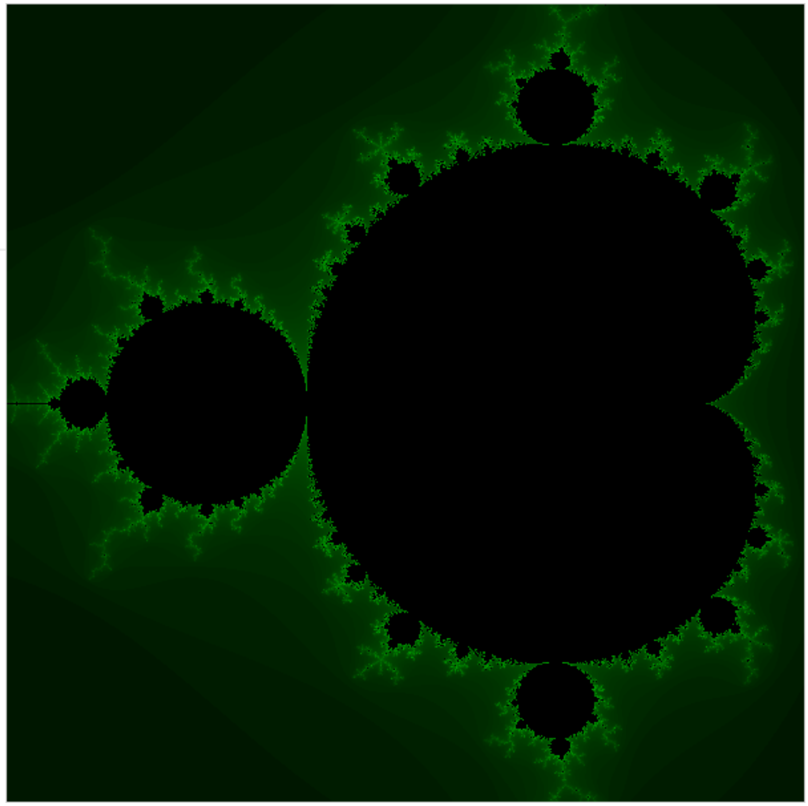
2IS50 – Software Development for Engineers – 2022-2023

Lecture 1.A (Python)

Lecturer: Tom Verhoeff

Also see the book *Think Python* (2e), by Allen Downey



Review of Recap Lecture

- Values, types, literals
  - `int`, `float`, `bool`, `str`
- `print()` and [`.format()`]()
- Expressions
  - Operators and priorities (precedence)
  - Calling functions: built-in, libraries
- Names, variables, assignment statement
  - `name = expression`
  - `x, y = y, x`
- `# Comments`
- [`if`-`elif`-`else` statement]() for conditional execution
  - can be *nested*

Preview of Lecture 1.A (Python)

Programming: easy and hard, fun and frustrating

- `while` loop
    - invariant relation
    - termination, `break`, `continue`, `else`
    - nesting
- Defining functions: `def`
    - parameters, type hints
    - `return`, return type, void or fruitful
    - docstring
    - local variables
    - default arguments

