]
tmp04q8vzlw.py::test_flip PASSED
                   [100%]

======================================== 5 passed in 0.02s ===============
==========================
```

`pytest` Test Fixture for Set-up

Uses *function decorator* `@pytest.fixture`

In [39]:

```
%%ipytest -qq -s
# -qq: extra quiet mode
# -s: don't capture printed output

import pytest

@pytest.fixture
def card_ut():
    """Set up the card under test.
    """
    card = Card('#', 'O')
    print(f"\nSet up {card!r}", end='')
    return card

def test_constructor_assert():
    with pytest.raises(AssertionError):
        card = Card('', 'O')

def test_constructor_attributes(card_ut):
    assert card_ut.up == '#'
    assert card_ut.down == 'O'
```

```
def test_repr(card_ut):
    assert repr(card_ut) == "Card('#', 'O')"

def test_str(card_ut):
    assert str(card_ut) == "# (O)"

def test_flip(card_ut):
    card_ut.flip()
    assert card_ut.up == 'O'
    assert card_ut.down == '#'
```

```
.
Set up Card('#', 'O').
Set up Card('#', 'O').
Set up Card('#', 'O').
Set up Card('#', 'O').
```

In *quiet* mode (option `-q` or `-qq`)

- `.` means test case *passed*
- `F` means test case *failed* or contains *error*

In [40]:

```
%%ipytest -qq

def test_pass():
    assert True

def test_failure():
    assert False
```

```
.F
                         [100%]
============================================= FAILURES ===================
==========================
__________________________________________ test_failure _________________
__________________________

    def test_failure():
>       assert False
E       assert False

/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_13543/268591915
9.py:5: AssertionError
===================================== short test summary info ============
==========================
FAILED tmpubo9ca6y.py::test_failure - assert False
```

`pytest` Test Fixture for Tear-down

- After test cases using the same data
  - Clear the data structure
  - Close the file
  - Close connection to web site

Close connection to data base

In [41]:
```
%%ipytest -qq -s

import pytest

@pytest.fixture
def card_ut():
    """Set up the card under test.
    """
    card = Card('#', 'O')  # set up resource
    print(f"\nSet up {card!r}", end='')
    yield card  # make resource available
    print(f"Tear down {card!r}", end='')  # tear down resource

def test_constructor_assert():
    with pytest.raises(AssertionError):
        card = Card('', 'O')

def test_constructor_attributes(card_ut):
    assert card_ut.up == '#'
    assert card_ut.down == 'O'

def test_repr(card_ut):
    assert repr(card_ut) == "Card('#', 'O')"

def test_str(card_ut):
    assert str(card_ut) == "# (O)"

def test_flip(card_ut):
    card_ut.flip()
    assert card_ut.up == 'O'
    assert card_ut.down == '#'
```

```
.
Set up Card('#', 'O').Tear down Card('#', 'O')
Set up Card('#', 'O').Tear down Card('#', 'O')
Set up Card('#', 'O').Tear down Card('#', 'O')
Set up Card('#', 'O').Tear down Card('O', '#')
```

More information on testing in Python

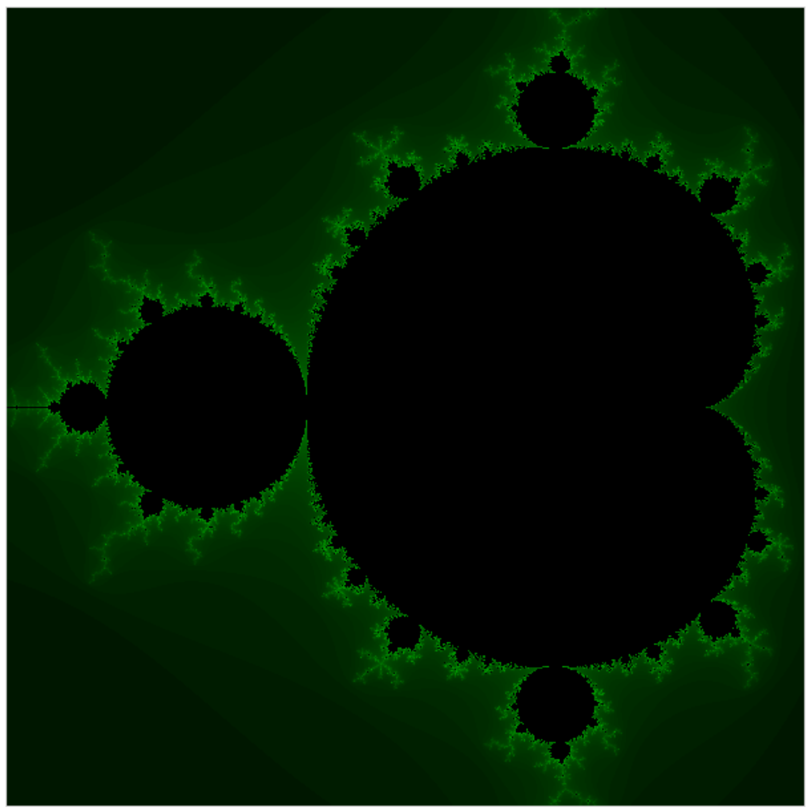
- [Getting Started With Testing in Python]
- [Effective Python Testing With Pytest]
- [Pytest documentation]

(End of Notebook)

In [1]:
```python
# enable mypy type checking
try:
    %load_ext nb_mypy
except ModuleNotFoundError:
    print("Type checking facility (Nb Mypy) is not installed.")
    print("To use this facility, install Nb Mypy by executing (in a cell):
")
    print("  !python3 -m pip install nb_mypy")
```

```
Version 1.0.3
```

2IS50 – Software Development for Engineers – 2022-2023



Lecture 5.B (Sw. Eng.)

Lecturer: Lars van den Haak

Review of Lecture 4.B

- Checking type hints
  - Extra type hint features
- Functional decomposition
  - Problem solving: Divide, Conquer & Rule
  - Single Responsibility Principle (SRP)
  - Jargon: *refactoring*
- `pytest` set-up and tear-down

Preview of Lecture 5.B

- Using exceptions: EAFP versus LBYL
- Sphinx documentation
  - reStructuredText (reST, RST)
- Interface design
  - Application Programming Interface (API)
  - Command-Line Interface (CLI)
  - Graphical User Interface (GUI), `PyQt5`
- Data decomposition

In [2]:
```
from collections import defaultdict, Counter
from typing import Tuple, List, Dict, Set, DefaultDict, Counter
from typing import Any, Sequence, Mapping, Iterable
from typing import Callable, Iterator, Generator
import math
import doctest
```

Using Exceptions

Two sides:

- *Inside* function being called:
  - `raise` exception
  - to signal exceptional situation
  - when 'normal' response is not possible/appropriate
  - N.B. exception cannot be accidentally overlooked

- *Outside* function when calling it:
  - let execution abort, or
  - *catch* and *handle* exception

Two Styles: LBYL and EAFP

- **Look Before You Leap** (LBYL)
  - Check assumptions before call (with `if`)
  - Only call if assumptions hold
  - Avoid triggering of exceptions

- **Easier to Ask for Forgiveness than Permission** (EAFP)
  - Just make the call (with `try`)
  - knowing that you'll be 'forgiven', if assumptions don't hold
  - i.e., no hard disk wiped, but exception raised

See: Python Glossary

Intermezzo: Grace Murray Hopper

- Rear Admiral *Grace Murray Hopper*: early programmer
- Introduced the term *bug* for computer/software defect
- "It's easier to ask for forgiveness than it is to get permission"
  - "If it's a good idea, go ahead and do it.
    It is much easier to apologize than it is to get permission."
- "Life was simple before World War II. After that, we had systems."
- ACM Grace Murray Hopper Award
  - given to the outstanding young computer professional of the year

Image source: https://commons.wikimedia.org/wiki/Category:Grace_Hopper#/media/File:Grace_Hopper.jpg

Two Examples

In [3]:
```
# LBYL

x = -1.0   # float computed earlier

if x >= 0:   # Look
    print(math.sqrt(x))   # Leap
else:
    print("x is negative")
```

```
x is negative
```

In [4]:
```
# EAFP

x = -1.0   # float computed ealier

try:
    print(math.sqrt(x))   # ask for forgiveness
except ValueError:
    print("x is invalid argument for math.sqrt()")
```

```
x is invalid argument for math.sqrt()
```

In [5]:
```python
# EAFP (better: less code in try)
from typing import Optional

x = -1.0  # float computed ealier
root: Optional[float]

try:
    root = math.sqrt(x)
except ValueError:
    root = None
    print("x is invalid argument for math.sqrt()")
else:
    print(root)
```

```
x is invalid argument for math.sqrt()
```

In [6]:
```python
# LBYL

user_input = input("Give me float x: ")

if isfloat(user_input):  # Look (not defined; hard)
    print(f"x squared is {float(user_input) ** 2}")  # Leap
else:
    print("x must be a float")
```

```
<cell>5: error: Name "isfloat" is not defined
```

```
---------------------------------------------------------------------------
-
StdinNotImplementedError                  Traceback (most recent call last
)
/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_40316/391508567
4.py in <module>
      1 # LBYL
      2
----> 3 user_input = input("Give me float x: ")
      4
      5 if isfloat(user_input):  # Look (not defined; hard)

~/opt/anaconda3/lib/python3.9/site-packages/ipykernel/kernelbase.py in raw
_input(self, prompt)
   1001         """
   1002         if not self._allow_stdin:
-> 1003             raise StdinNotImplementedError(
   1004                 "raw_input was called, but this frontend does not
support input requests."
   1005             )

StdinNotImplementedError: raw_input was called, but this frontend does not
 support input requests.
```

In [7]:
```python
# EAFP

user_input = input("Give me float x: ")
```

```
x: Optional[float]

try:
    x = float(user_input)  # Ask for forgiveness
except ValueError:
    x = None
    print("x must be a float")
else:
    print(f"x squared is {x ** 2}")
```

```
---------------------------------------------------------------------------
StdinNotImplementedError                  Traceback (most recent call last
)
/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_40316/415304165
.py in <module>
      1 # EAFP
      2 
----> 3 user_input = input("Give me float x: ")
      4 x: Optional[float]
      5 

~/opt/anaconda3/lib/python3.9/site-packages/ipykernel/kernelbase.py in raw
_input(self, prompt)
   1001         """
   1002         if not self._allow_stdin:
-> 1003             raise StdinNotImplementedError(
   1004                 "raw_input was called, but this frontend does not
support input requests."
   1005             )

StdinNotImplementedError: raw_input was called, but this frontend does not
 support input requests.
```

Checking whether a string can be converted to `float`

- is hard (https://stackoverflow.com/questions/736043/checking-if-a-string-can-be-converted-to-float-in-python)

Command to parse	Is it a float?	Comment
`print(isfloat(""))`	`False`	
`print(isfloat("1234567"))`	`True`	
`print(isfloat("NaN"))`	`True`	nan is also float
`print(isfloat("NaNananana BATMAN"))`	`False`	
`print(isfloat("123.456"))`	`True`	
`print(isfloat("123.E4"))`	`True`	
`print(isfloat(".1"))`	`True`	
`print(isfloat("1,234"))`	`False`	
`print(isfloat("NULL"))`	`False`	Case insensitive
`print(isfloat(",1"))`	`False`	
`print(isfloat("123.EE4"))`	`False`	

`print(isfloat("6.523537535629999e-07"))`	`True`	
`print(isfloat("6e777777"))`	`True`	This is same as Inf
`print(isfloat("-iNF"))`	`True`	
`print(isfloat("1.797693e+308"))`	`True`	
`print(isfloat("infinity"))`	`True`	
`print(isfloat("infinity and BEYOND"))`	`False`	
`print(isfloat("12.34.56"))`	`False`	Two dots not allowed
`print(isfloat("#56"))`	`False`	
`print(isfloat("56%"))`	`False`	
`print(isfloat("0E0"))`	`True`	
`print(isfloat("x86E0"))`	`False`	
`print(isfloat("86-5"))`	`False`	
`print(isfloat("True"))`	`False`	Boolean is not a float
`print(isfloat(True))`	`True`	Boolean is a float
`print(isfloat("+1e1^5"))`	`False`	
`print(isfloat("+1e1"))`	`True`	
`print(isfloat("+1e1.3"))`	`False`	
`print(isfloat("+1.3P1"))`	`False`	
`print(isfloat("-+1"))`	`False`	
`print(isfloat("(1)"))`	`False`	Brackets not interpreted

In [8]:
```
def isfloat(s: str) -> bool:
    """Check whether string is convertible to float."""
    try:
        float(s)  # result discarded
        return True  # cannot raise ValueError
    except ValueError:
        return False
```

Trade-offs between LBYL and EAFP

- Amount of code
- Ease of checking up front
  - `math.sqrt()`: simple
  - `float()`: hard
- Performance
  - if assumption usually satisfied, `try` is faster
  - otherwise, `if` faster
  - in case of doubt, measure

Sphinx: Documentation Generator

Originally developed to document Python

- Based on *Docutils*
- Uses [*reStructuredText*]() format
- File extension `*.rst`
- Advice: Imitate given examples (HA-0, HA-1)

reStructuredText

- Text markup
  - Similar to *MarkDown* (used in Jupyter notebooks)
  - But not the same!
  - [reStructuredText Primer]()
- [Interpreted text roles]()
- [Directives]()

reST versus MarkDown

- Cannot use `_` (underscore) for italic/bold
  - Must use `*italic*` and `**bold**`
- Cannot use `` `typewriter` ``
  - Must use ``` ``typewriter`` ``` or *code role*
- Cannot use
  ```` ```python ````
  `code`
  ```` ``` ````
  - Must use *code directive*
- Cannot use `$math$` or `$$math$$`
  - Must use *math role* or *math directive*
- Bullet/enumerated list must be preceded by *empty line*

reStructedText: Text roles

- For *inline* use
- Syntax

```
... :role:`interpreted text` ...
```

- `:code:`
- `:math:`
- Sphinx adds its own
  - `:const:`
  - `:data:`
  - `:func:`
  - `:class:`

- `:attr:`
- `:meth:`

- reStructuredText can be used in *docstrings*
- For functions, use the following *fields*:
  - `:param name: description`
  - `:return: description`
  - `:raise exc: description`
- Do *not* duplicate type information; *avoid*
  - `:type name: ...`
  - `:rtype: ...`

Sphinx Example

In [9]:
```
#: The encoding of the three choice options
OPTIONS = {0: "Rock", 1: "Paper", 2: "Scissors"}

#: The valid choice letters
RPS = "".join(name[0].lower() for name in OPTIONS.values())
```

In [10]:
```
def rps_choice(letter: str) -> int:
    """Return choice integer corresponding to given letter.

    The letter is first converted to lower case.

    Assumptions:

    * ``len(letter) == 1``
    * ``letter.lower() in RPS``

    :param letter: letter to convert to integer
    :return: integer in :const:`OPTIONS` corresponding to ``letter``
    :raise AssertionError: if  ``letter`` is invalid

    :examples:

    >>> rps_choice('r')
    0
    >>> rps_choice('P')
    1
    >>> rps_choice('s')
    2
    >>> rps_choice('X')
    Traceback (most recent call last):
        ...
    AssertionError: letter.lower() must be in RPS
    """
    assert letter.lower() in RPS, "letter.lower() must be in RPS"

    return RPS.index(letter.lower())
```

`rps.rps_choice(letter: str) → int` [source]

Return choice integer corresponding to given letter.

The letter is first converted to lower case.

Assumptions:

- `len(letter) == 1`
- `letter.lower() in RPS`

Parameters

letter – Letter to convert to integer

Returns

Integer in `OPTIONS` corresponding to `letter`

Raises

AssertionError – If `letter` is invalid

Examples

```
>>> rps_choice('r')
0
>>> rps_choice('P')
1
>>> rps_choice('s')
2
>>> rps_choice('X')
Traceback (most recent call last):
    ...
AssertionError: letter.lower() must be in RPS
```

In Python documentation:

- **Show Source**, in panel on the left
- E.g. Built-in Functions: Show Source

reStructuredText: Directives

- For use on *blocks*
- Syntax:

```
.. directive type:: argument
   :option: value
   :option: value

   content
```

- Block content consists of (multiple) indented lines

```
.. image:: picture.png
```

Can tweak options (see HA-1)

```
.. code:: python

   def hello():
       print("Hello")
```

- Sphinx directives
- Sphinx Autodoc generates most of these from source code
- In project root directory, run (in Terminal):