Programming

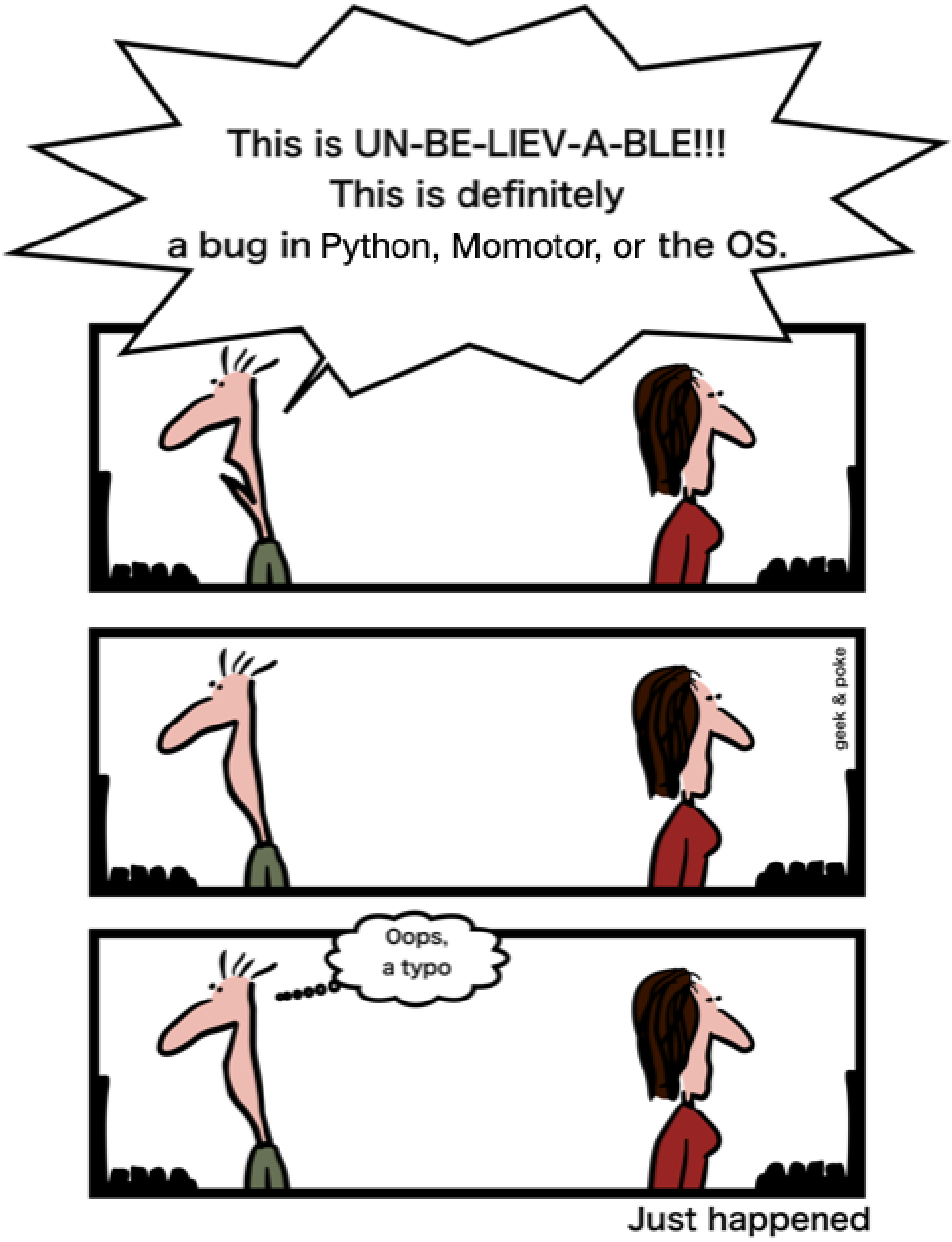
- Programming = The art and discipline of developing computer software
    - Developing = designing and implementing (writing code)

- Programming: Easy or hard?

- **Easy** in the sense that anybody
    - can get software develpment tools for free, and
    - can start creating software products
    - Unlike most other engineering disciplines

- **Hard** in the sense of developing quality software products
    - Correct, high-performance, secure, usable, adaptable

- Programming: Fun or frustrating?



Adapted by Tom Verhoeff for 2IS50

- **Fun** in the sense of
  - constructive, creative, satsifying (programmer controls computer)
- **Frustrating** in the sense of
  - computer is unforgiving
  - programming languages have quirks
  - finding mistakes takes unpredictable amount of time

I hope you will perservere ('hang in'): *Don't hesitate to ask for help*

Learning to Program (Better), The Dangers of ...

1. Not spending enough time (at least 10 hours per week needed)
   - You only learn programming by doing, trying, failing, fixing, reading (code and documentation)

1. Spending too much time (because you like it so much)
   - Ask help if you get stuck; don't just run around
   - Leave time for other things

More Python

Executing statements more than once (1)

In programs so far:

- each statement is executed *at most once*.

Such programs will not run for very long.

We need a way to execute statements more than once:

- repetition

`while` statement, also known as `while` loop

Syntax:

```
while condition:
    statement_suite
```

Semantics:

1. Evaluate `condition`.
2. When `True`, execute `statement_suite` and repeat from 1.
3. When `False`, the `while` statement is done.

In [1]:
```
# print a triangle of letters "o"
n = 4

while n > 0:
```

```
    print(n * "o")
    n = n - 1
```

```
oooo
ooo
oo
o
```

Notes

- Keep in mind that `condition` could be `False` at the start.
- Usually, a `while` loop should be designed to **terminate**.
  - The `statement_suite` needs to do something to make `condition` become `True` in the future.
- A running program can be interrupted by **Kernel > Interrupt**.
- Termination of a loop can be enforced by a `break` statement.

`break` Statement

Syntax:

```
    break
```

Semantics:

1. Exit *innermost enclosing* loop. (Outer loops will not be terminated.)
2. Proceed immediately after the exited loop.

`break` can only be used inside a loop (also `for` loop).