```
$ sphinx-apidoc -f -o docs/source src tests
```

- Can include option `-n` (before `-o`) for *dry run*
  - Shows which files will be created
  - Does not create any files

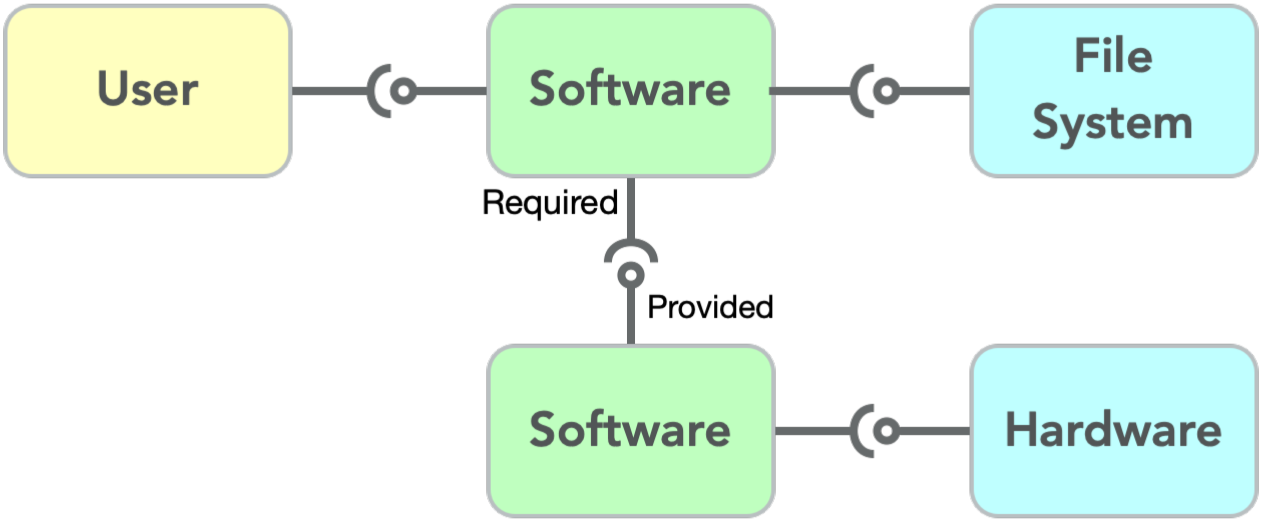
Advice on Documentation

- Keep It Simple, Stupid (KISS)
- This is not the main goal of the course
- (But software documentation is too often forgotten)

Interface Design

- Interface:
  - Sits between two parties
  - *Connects* and *separates*
  - Passes *control* and *data*
  - Control is usually *unidirectional*
  - Data can be *bidirectional*

Types of interfaces in software



- File system, hardware
- Other software (API)
- Human users (CLI batch/text dialog, GUI)

Application Programming Interface (API)

Program is like a Python *class* or *module*

- Program serves as *library* offering *services*:
  - constants
  - types (classes)
  - functions
- Environment *controls* the program
  - Can call functions in the program
  - Provide input data
  - Receive output data

Command-Line Interface (CLI)

- User selects (some) inputs *before* starting program
  - *options*, *arguments*
- Batch mode
  - Programs produces output (on screen, in files)
  - Terminates when done
  - E.g. `sphinx-apidoc`
- Text dialog
  - Program interactively offers choices one by one
  - User responds
  - *Program controls the user*
  - E.g. `sphinx-quickstart`

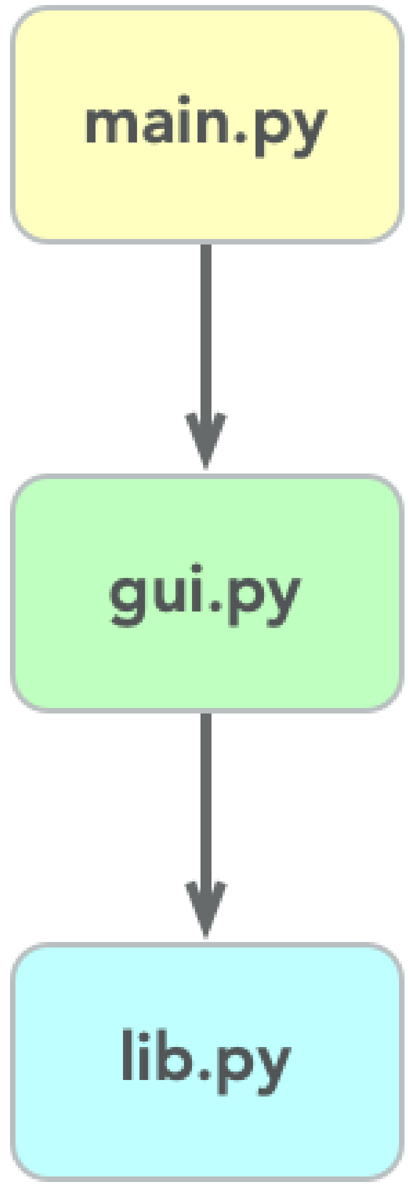
In case you really want/have to go there:

- [How to Build Command Line Interfaces in Python with `argparse`]()

Graphical User Interface (GUI)

- *User controls the program*
  - A.k.a. *direct manipulation*
- User **generates events** (keyboard, mouse)
- In software
  - *main event loop* **dispatches events** (calls *event handlers*)
  - *event handler* **responds to events**

Structure of program with GUI



- Initialization/set-up code
  - **front-end** / GUI
- Main event loop
- Underlying event handlers and utility code
  - **back-end** / business logic

Control flow is partly invisible, hidden in main event loop

GUI with `PyQt5`

Qt5 is a professional C++ GUI library.

- PyQT5 is a binding to it
    - It has the exact same methods and attributes
    - uses Python equivalent types
- Qt is used by many programmers and companies.
    - E.g. LG, Mercedes-Benz

More details:

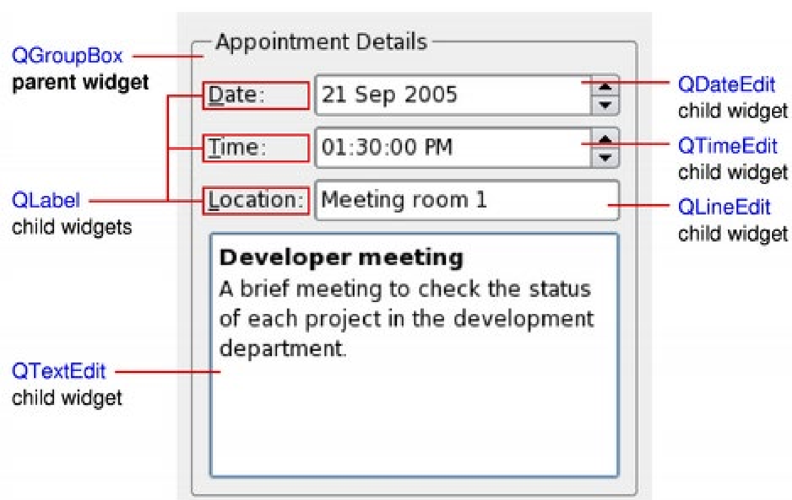
- *DelftStack* Tutorial
- Official Qt5 docs Very good & complete
- Official PyQt5 docs Unfortunately not so complete, better stick to the QT5 documentation!
- PyQt5 Tutorial, Create GUI Applications with Python & Qt — Martin Fitzpatrick
- PyQt5 YouTube Tutorial

GUI Organization in `PyQt5`

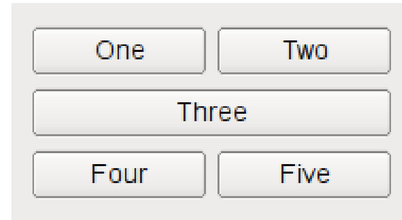
- **QWidgets** (*Interactive objects*):
    - windows, buttons, text areas, frames, ...
    - *Hierarchical*: widgets can contain other widgets
- **Styles**:
    - Look and feel of all the widgets
- **Geometry managers**:
    - Exact placements or using layouts
- **Events handling**:
    - Event = function being called
- *Main event loop*

QWidgets

- **QMainWindow, QWidget**, (main) window or create new windows
- **QPushButton**, to click
- **QCheckBox, QRadioButton**, to select
- **QLineEdit, QTextEdit**, to enter data
- **QLabel**, to present short text
- **QPixmap**, for drawing
- **QGroupBox**, to group widgets
- **QMenuBar**, horizontal menubar
- **QDialog**, to show a message dialog



Geometry Managers



- QLayout
  - Recommended: [QGridLayout]
- use `setGeometry()` (but keep track of resize Events yourself)

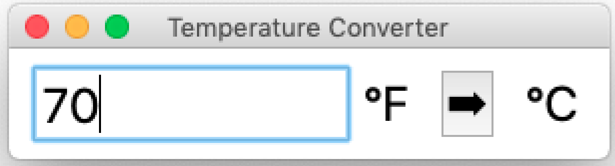
In [11]:
```python
import sys
from PyQt5 import QtGui, QtWidgets

app = QtWidgets.QApplication(sys.argv)
```

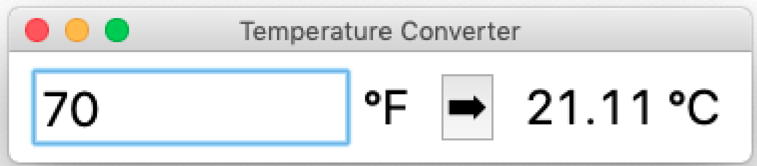
In [12]:
```python
class Window(QtWidgets.QWidget):
    def __init__(self) -> None:
        super().__init__()
        self.grid_layout = QtWidgets.QGridLayout()
        self.setLayout(self.grid_layout)

        for y in range(3):
            for x in range(2):
                label = QtWidgets.QPushButton(f"Button ({x}, {y})")
                self.grid_layout.addWidget(label, y, x)
        big_button = QtWidgets.QPushButton("Big Button")
        self.grid_layout.addWidget(big_button, 3, 0, 1, 2)


window = Window()
window.show()
result = app.exec_()
```

Advice on GUI Design

- Start with a simple sketch
- Make a *mock-up* in PowerPoint
- Do group related elements in frames
- Keep It Simple, Stupid (KISS)

GUI Example (adapted from RealPython)





In [13]:
```python
# Back-end code (business logic)
# should not include GUI-related code


def fahrenheit_to_celsius(t: float) -> float:
    """Convert the value for Fahrenheit to Celsius."""
    celsius = (5 / 9) * (t - 32)
    return round(celsius, 2)
```

In [14]:
```python
# Front-end code (GUI)


class Window(QtWidgets.QMainWindow):
    def __init__(self) -> None:
        # Create root window
        super().__init__()
        main = QtWidgets.QWidget()
        layout = QtWidgets.QHBoxLayout()
        self.setCentralWidget(main)
        main.setLayout(layout)

        # increase font size for demo
        bigger_font = self.font()
        bigger_font.setPointSize(36)
        self.setFont(bigger_font)

        # Create widgets and relationships
        self.frm_entry = QtWidgets.QLineEdit("32")
        lbl_temp = QtWidgets.QLabel("\N{DEGREE FAHRENHEIT}")
        btn_convert = QtWidgets.QPushButton("\N{BLACK RIGHTWARDS ARROW}")
        self.lbl_result = QtWidgets.QLabel("       \N{DEGREE CELSIUS}")
        btn_convert.clicked.connect(self.set_temp)
```

```
        # Place widgets
        layout.addWidget(self.frm_entry)
        layout.addWidget(lbl_temp)
        layout.addWidget(btn_convert)
        layout.addWidget(self.lbl_result)

    def set_temp(self) -> None:
        temp_f = fahrenheit_to_celsius(float(self.frm_entry.text()))
        self.lbl_result.setText(f"{temp_f} \N{DEGREE CELSIUS}")


# Start main event loop
window = Window()
window.show()
result = app.exec_()
```

Imperative Programming: The Big Picture

('imperative' = 'by giving commands')

- **Data**: *variables*

  Python: named & typed objects

- **Actions** on data: *statements* (commands)

  Python: `name = expr`, `if`, `while`, `function(...)`, `object.method(...)`

Statements can be *grouped* into a named, parameterized *function*

```
def function_name(parameters):
    statements
```

Variables can be *grouped* into a named, instantiable *class*,
together with relevant operations (*methods*) on these variables

```
class Class_name:
    variables_and_methods
```

This grouping is also known as **encapsulation**.

Functional decomposition

- Traditional view of computational problems: to define a (single) function.
- Client provides arguments (input), and function produces desired result (output).
- Instead of writing all statements of the solution in that single function,

break it up into smaller functions, whose *composition* solves the problem.

You can also use predefined libary functions.

- **Decomposition** = breaking 'large' thing up into composition of 'smaller' things

Advantages of decomposition (*Divide & Conquer*):

- Easier to understand why it works
- Easier to get it to work
- Easier to document
- Easier to test
- Easier to reuse parts

Data decomposition

Computational problems often concern *multiple* related operations on data.

- 'Modern' (OO) view on computational problems: to define a (single) class holding all the data, and offering methods as operations (services).

  Think of an electronic calculator: each button corresponds to a method

- Client instantiates class, and repeatedly calls methods.

- Instead of writing all variables of the solution in that single class, break it up into smaller classes, whose *composition* solves the problem.

  You can also use predefined library classes.

GUI Library

- GUI library (like `PyQt5`) is example of data decomposition
- Lots of data involved in GUI
  - configuration details
  - state (what data did user enter)
- Data is distributed over separate classes (objects)

OO Design: Nouns and verbs

- Consider the story behind your software
  - **nouns** relate to data
  - **verbs** relate to functions (actions)
- Functional decomposition:
  - decompose actions (data is secondary)
- Data decomposition:
  - decompose data (actions are secondary)

- **Top-down** view
  - initially consider problem as one whole
  - break it up into smaller pieces
- **Bottom-up** view
  - start with fragments
  - compose them into larger pieces

Separation of Concerns

Source: Building Skills in Object-Oriented Design by Steven F. Lott

When simulating Roulette, you encounter nouns:

- Wheel
- Bet
- Bin
- Table
- Red, Black, Green
- Number
- Odds
- Player
- House

Image source: https://pixabay.com/photos/roulette-roulette-wheel-ball-turn-1003120/

Some roulette classes:

- `Outcome`
- `Wheel`
- `Table`
- `Player`
- `Game`

`Outcome`

Responsibilities.

- A name for the bet and the payout odds.
- This isolates the calculation of the payout amount.
- Example: `"Red", "1:1"`.

Collaborators.

- Collected by a `Wheel` object into the bins that reflect the bets that win;
- collected by a `Table` object into the available bets for the `Player`;
- used by a `Game` object to compute the amount won from the amount that was bet.

`Wheel`

Responsibilities.

- Selects the `Outcome` instances that win.
- This isolates the use of a random number generator to select `Outcome` instances.
- It encapsulates the set of winning `Outcome` instances that are associated with each individual number on the wheel.
- Example: the "1" bin has the following winning `Outcome` instances:
    - `"1", "Red", "Odd", "Low", "Column 1", "Dozen 1-12", "Split 1-2", "Split 1-4", "Street 1-2-3", "Corner 1-2-4-5", "Five Bet", "Line 1-2-3-4-5-6", "00-0-1-2-3", "Dozen 1", "Low" and "Column 1"`.

Collaborators.

- Collects the `Outcome` instances into bins;
- used by the overall `Game` to get a next set of winning `Outcome` instances.

`Table`

Responsibilities.

- A collection of bets placed on `Outcome` instances by a `Player`.
- This isolates the set of possible bets and the management of the amounts currently at risk on each bet.
- This also serves as the interface between the `Player` and the other elements of the game.

Collaborators.

- Collects the `Outcome` instances;
- used by `Player` to place a bet amount on a specific `Outcome`;
- used by `Game` to compute the amount won from the amount that was bet.

`Player`

Responsibilities.

- Places bets on `Outcome` instances,
- updates the stake with amounts won and lost.

Collaborators.

- Uses `Table` to place bets on `Outcome` instances;
- used by `Game` to record wins and losses.

`Game`

Responsibilities.

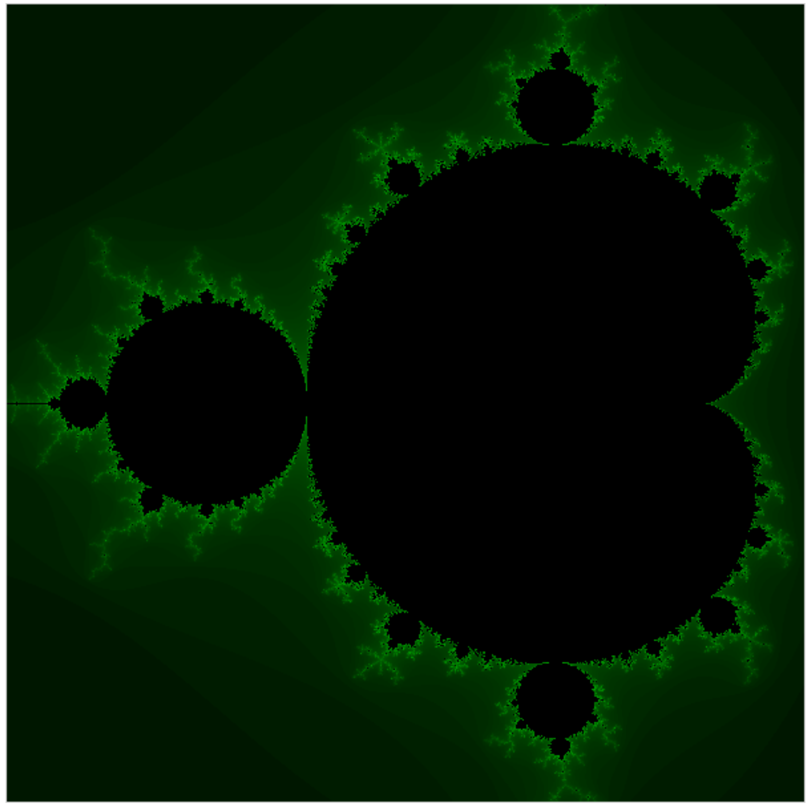
- Runs the game:
    - gets bets from `Player`,
    - spins `Wheel`,
    - collects losing bets,
    - pays winning bets.
- This encapsulates the basic sequence of play into a single class.

Collaborators.

- Uses `Wheel`, `Table`, `Outcome`, `Player`.
- The overall statistical analysis will
    - play a finite number of games and
    - collect the final value of the `Player`'s stake.

(End of Notebook)

2IS50 – Software Development for Engineers – 2022-2023



Lecture 6.B (Sw. Eng.)

Lecturer: Tom Verhoeff

Review of Lecture 5.B

- Using exceptions: EAFP versus LBYL
- Sphinx documentation
  - reStructuredText (reST, RST)
- Interface design
  - Application Programming Interface (API)
  - Command-Line Interface (CLI)
  - Graphical User Interface (GUI), `PyQt5`
- Data decomposition

Preview of Lecture 6.B

- Open-Source
  - Software: (F(L))OSS
  - Hardware
  - Standards
- Revisit Dice Game of Exercises 5

        - Trade-offs
        - The price of cleverness
- Study Markov Analysis
    - Class design

In [1]:
```
# enable mypy type checking
try:
    %load_ext nb_mypy
except ModuleNotFoundError:
    print("Type checking facility (Nb Mypy) is not installed.")
    print("To use this facility, install Nb Mypy by executing (in a cell):
")
    print("  !python3 -m pip install nb_mypy")
```