In [2]:
```
i = 0

# Infinite loop; abort with Kernel > Interrupt
while True:
    i = i + 1
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last
)
/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_5113/2619195605
.py in <module>
      3 # Infinite loop; abort with Kernel > Interrupt
      4 while True:
----> 5     i = i + 1

KeyboardInterrupt: 
```

In [3]:
```
i = 0

# Infinite loop, with break
while True:
```

```
    if i == 100:
        print(i)
        break
    i = i + 1
```

```
100
```

`continue` Statement

Syntax:

```
continue
```

Semantics:

1. Skip to next iteration of *innermost enclosing* loop. (Outer loops will not be affected.)
2. Proceed immediately to beginning of loop (test condition).

`continue` can only be used inside a loop (also `for` loop).

In [4]:
```
# speaking French

for char in "Hello World!":
    if char.isupper():
        continue
    print(char, end='')
```

```
ello orld!
```

`else` Clause

Syntax:

```
while condition:
    statement_suite
else:
    statement_suite
```

Semantics:

1. Execute `while` part,
2. until either condition no longer holds, or `break` encountered.
3. If loop ended normally (condition fails), execute `else` part.

- `else` clause only executed if no `break` encountered.

In [5]:
```
name   = "Roger Rabbit"
index, target = 0, ' '

while index < len(name):
    if name[index] == target:
```

```
        print("'{}' found at index {}".format(target, index))
        break
    index += 1  # augmented assignment: index = index + 1
else:
    print('Not present')
```

```
' ' found at index 5
```

Designing `while` loops

Reasoning about `while` loops:

- Thinking about what has been accomplished so far, *looking back*
- Thinking about what remains to be done, *looking ahead*

Example

Goal: Print a triangle of letters "o" with a base of length `n` (int, at least 0).

For example, output when `n == 4`:

```
oooo
ooo
oo
o
```

Looking back

Let us assume that `N` is the initial value of `n`.

When `N == 4` and `n == 2` at loop start, the output *so far* is

```
oooo
ooo
```

In general, what *has been accomplished* each time the condition is evaluated?

- All rows of sizes `i` with `n < i <= N` *have been printed*, in decreasing order.

Observe that

1. Initially, when `n == N`, this relationship holds trivially.
2. Finally, when `n == 0`, this relationship implies that the complete triangle has been printed; so, we are done.
3. When `n > 0`, printing the row with `n` times "o" and decreasing `n` by 1 will ensure that the relationship still holds.

Look ahead

When `N == 4` and `n == 2` at loop start, output *to be printed* is

```
oo
o
```

In general, what still remains to be done, each time the condition is evaluated?

- All rows of sizes `i` with `0 < i <= n` *must still be printed*, in decreasing order.

Observe that

1. Initially, when `n == N`, this relationship holds trivially.
2. Finally, when `n == 0`, this relationship implies that nothing needs to be printed; so, we are done.
3. When `n > 0`, printing the row with `n` times "o" and decreasing `n` by 1 will ensure that the relationship still holds.

Invariant (Relation)

In both cases, the reasoning is based on an **invariant relation** (**invariant** for short), such that

1. The invariant holds initially.
2. When the `while` loop terminates, the negation of the looping condition together with the invariant implies our goal.
3. When the loop makes another cycle, the invariant is preserved, but some kind of progress towards the goal was made.

Compare this to an **inductive proof** with a **base case** and **inductive step** involving an **induction hypothesis**.

Termination

One way of reasoning about termination is to come up with a *measure of progress* for the loop.

The fundamental measure of progress is based on this property:

- Every strictly decreasing chain of non-negative integer values is finite.

Phrased differently:

- An integer value that strictly decreases in every loop step will eventually become negative.

In case of printing the triangle:

- the progress measure is "the number of rows still to be printed"
- this decreases in every iteration of the `while` loop
- when equal `0`, the loop terminates

Nesting inside `while` statements

- `if` inside `while`
- `while` inside `while`

Print the numbers from 1 to 40 that are not multiples of 2 and 3.

In [6]:
```
n = 1

while n <= 40:
    if n % 2 != 0 and n % 3 != 0:
        print(n)
    n = n + 1
```

```
1
5
7
11
13
17
19
23
25
29
31
35
37
```

Print the table of multiplication (outcomes only) up to 12×12.

In [7]:
```
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
```

Executing statements more than once (2)

Another way to execute statements more than once:

1. Give them a name (in a function definition), and
2. Invoke that definition from several places.

Also called: *sharing* or *re-use*.