```
Version 1.0.3
```

In [2]:
```
import math
import random
import collections as co
import itertools as it
from typing import Tuple, List, Dict, Set, DefaultDict, Counter
from typing import Any, Optional, Sequence, Mapping, MutableMapping, Itera
ble
from typing import Hashable, Callable, Iterator, Generator
from typing import NewType, TypeVar, Generic
import doctest
```

Open-Source

With *free* access to source code, incl. design details

- *License* regulates rights and responsibilities
- Can apply to
    - *software*
    - *hardware*
    - *standards* (e.g., WiFi)
- Free/Libre and Open-Source Software: F(L)OSS
- Opposite of *commercial* or *proprietary*: **closed-source**

Also see: https://en.wikipedia.org/wiki/Open-source_software

- Cf. *open-access* academic publications

Free

- Free = gratis (at no cost, as in "a free lunch")
- Free = libre (with freedom of use, as in "free speech")

See:

- https://www.gnu.org/philosophy/floss-and-foss.html
    - Richard Stallman, GNU Project,
    - **Free Software Foundation** (FSF)
- https://en.wikipedia.org/wiki/Free_and_open-source_software

Four Essential Freedoms

(according to FSF)

- Freedom to *run program as you wish*, for any purpose (freedom 0).
- Freedom to *study how program works*, and
  change it so it does your computing as you wish (freedom 1).
    - Access to the source code is a precondition for this.
- Freedom to *redistribute copies* so you can help others (freedom 2).
- Freedom to *distribute copies of your modified versions* to others (freedom 3).
    - By doing so, you can give whole community chance to benefit from your changes.
    - Access to the source code is a precondition for this.

Copyright

As author of program you own the **copyright**, unless ...

- Even if you don't *claim it* (by writing a copyright notice)
- No need to *pay* for copyright
- Software that you write (from scratch) is your **intellectual property** (IP)

Software Licenses

License can regulate

- application of software (what to use it for)
- access to source code
- whether you may reverse engineer
- whether you may modify
- whether you may redistribute (free, or for money)
- whether you may reuse it within other software
- ...

Also see:

- https://en.wikipedia.org/wiki/Software_license
- https://en.wikipedia.org/wiki/Free-software_license

Two sides of sofware licenses:

- as author

choose appropriate license
    - must agree with licenses of third-party software you use
- as user
    - read license of software you use
    - adhere to license of software you use

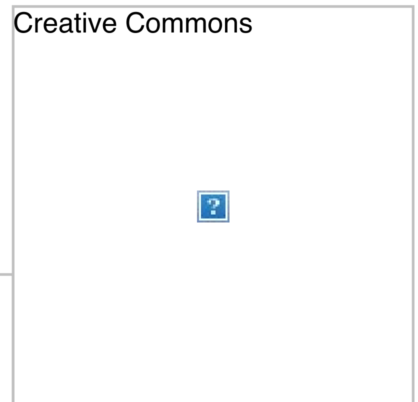
Differences across the globe:

- Europe: no software patents
- US of A: software patents
- Asia: ...

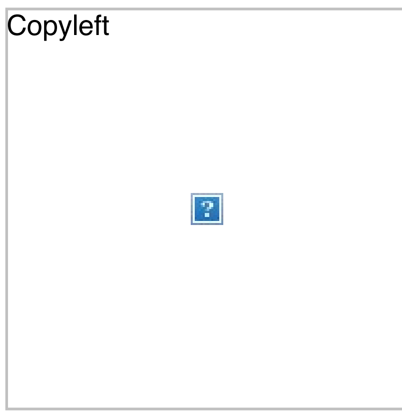
Open-Source Software Licenses

- Many flavors
- Subtle differences
- Public Domain
- Creative Commons, several 'levels'
- Permissive licenses
    - BSD, Apache, MIT, ...
- GPL and LGPL
    - GNU General Public License
    - GNU Library/Lesser GPL
    - **Copyleft**

Creative Commons



Copyleft



UMax Dice Game Revisited

See Exercises 5:

- Game with `n` players
- Player 1 rolls with *one dodecahdron*
- Other players roll with *two regular dice*
- Round is won by player with *unique maximum*
    - If maximum not unique: *tie*

Question: Who has best winnings odds?

Monolithic code given

Function `simulate(n, r)` will

- simulate r rounds with n players, and
- return win counts per player, where
- Player 0 represents *TIE*

In [3]:
```
def simulate(n: int, r: int) -> Sequence[int]:
    """Simulate r rounds of the n-player game UMax,
    returning a sequence with win counts.
    """
    result = (n + 1) * [0]

    for _ in range(r):
        # simulate one round
        rolls = [0]  # dummy roll at index 0

        for i in range(1, 1 + n):
            # roll dice for player i
            if i == 1:
                roll = random.randint(1, 12)
            else:
                roll = random.randint(1, 6) + random.randint(1, 6)
            rolls.append(roll)

        m = 0  # maximum so far

        for i in range(1, 1 + n):
            if rolls[i] > m:
                m = rolls[i]

        c = 0  # count of m so far

        for i in range(1, 1 + n):
            if rolls[i] == m:
                c += 1

        if c > 1:
            # no winner
            winner = 0
        else:

            for winner in range(1, 1 + n):
                if rolls[winner] == m:
                    break

        result[winner] += 1

    return result
```

Exercises 5 asks for

- *Functional decomposition*
- *OO/data decomposition*
- Both using functions from *Python Standard Library*
    - `max`
    -

    - `list.count`
    - `list.index`

Decomposition trade-offs (mantra):

- Benefits
    - easier to understand (if you know ...)
    - easier to get it to work
    - easier to document
    - easier to test
    - easier to modify
    - easier to reuse (but: ...)
- Costs (overhead)
    - more code
    - harder to understand (if you don't know ...)
    - performance penalty (function calls, objects)

Let's improve performance of round simulation:

- Now: 3 loops, viz. in `max`, `count`, `index`
- Wanted: 1 loop

In [4]:

```python
def simulate_round(rolls: List[int]) -> int:
    """Return winner for given rolls (0 if no winner).

    >>> simulate_round([1, 2, 3])
    3
    >>> simulate_round([3, 1, 3])
    0
    >>> simulate_round([3, 1, 3, 4])
    4
    """
    n = len(rolls)
    rolls.insert(0, 0)  # see monolithic code above

    m = 0  # maximum so far

    for i in range(1, 1 + n):
        if rolls[i] > m:
            m = rolls[i]

    c = 0  # count of m so far

    for i in range(1, 1 + n):
        if rolls[i] == m:
            c += 1

    if c > 1:
        # no winner
        winner = 0
    else:
```

```
        for winner in range(1, 1 + n):
            if rolls[winner] == m:
                break

    return winner
```

In [5]:
```
doctest.run_docstring_examples(simulate_round, globs=globals(), name="simu
late_round")
```

In [6]:
```
def simulate_round_clever(rolls: List[int]) -> int:
    """Return winner for given rolls (0 if no winner).

    >>> simulate_round_clever([1, 2, 3])
    3
    >>> simulate_round_clever([3, 1, 3])
    0
    >>> simulate_round_clever([3, 1, 3, 4])
    4
    """
    rolls.insert(0, 0)  # see monolithic code above

    maximum = 0  # maximum so far
    winner = 0  # winner so far

    for player, roll in enumerate(rolls):
        if roll > maximum:
            maximum, winner = roll, player
        elif roll == maximum:
            winner = 0
#          print(f"maximum, winner == {maximum}, {winner}")

    return winner
```

In [7]:
```
doctest.run_docstring_examples(
    simulate_round_clever, globs=globals(), name="simulate_round_clever"
)
```

Can even integrate this into `rolls` generation loop

- 4 loops merged into 1 loop (save time)
- list `rolls` is not needed (save memory)

In [8]:
```
def simulate_clever(n: int, r: int) -> Sequence[int]:
    """Simulate r rounds of the n-player game UMax,
    returning a sequence with win counts.
    """
    result = (n + 1) * [0]

    for _ in range(r):
        # simulate one round
        maximum = 0  # maximum so far
        winner = 0  # winner so far
```

```
        for player in range(1, 1 + n):
            if player == 1:
                roll = random.randint(1, 12)
            else:
                roll = random.randint(1, 6) + random.randint(1, 6)
            if roll > maximum:
                maximum, winner = roll, player
            elif roll == maximum:
                winner = 0

        result[winner] += 1

    return result
```

In [9]:
```
simulate_clever(5, 1000)
```

Out[9]:
```
[209, 210, 159, 140, 135, 147]
```

Lessons for loop design

- Determine which data is relevant for result
- Determine which data is relevant to update data in loop
- Write the loop:
  - initialize the data *before loop*
  - update data *inside loop*
  - use (some) data *after loop*, for result

- You can avoid most loops of the form
  - `for i in range(...):`
- If you need the index, use `enumerate(...)`
  - Can choose *start index*: `enumerate(..., start)`

Price of Cleverness

Yes, it may be a little *faster* and *shorter*, but

- Harder to understand
- Harder to modify
- Harder to reuse (harder than `max`, `count`, `index`)

Only improve performance *when* and *where* needed

- "... programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times;\ **premature optimization is the root of all evil** (or at least most of it) in programming."\ (Donald Knuth in *The Art of Computer Programming*)
- Before optimizing: Measure!

Class Design: Markov Analysis

- *Markov (Chain) Models*

Also see: *Think Python* (2e, Ch.13)

Two aspects of randomness (Lecture 6.A):

- **Uniformity**: overall frequencies are equal
  - concerns probabilities for options regardless of event
- **Independence**: frequencies are independent of past
  - concerns probabilities of options across events

Randomness in Rock-Paper-Scissors:

- `{ROCK: 0.1, PAPER: 0.4, SCISSORS: 0.5}`
  - not uniform
  - independent
- Avoid previous choice; choose 50-50 between other two
  - uniform (overall)
  - not independent

Natural language (per letter):

- not uniform
  - 'e' most frequent (11%)
  - 'z' least frequent (0.08%)
- not independent
  - 'u' after 'q' much more frequent than
  - 'u' after other letter

Also see: [Letter frequency]()

Can also study language *per word*

Markov Model of Order `n`

- Captures *memory effect*
  - dependence of distribution on past `n` items
- State: tuple of length `n`
- For each (observed) state:
  - store distribution of next items after that state

Disclaimer:

- There are various pitfalls when using Markov models
- Here, we only touch on the basics

Notes about the following code:

- In Python, syntax for tuple of length 1: `(item,)`
  - N.B. comma required

Execute following code, but skip details on first reading

In [10]:
```
K = TypeVar("K", bound=Hashable)  # not exam material; think of K as str o
r int
State = Tuple[K, ...]
MM = Mapping[State, Mapping[K, int]]


class MarkovModel(Generic[K]):
    """A MarkovModel of order n stores tuples over type K of length n,
    and associates them with a Counter[K] (a distribution over K).

    An order-0 MarkovModel is just a distribution over K.

    A MarkovModel can be updated, and it can serve as an iterable
    to generate items according to the current model.
    """

    def __init__(self, order: int, model: Optional[MM] = None) -> None:
        """Initialize an empty Markov model of given order.

        Assumptions:

        * all(len(state) == order for state in model) if model is not None
        """
        self.order = order
        self.model: DefaultDict[State, Counter[K]]
        if model is None:
            self.model = co.defaultdict(co.Counter)
        else:
            # convert model to type MM
            self.model = co.defaultdict(
                co.Counter, {item: Counter(counts) for item, counts in mod
el.items()}
            )

    def __repr__(self) -> str:
        return f"{self.__class__.__name__}({self.order}, {self.get_model()
!r})"

    def get_model(self) -> MM:
        """Get the model in plain form."""
        return {state: dict(distr) for state, distr in self.model.items()}

    def get_totals(self) -> Mapping[State, int]:
        """Get total count per state."""
        return {state: sum(distr.values()) for state, distr in self.model.
items()}

    def get_weights(self) -> Tuple[int, ...]:
        """Get total count as vector."""
```

```
        return tuple(sum(distr.values()) for distr in self.model.values())

    def split(self, iterable: Iterable[K]) -> Tuple[State, Iterable[K]]:
        """Split iterable in state (of length self.order) and remainder.

        Assumption: iterable yields at least self.order items
        """
        if isinstance(iterable, Iterator):
            iterator = iterable
        else:
            iterator = iter(iterable)
        return tuple(it.islice(iterator, self.order)), iterator

    def update(self, state: State, iterable: Iterable[K]) -> State:
        """Update the model with items from given iterable after given sta
te,
        and return next state.

        Assumption: len(state) == self.order
        """
        for item in iterable:
            self.model[state][item] += 1
            state = (state + (item,))[1:]  # append item, then drop first
            # not correct (when self.order == 0): state = state[1:] + (ite
m, )

        return state

    def generate_state(self) -> State:
        """Generate a state according to the model.

        Assumption: model is not empty
        """
        assert self.model, "model must not be empty"

        return random.choices(
            tuple(self.model.keys()), weights=self.get_weights(), k=1
        )[0]

    def generate(self, state: State) -> Tuple[K, State]:
        """Generate an item and the next state for given state state.
        The last item of the returned state is the newly generated item.
        The returned tuple has length self.order.

        Assumption: len(state) == self.order
        """
        if state not in self.model:
            state = self.generate_state()
        distr = self.model[state]
        item = random.choices(tuple(distr.keys()), weights=tuple(distr.val
ues()), k=1)[
            0
        ]
        return item, (state + (item,))[1:]

    def __iter__(self) -> Iterator[K]:
        """Implement iter(self)."""
```

```python
        state = self.generate_state()
        yield from state

        while True:
            item, state = self.generate(state)
            yield item


# Note: Could use Deque[K] instead of Tuple[K, ...]
```

In [11]:
```python
help(MarkovModel)
```

```
Help on class MarkovModel in module __main__:

class MarkovModel(typing.Generic)
 |  MarkovModel(order: int, model: Optional[Mapping[Tuple[~K, ...], Mappin
g[~K, int]]] = None) -> None
 |
 |  A MarkovModel of order n stores tuples over type K of length n,
 |  and associates them with a Counter[K] (a distribution over K).
 |
 |  An order-0 MarkovModel is just a distribution over K.
 |
 |  A MarkovModel can be updated, and it can serve as an iterable
 |  to generate items according to the current model.
 |
 |  Method resolution order:
 |      MarkovModel
 |      typing.Generic
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, order: int, model: Optional[Mapping[Tuple[~K, ...], Map
ping[~K, int]]] = None) -> None
 |      Initialize an empty Markov model of given order.
 |
 |      Assumptions:
 |
 |      * all(len(state) == order for state in model) if model is not None
 |
 |  __iter__(self) -> Iterator[~K]
 |      Implement iter(self).
 |
 |  __repr__(self) -> str
 |      Return repr(self).
 |
 |  generate(self, state: Tuple[~K, ...]) -> Tuple[~K, Tuple[~K, ...]]
 |      Generate an item and the next state for given state state.
 |      The last item of the returned state is the newly generated item.
 |      The returned tuple has length self.order.
 |
 |      Assumption: len(state) == self.order
 |
 |  generate_state(self) -> Tuple[~K, ...]
 |      Generate a state according to the current model.
 |
```

```
 |      Assumption: model is not empty
 |
 |  get_model(self) -> Mapping[Tuple[~K, ...], Mapping[~K, int]]
 |      Get the model in plain form.
 |
 |  get_totals(self) -> Mapping[Tuple[~K, ...], int]
 |      Get total count per state.
 |
 |  get_weights(self) -> Tuple[int, ...]
 |      Get total count as vector.
 |
 |  split(self, iterable: Iterable[~K]) -> Tuple[Tuple[~K, ...], Iterable[
~K]]
 |      Split iterable in state (of length self.order) and remainder.
 |
 |      Assumption: iterable yields at least self.order items
 |
 |  update(self, state: Tuple[~K, ...], iterable: Iterable[~K]) -> Tuple[~
K, ...]
 |      Update the model with items from given iterable after given state,
 |      and return next state.
 |
 |      Assumption: len(state) == self.order
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __orig_bases__ = (typing.Generic[~K],)
 |
 |  __parameters__ = (~K,)
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from typing.Generic:
 |
 |  __class_getitem__(params) from builtins.type
 |
 |  __init_subclass__(*args, **kwargs) from builtins.type
 |      This method is called when a class is subclassed.
 |
 |      The default implementation does nothing. It may be
 |      overridden to extend subclasses.
```

Some automated test cases:

In [12]:
```
MM_examples = """
    >>> # Order-0
```