In [8]:
```
def print_line(n: int) -> None:
    """Print a line of n times letter "o".
    """
    print(n * "o")
```

In [9]:
```
print_line(3)
print_line(2)
print_line(1)
```

```
ooo
oo
o
```

In [10]:
```
def print_triangle(n: int) -> None:
    """Print a triangle of letters "o" of size n (n >= 0).
    """
    while n > 0:
        print_line(n)
        n = n - 1
```

In [11]:
```
print_triangle(4)
```

```
oooo
ooo
oo
o
```

In [12]:
```
def hypothenuse(a: float, b: float) -> float:
    """Compute the length of the hypothenuse
    in right-angled triangle with perpendicular sides a and b.
    """
    return (a * a + b * b) ** 0.5
```

In [13]:
```
help(hypothenuse)
```

```
Help on function hypothenuse in module __main__:

hypothenuse(a: float, b: float) -> float
    Compute the length of the hypothenuse
    in right-angled triangle with perpendicular sides a and b.
```

In [14]:
```
hypothenuse(3, 4), hypothenuse(5, 12)
```

Out[14]:
```
(5.0, 13.0)
```

Defining Functions in Python

Syntax of **function definition** (*full form*):

```
def function_name(parameter_list) -> return_type:
    """Docstring to explain purpose and assumptions.
    """
    statement_suite
```

where `parameter_list` is a comma-separated list of parameter declarations of the form

```
name: type
```

or

```
name: type = expression
```

In the *short form*, the **docstring** `"""..."""` and **type hints** ('`: type`' and '`-> return_type`') can be dropped (not recommended):

```
def function_name(parameter_list):
    statement_suite
```

where `parameter_list` is a comma-separated list of parameter declarations of the form

```
name
```

or

```
name = expression
```

The `statement_suite` is also known as the **function body**.

If `return_type` is not `None`, then the statement

```
return expression
```

must appear in the body to determine what value is returned.

In *Think Python*, functions returning a proper value are called **fruitful functions**, and otherwise they are called **void functions** (that 'return' `None`).

Semantics of function *definition*:

1. Bind the parameter list and statement suite to the function name as a *function object*.

Note that the statement suite is not (yet) executed.
Compare this to the assignment statement, but now an executable recipe (a function object) is bound to the name.

A function definition can be viewed as an **abbreviation**.

In [15]:
```
type(hypothenuse)
```

Out[15]:
```
function
```

In [16]:
```
??hypothenuse
```

Whenever a *call* to this function is made (see Recap Lecture), the statement suite is executed:

1. Initialize *parameters* from the call *arguments*.
2. Execute function's `statement_suite` until
   - either 'running off the end'
   - or a `return` statement is encountered
3. Concerning any name appearing in `statement_suite`:
   - if already bound in the static calling context, then treated as **global name**
   - otherwise, it is a **parameter** or (fresh) **local name**

Notes

- Function names follow the same rules as variable names (Recap Lecture).
- Parameter list in function definition can be *empty*.
- Python functions are not the same as mathematical functions.
  - Python functions can have *side effects*.
  - Python functions need not always return the same value for the same arguments.
- The function's statement suite can call other functions.
  - This is a kind of **function composition**.
- Python does not require type hints and docstrings
  - In this course, **type hints and docstrings are required**

Good Practices

- Document purpose and assumptions of *each* function in **docstring**.
- Indicate **types** of parameters and return value in **type hints**.
- After defining a function, **test** it by executing some calls and verifying the results.
- A function should have *one* purpose:

- **Single Responsibility Principle** (SRP).
  - A function should be relatively short.
  - Otherwise, split it into multiple functions.
  - Such splitting is know as **refactoring**.
- Avoid code duplication: **Don't Repeat Yourself** (DRY).
  - Move duplicated code to a function definition, and call it in multiple places.
- Avoid deeply nested statements.
  Rather, put an inner block in a function, and call it in the outer block.

In [17]:
```
N = 12  # table size

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
```

Local variables

Variables that are defined within a function body are **local** to that function.

They only exist while the function is active (is being executed).

In [18]:
```
x = 0  # a global variable

def f() -> None:
```

```
    x = 42   # a local variable
    print(x)

f()
print(x)   # x was not affected
```

```
42
0
```

Benefits of functions

- They can improve readability, understandability, and reasoning.
- They localize change, because duplicate code is reduced.
- They make it easier to test and debug functionality.
- They allow easy reuse of code.

Abstraction and Encapsulation

- A function can be viewed as an *abstraction*:
  You can use (call) it without knowing how it was implemented (defined).
- The docstring is all that the user and implementer need to know.
  That description **abstracts from** implementation details.

A function is said to **encapsulate** its parameters, local variables, and statement suite, insulating them from other parts of the program.

Generalization by extra parameters

A function can be made more general by including extra parameters.

In [19]:
```
?? print_line
```

In [20]:
```
def print_line_3(n: int, s: str) -> None:
    """Print a row of n times string s.
    """
    print(n * s)
```

In [21]:
```
print_line_3(3, "+")
```

```
+++
```

In [22]:
```
def print_row_2(a: int, n: int) -> None:
    """Prints row a for multiplication table of size n.
    """
    b = 1
    # invariant: columns from b through 12 need printing

    while b <= n:
```

```python
        print(" {:3}".format(a * b), end='')
        b = b + 1

    print()
```

In [23]:
```python
print_row_2(7, 10)
```

```
   7  14  21  28  35  42  49  56  63  70
```

Default arguments

The downside of extra parameters is that every call needs to provide extra arguments.

The use of **default arguments** can address this (somewhat):

In [24]:
```python
def print_line_4(n: int, s: str = "o") -> None:
    """Print a row of n times string s.
    """
    print(n * s)
```

In [25]:
```python
print_line_4(5)
print_line_4(5, '*')
```

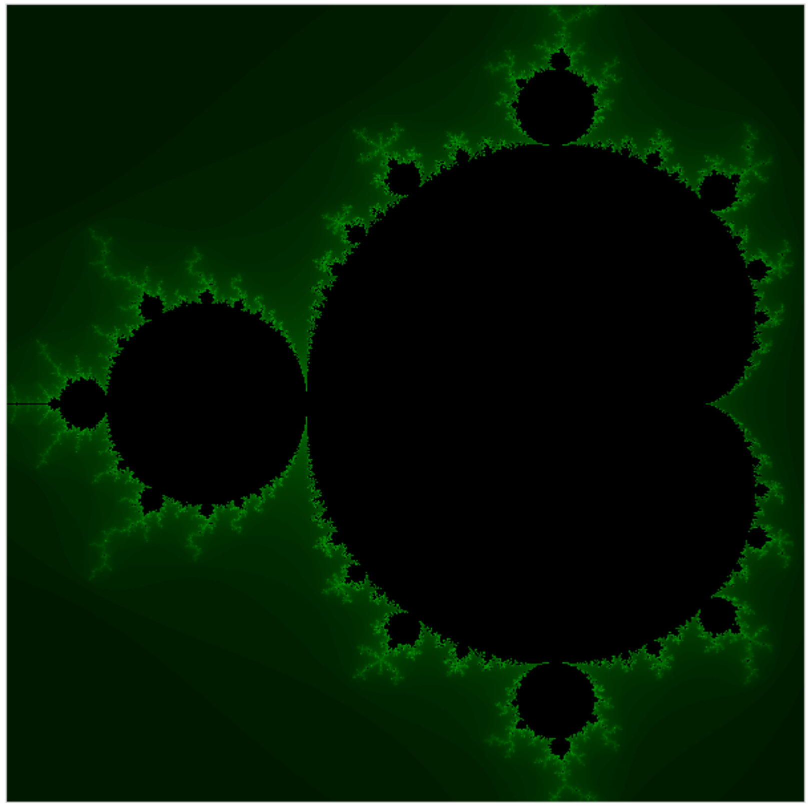
```
ooooo
*****
```

(End of Notebook)

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]:
```
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]:
```
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]:
```
%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]:
```python
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]:
```python
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]:
```
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]:
```
# 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]:
```
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

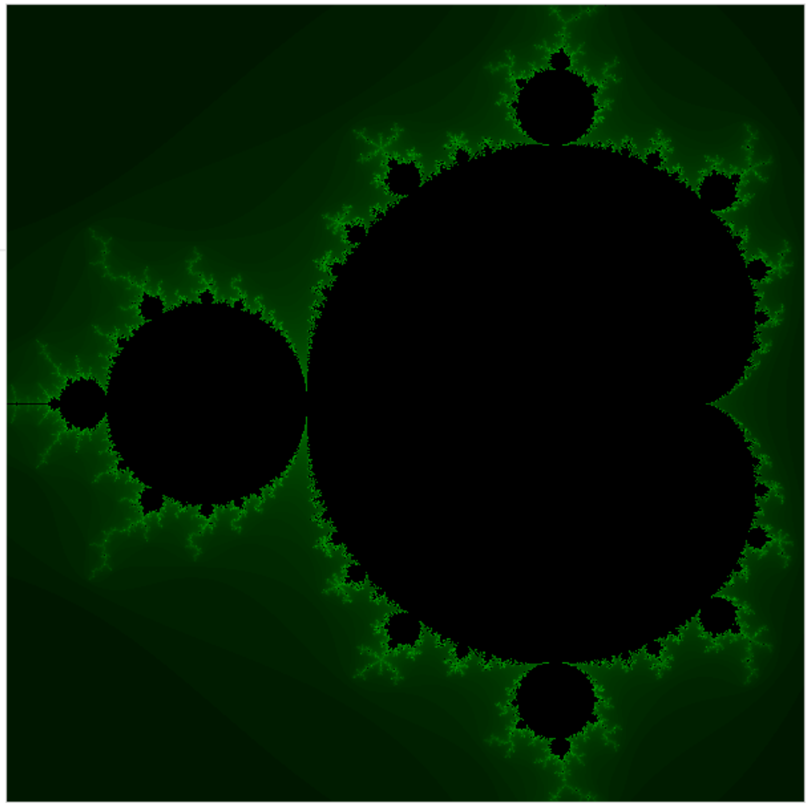
- If you did edit and get a **conflict**
    - If needed, copy edited file to outside the clone
    - Accept all incoming changes (overwrites your changes)