```
    >>> mm = MarkovModel(0, {(): {'T': 3}})
    >>> mm
    MarkovModel(0, {(): {'T': 3}})
    >>> mm.get_totals()
    {(): 3}
    >>> mm.get_weights()
    (3,)
    >>> mm.update((), [])
    ()
    >>> mm
    MarkovModel(0, {(): {'T': 3}})
    >>> mm.update((), ['H'])
    ()
    >>> mm
    MarkovModel(0, {(): {'T': 3, 'H': 1}})
    >>> # Order-1
    >>> mm = MarkovModel(1, {('H',): {'H': 1, 'T': 3},
    ...                      ('T',): {'H': 4, 'T': 1}})
    >>> mm
    MarkovModel(1, {('H',): {'H': 1, 'T': 3}, ('T',): {'H': 4, 'T': 1}})
    >>> mm.get_totals()
    {('H',): 4, ('T',): 5}
    >>> mm.get_weights()
    (4, 5)
    >>> mm.update(('H',), ['T'])
    ('T',)
    >>> mm
    MarkovModel(1, {('H',): {'H': 1, 'T': 4}, ('T',): {'H': 4, 'T': 1}})
    >>> # Order-2
    >>> mm = MarkovModel(2)
    >>> mm
    MarkovModel(2, {})
    >>> mm.update((0, 1), [0, 1, 2])
    (1, 2)
    >>> mm
    MarkovModel(2, {(0, 1): {0: 1, 2: 1}, (1, 0): {1: 1}})
    >>> state, rest = mm.split(range(5))
    >>> state
    (0, 1)
    >>> [item for item in rest]
    [2, 3, 4]
"""
```

In [13]:
```
doctest.run_docstring_examples(MM_examples, globs=globals(), name="MarkovModel")
```

Generating from Order-`n` Markov Model

Goal: Generate random sequence of items according to given MM

State is tuple of `n` items

1. Choose *initial state*
2. Yield its items, one by one
3. Choose *next item*, based on distribution for current state

4. Update state: *sliding window*
    - [0 1] -> 0 [1 2] -> 0 1 [2 3] -> 0 1 2 [3 4] -> 0 1 2 3 [4 5]
5. Repeat from 3.

In [14]:
```python
mm: MarkovModel[str]
mm = MarkovModel(0, {(): {"|": 1, "_": 1}})

gen_0_1_1 = "".join(item for item in it.islice(mm, 80))
print(gen_0_1_1)
```

```
|__|_|_|___|__||__|_||||__|_|||__||__|__||__||||______|__||__|____|_||_||_
_|||__
```

In [15]:
```python
mm: MarkovModel[str]
mm = MarkovModel(0, {(): {"|": 1, "_": 4}})

gen_0_1_4 = "".join(item for item in it.islice(mm, 80))
print(gen_0_1_4)
```

```
_____|__|||________||_|_____|__|__________________|_______|______|__||___|_
______
```

In [16]:
```python
mm: MarkovModel[str]
mm = MarkovModel(1, {("|",): {"|": 4, "_": 1}, ("_",): {"|": 1, "_": 4}})

gen_1_4_1_1_4 = "".join(item for item in it.islice(mm, 80))
print(gen_1_4_1_1_4)
```

```
____||||||||||||||||||____|||_________||||||_____|||_____||_||||||_||||||||
||____
```

In [17]:
```python
mm: MarkovModel[str]
mm = MarkovModel(1, {("|",): {"|": 1, "_": 4}, ("_",): {"|": 4, "_": 1}})

gen_1_1_4_4_1 = "".join(item for item in it.islice(mm, 80))
print(gen_1_1_4_4_1)
```

```
|_|_|__|_||_|_|__|_|_|_||_|_|_|_|_|||||__||_|__|_|_|_|_|||__|__||_||_|__|_
|_|_|_
```

Creating Order-`n` Markov Model

Goal: Given a sequence of items, produce MM

State is tuple of `n` items

1. Collect first `n` items
2. Set as *initial state*
3. For *next item*, update distribution for current state
4. Update *state*: slide window
5. Repeat from 3.

In [Machine Learning]() terminology:

- Given sequence: the *training set*
- Creating a model: to *learn* or *train*

In [18]:
```
mm: MarkovModel[str]
mm = MarkovModel(0)

print(gen_0_1_1)
mm.update(*mm.split(gen_0_1_1))   # Note the *
mm
```

```
|__|_|_|___|__||__|_||||__|_|||__||__|__||__||||______|__||__|____|_||_||_
_|||__
```

Out[18]:
```
MarkovModel(0, {(): {'|': 37, '_': 43}})
```

In [19]:
```
mm: MarkovModel[str]
mm = MarkovModel(0)

print(gen_0_1_4)
mm.update(*mm.split(gen_0_1_4))
mm
```

```
_____|__|||________||_|_____|__|_________________|_______|______|__||___|_
______
```

Out[19]:
```
MarkovModel(0, {(): {'_': 65, '|': 15}})
```

In [20]:
```
mm: MarkovModel[str]
mm = MarkovModel(1)

print(gen_1_4_1_1_4)
mm.update(*mm.split(gen_1_4_1_1_4))
mm
```

```
____||||||||||||||||||____|||________||||||_____|||_____||_||||||_||||||||
||____
```

Out[20]:
```
MarkovModel(1, {('_',): {'_': 24, '|': 7}, ('|',): {'|': 41, '_': 7}})
```

In [21]:
```
mm: MarkovModel[str]
mm = MarkovModel(1)

print(gen_1_1_4_4_1)
mm.update(*mm.split(gen_1_1_4_4_1))
mm
```

```
|_|_|__|_||_|_|__|_|_|_||_|_|_|_|_|||||__||_|__|_|_|_|_|||__|__||_||_|__|_
|_|_|_
```

Out[21]:
```
MarkovModel(1, {('|',): {'_': 31, '|': 11}, ('_',): {'|': 30, '_': 7}})
```

Models of English Text

Some experiments with the book *Emma* by Jane Austen

- Available from *Project Gutenberg*; copyright has expired

Version without all meta-data (headers) is available as `emma-plain.txt`

- Make sure it is in same folder as this notebook

In [22]:
```
with open("emma-plain.txt") as f:
    for line in it.islice(f, 12):
        print(line, end="")  # line already includes newline
```

```
Emma Woodhouse, handsome, clever, and rich, with a comfortable home
and happy disposition, seemed to unite some of the best blessings
of existence; and had lived nearly twenty-one years in the world
with very little to distress or vex her.

She was the youngest of the two daughters of a most affectionate,
indulgent father; and had, in consequence of her sister's marriage,
been mistress of his house from a very early period.  Her mother
had died too long ago for her to have more than an indistinct
remembrance of her caresses; and her place had been supplied
by an excellent woman as governess, who had fallen little short
of a mother in affection.
```

Convert file into character stream

- Open file is iterable over its lines
  - each line is iterable over its characters
- We want file as iterable over (some of) its characters
- `it.chain` to the rescue
  - also: `it.chain.from_iterable`

In [23]:
```
help(it.chain)
```

```
Help on class chain in module itertools:

class chain(builtins.object)
 |  chain(*iterables) --> chain object
 |
 |  Return a chain object whose .__next__() method returns elements from t
he
 |  first iterable until it is exhausted, then elements from the next
 |  iterable, until all of the iterables are exhausted.
 |
 |  Methods defined here:
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
```

```
 |  __next__(self, /)
 |      Implement next(self).
 |
 |  __reduce__(...)
 |      Return state information for pickling.
 |
 |  __setstate__(...)
 |      Set state information for unpickling.
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585
 |
 |  from_iterable(iterable, /) from builtins.type
 |      Alternative chain() constructor taking a single iterable argument
that evaluates lazily.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signa
ture.
```

In [24]:

```
with open("emma-plain.txt") as f:
    for char in it.islice(it.chain(*f), 147):  # Note the *
        print(char, end="")
```

```
Emma Woodhouse, handsome, clever, and rich, with a comfortable home
and happy disposition, seemed to unite some of the best blessings
of existence;
```

Order-0 Model of English Text

- Letter frequencies

In [25]:

```
mm_en_0: MarkovModel[str]
mm_en_0 = MarkovModel(0)

with open("emma-plain.txt") as f:
    mm_en_0.update(*mm_en_0.split(it.chain(*f)))

mm_en_0
```

Out[25]:

```
MarkovModel(0, {(): {'E': 1444, 'm': 17907, 'a': 53667, ' ': 147571, 'W':
1355, 'o': 52893, 'd': 28328, 'h': 40828, 'u': 20604, 's': 41554, 'e': 845
16, ',': 12018, 'n': 46984, 'c': 14815, 'l': 27539, 'v': 7645, 'r': 40698,
 'i': 42590, 'w': 14935, 't': 58068, 'f': 14598, 'b': 10532, '\n': 16757,
'p': 10284, 'y': 15266, 'g': 13525, 'x': 1346, ';': 2353, '-': 6774, '.':
8882, 'S': 952, 'q': 895, "'": 1116, 'H': 1685, 'M': 2793, 'T': 1077, 'B':
 598, '_': 741, 'j': 688, 'k': 4351, 'I': 3926, 'A': 654, ':': 174, '?': 6
21, '(': 107, ')': 107, 'L': 132, 'O': 303, 'N': 301, 'C': 592, '"': 4187,
```