(End of Notebook)

2IS50 – Software Development for Engineers – 2022-2023

Lecture 2.A (Python)

Lecturer: Tom Verhoeff



Also see the book *Think Python* (2e), by Allen Downey

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.A

- Organizing data: lists and tuples.
  - Sharing, aliasing
- Anonymous functions: `lambda` expressions
- *Sequences, Iterables,* and `for`-loops
- Reading from and writing to text files

- Turtle graphics

Organizing Data

Programs manipulate data values through variables.

So far, our programs can deal with *small* amounts of *simple* data:

- integers
- floating point numbers
- booleans
- strings

In Python, integers and strings can be become very large.

Still, it is hard to encode more complex data in them.

Examples of more complex data:

- A polynomial
- A matrix or a table
- A graph with labeled nodes and labeled (un)directed edges
- Etc.

Lists & Tuples (later: Sets & Dicts)

- These are **container** or **collection** types
- They hold *multiple* items
  - Compare to mathematical sets
- Items are organized in some way
- Collections can be operated on *as a whole*:
  - inspect, query
  - modify
  - break apart
  - combine

Lists (a CS miracle)

- Each value in the `list` type is a *sequence* of values.
- **Order** and **multiplicity** (number of occurrences) are relevant.

- List literals:
  - `[]` (**empty list**)
  - `[item_1, item_2, ...]`

- The values in a sequence can have different types, including `list` (nesting).
  - `[True, [3.14, "abc"]]`

In [1]:
```
subjects = ['mathematics', 'computer science', 'programming', 'modeling']
subjects
```

Out[1]:
```
['mathematics', 'computer science', 'programming', 'modeling']
```

Standard operations on sequences

- test if non-empty: use it as boolean expression
- `len`, for length
- *indexing* (`...[...]`), for access to single item
- *slicing* (`...[...:...]`), for access to subsequence of items
- `in`, `not in`, for membership
- concatenation (`+`) and replication (`*`)
- comparisons with `==`, `!=`, `<`, `<=`, `>`, `>=` (**lexicographic order**)
- `.count(target)`, `.index(item)`

In [2]:
```
def classify(lst: list) -> None:
    """Classify a list as empty or non-empty.
    """
    if lst:
        print("{} is not empty".format(lst))
    else:
        print("it is empty")
```

In [3]:
```
classify([]), classify(subjects);  # semicolon suppresses expression resul
t
```

```
it is empty
['mathematics', 'computer science', 'programming', 'modeling'] is not empt
y
```

Anti-patterns for emptiness check

How *not* to check whether list `lst` is not empty:

- `lst != []`
- `len(lst) > 0`

How to do it:

- `lst` (when used where a boolean is expected)

More examples of list operations

In [4]:
```
len([]), len(subjects)
```

Out[4]: (0, 4)

In [5]:
```
subjects[0], subjects[3]  # indexing starts at 0
```
Out[5]: ('mathematics', 'modeling')

In [6]:
```
subjects[len(subjects)]  # this is out of bounds: IndexError
```
```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last
)
/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_19616/405106359
6.py in <module>
----> 1 subjects[len(subjects)]  # this is out of bounds: IndexError

IndexError: list index out of range
```

In [7]:
```
subjects[-1]  # negative index starts from the end
```
Out[7]: 'modeling'

In [8]:
```
subjects[1:3]  # from index 1 up to and excluding index 3
```
Out[8]: ['computer science', 'programming']

In [9]:
```
subjects[:2], subjects[2:]  # can omit start or stop
```
Out[9]: (['mathematics', 'computer science'], ['programming', 'modeling'])

In [10]:
```
subjects[:]  # omitting both start and stop copies all items
```
Out[10]: ['mathematics', 'computer science', 'programming', 'modeling']

In [11]:
```
subjects[:4:2]  # optionally include step size
```
Out[11]: ['mathematics', 'programming']

In [12]:
```
subjects[::-1]  # negative step size for reversed
```
Out[12]: ['modeling', 'programming', 'computer science', 'mathematics']

In [13]:
```
"computer science" in subjects, "data science" not in subjects
```
Out[13]: (True, True)

In [14]:
```
subjects + 4 * ["fun"]
```
Out[14]:
```
['mathematics',
 'computer science',
 'programming',
 'modeling',
 'fun',
 'fun',
 'fun',
```

```
 'fun']
```

In [15]:
```
subjects.count("programming"), subjects.count("informatics")
```
Out[15]:
```
(1, 0)
```

In [16]:
```
subjects.index("mathematics")  # returns index where target occurs
```
Out[16]:
```
0
```

In [17]:
```
subjects.index("informatics")  # ValueError if not present
```
```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call last
)
/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_19616/300904457
3.py in <module>
----> 1 subjects.index("informatics")  # ValueError if not present

ValueError: 'informatics' is not in list
```

Lists are *mutable*

Use indexed or sliced list as target in assignment statement to modify it.

In [18]:
```
subjects[1] = "informatics"
subjects
```
Out[18]:
```
['mathematics', 'informatics', 'programming', 'modeling']
```

Other ways of modifying a list:

- `.append(item)` (appends item at end)
- `.extend(lst)` (appends all items from lst at end)
- `.clear()` (removes all items)
- `.pop()` (removes last item)
- `.remove(item)` (removes given item)
- `.reverse()` (reverses list)
- `.sort()` (sorts list)
- Use **TAB for code completion** and **SHIFT TAB for documentation**

In [19]:
```
subjects. # TAB-complete to subject.sort, then add parentheses: subjects.s
ort()
subjects
```
```
  File "/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_19616/2
581254541.py", line 1
    subjects. # TAB-complete to subject.sort, then add parentheses: subjec
ts.sort()
              ^
SyntaxError: invalid syntax
```

Map, Filter, Reduce

Very common operations on lists (actually, on *iterables*) are:

- **map**: apply a given function to each item of a given list
- **filter**: from a given list, select those items for which a given function returns `True`
- **reduce**: combine all items in a given list using a given binary operator E.g. `[1, 2, 3, 4] ->  1 + 2 + 3 + 4`

Python Standard Library - built-in functions

- `map(func, lst)` returns iterable with `func` applied to each item in `lst`
- `filter(func, lst)` returns iterable with items from `lst` for which `func` returns `True`
- `functools.reduce(op, lst)`, where `op` is function with 2 arguments (binary operator), returns result of evaluating **accumulation** expression `lst[0] op lst[1] op ... op lst[-1]`
- `functools.reduce(op, lst, initial)` first prepends `initial` to make list non-empty

Notes

- These operations are *lazy* and do not return a list object
- Turn result of `map` and `filter` into list by applying `list`.

In [20]:
```python
map(len, subjects)  # result is a map-object
```

Out[20]:
```
&lt;map at 0x7faa7841b970&gt;
```

In [21]:
```python
list(map(len, subjects))
```

Out[21]:
```
[11, 11, 11, 8]
```

Anonymous functions: `lambda` expressions

Syntax of **`lambda` expression**:

```python
lambda parameter_list: expression
```

Here, `lambda` is a reserved word (name of the Greek letter λ).

Semantics:

1. Behave as function `f` defined by

```python
def f(parameter_list):
    return expression
```

except that the function is not given any name.

In [22]:
```python
list(filter(lambda s: len(s) == 11, subjects))
```

Out[22]: ['mathematics', 'informatics', 'programming']

In [23]:
```python
from functools import reduce

reduce(lambda s, t: s + " | " + t, subjects)
```

Out[23]: 'mathematics | informatics | programming | modeling'

Notes for `lambda` expressions

- In general, prefer `def` to define functions

  Advantages:
  - They have a name
  - Easier to add type hints
  - Can add docstring
  - Easier to reuse in more places
- Advantage of `lambda` expression: concise
- *Avoid* assigning `lambda` expression to name
  - Even though it works
  - In that case, use `def`

In [24]:
```python
from functools import reduce  # all imports will be given in Final Test

operator = lambda s, t: s + " | " + t  # AVOID THIS

reduce(operator, subjects)
```

Out[24]: 'mathematics | informatics | programming | modeling'

List Comprehension

Functions `map` and `filter` are nice to construct lists.