```
 'P': 202, '!': 1063, 'R': 161, 'J': 432, 'Y': 439, 'K': 412, 'D': 254, '`
': 112, 'z': 175, 'F': 541, 'G': 147, 'U': 38, 'Q': 15, 'V': 69, '8': 3, '
2': 5, '3': 1, '4': 1, '&': 3, '7': 1, '1': 2, '0': 8, '[': 1, ']': 1, '6'
: 1}})
```

In [26]:
```
mm_en_0.model[()].most_common()
```

Out[26]:
```
[(' ', 147571),
 ('e', 84516),
 ('t', 58068),
 ('a', 53667),
 ('o', 52893),
 ('n', 46984),
 ('i', 42590),
 ('s', 41554),
 ('h', 40828),
 ('r', 40698),
 ('d', 28328),
 ('l', 27539),
 ('u', 20604),
 ('m', 17907),
 ('\n', 16757),
 ('y', 15266),
 ('w', 14935),
 ('c', 14815),
 ('f', 14598),
 ('g', 13525),
 (',', 12018),
 ('b', 10532),
 ('p', 10284),
 ('.', 8882),
 ('v', 7645),
 ('-', 6774),
 ('k', 4351),
 ('"', 4187),
 ('I', 3926),
 ('M', 2793),
 (';', 2353),
 ('H', 1685),
 ('E', 1444),
 ('W', 1355),
 ('x', 1346),
 ("'", 1116),
 ('T', 1077),
 ('!', 1063),
 ('S', 952),
 ('q', 895),
 ('_', 741),
 ('j', 688),
 ('A', 654),
 ('?', 621),
 ('B', 598),
 ('C', 592),
 ('F', 541),
 ('Y', 439),
 ('J', 432),
 ('K', 412),
```

```
 ('O', 303),
 ('N', 301),
 ('D', 254),
 ('P', 202),
 ('z', 175),
 (':', 174),
 ('R', 161),
 ('G', 147),
 ('L', 132),
 ('`', 112),
 ('(', 107),
 (')', 107),
 ('V', 69),
 ('U', 38),
 ('Q', 15),
 ('0', 8),
 ('2', 5),
 ('8', 3),
 ('&', 3),
 ('1', 2),
 ('3', 1),
 ('4', 1),
 ('7', 1),
 ('[', 1),
 (']', 1),
 ('6', 1)]
```

Lump non-alpha, don't distinguish upper/lower case

Options:

- *Generator expression*: `(s.lower() if s.isalpha() else ' ' for s in ...)`
- *Generator function*:

In [27]:
```
def smash(chars: Iterable[str]) -> Iterator[str]:
    """Map upper case letters to lower case, and
    map all non-alphabetic characters to a space.

    Assuption: all(len(char) == 1 for char in chars)

    >>> ''.join(smash('AbC.dEf,GhI jKl-MnO\\npQr'))  # N.B. double backsla
sh
    'abc def ghi jkl mno pqr'
    """
    for char in chars:
        yield char.lower() if char.isalpha() else " "
```

In [28]:
```
doctest.run_docstring_examples(smash, globs=globals(), name="smash")
```

In [29]:
```
mm_en_0: MarkovModel[str]
mm_en_0 = MarkovModel(0)

with open("emma-plain.txt") as f:
    mm_en_0.update(*mm_en_0.split(smash(it.chain(*f))))
```

```
mm_en_0
```

Out[29]:
```
MarkovModel(0, {(): {'e': 85960, 'm': 20700, 'a': 54321, ' ': 202610, 'w':
 16290, 'o': 53196, 'd': 28582, 'h': 42513, 'u': 20642, 's': 42506, 'n': 4
7285, 'c': 15407, 'l': 27671, 'v': 7714, 'r': 40859, 'i': 46516, 't': 5914
5, 'f': 15139, 'b': 11130, 'p': 10486, 'y': 15705, 'g': 13672, 'x': 1346,
'q': 910, 'j': 1120, 'k': 4763, 'z': 175}})
```

In [30]:
```
mm_en_0.model[()].most_common()
```

Out[30]:
```
[(' ', 202610),
 ('e', 85960),
 ('t', 59145),
 ('a', 54321),
 ('o', 53196),
 ('n', 47285),
 ('i', 46516),
 ('h', 42513),
 ('s', 42506),
 ('r', 40859),
 ('d', 28582),
 ('l', 27671),
 ('m', 20700),
 ('u', 20642),
 ('w', 16290),
 ('y', 15705),
 ('c', 15407),
 ('f', 15139),
 ('g', 13672),
 ('b', 11130),
 ('p', 10486),
 ('v', 7714),
 ('k', 4763),
 ('x', 1346),
 ('j', 1120),
 ('q', 910),
 ('z', 175)]
```

Let's turn this into percentages, ignoring non-alpha:

In [31]:
```
letter_bag = mm_en_0.model[()].most_common()[1:]
total = sum(count for letter, count in letter_bag)
print(f"distinct, total: {len(letter_bag)}, {total}")

{letter: round(100 * count / total, 2) for letter, count in letter_bag}
```

```
distinct, total: 26, 683753
```

Out[31]:
```
{'e': 12.57,
 't': 8.65,
 'a': 7.94,
 'o': 7.78,
 'n': 6.92,
 'i': 6.8,
 'h': 6.22,
```

```
 's': 6.22,
 'r': 5.98,
 'd': 4.18,
 'l': 4.05,
 'm': 3.03,
 'u': 3.02,
 'w': 2.38,
 'y': 2.3,
 'c': 2.25,
 'f': 2.21,
 'g': 2.0,
 'b': 1.63,
 'p': 1.53,
 'v': 1.13,
 'k': 0.7,
 'x': 0.2,
 'j': 0.16,
 'q': 0.13,
 'z': 0.03}
```

Generate Order-0 English Text

In [32]:
```
"".join(char for char in it.islice(mm_en_0, 80))
```

Out[32]:
```
'w   ydnfepssb t hhofa rattloyeha     niviww ai nant tdsoo n iwex igbr ora
mht eso'
```

Order-1 Model of English Text

In [33]:
```
mm_en_1: MarkovModel[str]
mm_en_1 = MarkovModel(1)

with open("emma-plain.txt") as f:
    mm_en_1.update(*mm_en_1.split(smash(it.chain(*f))))
```

What is distribution for letter following "q" and following "j"?

In [34]:
```
mm_en_1.model[("q",)], mm_en_1.model[("j",)]
```

Out[34]:
```
(Counter({'u': 910}), Counter({'u': 335, 'o': 249, 'a': 327, 'e': 209}))
```

For each character, how often is it followed by "u", reverse sorted by percentage?

In [35]:
```
totals = mm_en_1.get_totals()

Counter(
    {
        state: round(100 * mm_en_1.model[state]["u"] / totals[state], 2)
        for state in mm_en_1.model   # state has length 1
    }
).most_common()
```

Out[35]:
```
[(('q',), 100.0),
 (('j',), 29.91),
 (('o',), 15.88),
 (('b',), 15.34),
 (('m',), 5.9),
 (('s',), 4.85),
 (('c',), 3.49),
 (('f',), 3.2),
 (('p',), 1.99),
 (('g',), 1.58),
 (('t',), 1.51),
 (('x',), 1.19),
 (('l',), 1.17),
 (('h',), 1.15),
 (('z',), 1.14),
 (('d',), 1.09),
 (('r',), 0.99),
 (('a',), 0.88),
 ((' ',), 0.67),
 (('n',), 0.43),
 (('v',), 0.13),
 (('i',), 0.03),
 (('e',), 0.01),
 (('w',), 0.0),
 (('u',), 0.0),
 (('y',), 0.0),
 (('k',), 0.0)]
```

Generate Order-1 English Text

In [36]:
```
"".join(char for char in it.islice(mm_en_1, 80))
```

Out[36]:
```
'surshaplitl f freay hatouss hrugermind f  alal  ra t at ad ed  mig end w
win sha'
```

This is almost prounounceable

Higher Order Models of English Text

In [37]:
```
mm_en: Mapping[int, MarkovModel[str]]
mm_en = {order: MarkovModel(order) for order in range(0, 5 + 1)}

for order in range(0, 5 + 1):
    with open("emma-plain.txt") as f:
        mm_en[order].update(*mm_en[order].split(smash(it.chain(*f))))
```

In [38]:
```
for order in range(0, 5 + 1):
    print(
        f"{order}:",
        repr("".join(char for char in it.islice(mm_en[order], 80))),
        end="\n\n",
    )
```

```
0: '  mht is neme ehsai  niwuo iywreyadevssasuor d llsyid  oonri hdhit rh
tlme   go '

1: ' atookneraver wecugestite ech er nshoeterashoone ing ftheot cente hers
 vecoaceel'

2: 'mitesid it thered harmand bas   thes and orin tre at blencieverty war
 beend an '

3: 'dere you findown conce  but preture armedit as him this not gened  alw
ays becomp'

4: 't would   which see mightley said her vision of hoarse  and this with
you will s'

5: 'need not do her how resolutely regretted no such astonishing after and
 it   the '
```

Order 5 without smashing

In [39]:
```python
mm_en_5: MarkovModel[str]
mm_en_5 = MarkovModel(5)

with open("emma-plain.txt") as f:
    mm_en_5.update(*mm_en_5.split(it.chain(*f)))
```

In [40]:
```python
print("".join(char for char in it.islice(mm_en_5, 200)))
```

```
tch."

"If I have disparity for Emma, too."

Harriet been very strong throughly deserve, and talking how much of grosse
d to be very thing,
his better suspectacles, admirations.  You think, indeed, equ
```

ChatGPT is based on this idea but

- using "tokens" (syllables) rather than letters,
- in a more clever way

(End of Notebook)