Even nicer is *list comprehension*, with syntax

```python
[ expression for name in iterable ]

[ expression for name in iterable if condition ]
```

Semantics:

- Construct the list consisting of
  - all `expression` values obtained by
  - iterating `name` over the `iterable`,
  - optionally where `name` satisfies the `condition`.

In [25]:
```
[s.capitalize() for s in subjects if len(s) == 11]
```
Out[25]:
```
['Mathematics', 'Informatics', 'Programming']
```

In [26]:
```
# Using map and filter, with lambda expressions for same result

list(map(lambda s: s.capitalize(), filter(lambda s: len(s) == 11, subjects
)))
```
Out[26]:
```
['Mathematics', 'Informatics', 'Programming']
```

List comprehension and loops

Alternatives to construct lists, from more preferred to less preferred:

1. list comprehension
2. `map` - `filter` - `reduce`
3. `for` -loop with *accumulation* variable

In [27]:
```
result = []

for s in subjects:
    if len(s) == 11:
        result.append(s.capitalize())

result
```
Out[27]:
```
['Mathematics', 'Informatics', 'Programming']
```

Disadvantages of `for` -loop for this purpose:

- You need to choose a name for the resulting list
- Takes more code (more opportunity to make mistakes)
- Less readable and understandable

Tuples

Tuples are like lists, but **immutable**

Tuple literals:

- `()` (**empty tuple**)
- `(item, )` (tuple with one item, **comma required**)
- `(item_1, item_2, ...)` (tuple consisting of given items in given order)

In many places, the parentheses can be omitted.

Comma *after last item* is acceptable.

In [28]:
```python
n, a, b = 0, 0, 1  # short for (n, a, b) = (0, 0, 1)
# invariant: a, b == fib(n), fib(n+1) for n >= 0

while n != 100:
    n, a, b = n + 1, b, a + b  # next fibonacci number

print("fib({}) == {}".format(n, a))
```

```
fib(100) == 354224848179261915075
```

Trade-offs between lists and tuples

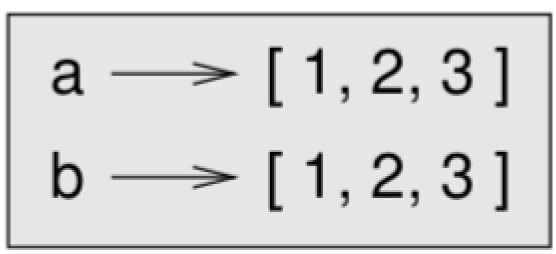
- **Flexibility**: lists are mutable, tuples are immutable
- **Performance**: lists have more overhead than tuples
    - Lists take up more memory space
    - List operations are slower

Flexibility (mutability) has a drawback:

- Complicates reasoning and understanding: **aliasing**

Aliasing

- An assignment statement binds a **name** to an **object**. We say: the variable **references** the object.
- An object has a **value**.
- Two *different* names can be bound to the *same* object.
- Two *different* objects can have the *same* value.
- This matters when modifying object values.



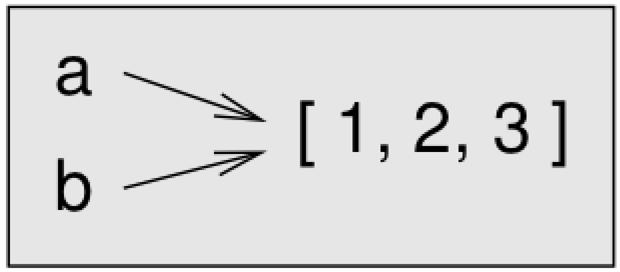
In [29]:
```python
a = [1, 2, 3]
b = [1, 2, 3]

a == b, a is b  # compare values, references; no aliasing
```

Out[29]:
```
(True, False)
```

In [30]:
```python
a[0] = 0  # modify a

a, b  # only a has changed
```

Out[30]: ([0, 2, 3], [1, 2, 3])

Aliasing: when the *same object* is know by *different names*



In [31]:
```
a = [1, 2, 3]
b = a

a == b, a is b  # a and b are aliases
```

Out[31]: (True, True)

In [32]:
```
a[0] = 0  # modify a

a, b  # both a and b have changed
```

Out[32]: ([0, 2, 3], [0, 2, 3])

Since tuples and strings are immutable, aliasing never causes problems.

Aliasing (sharing) can help improve memory efficiency.

"There are two kinds of programmers"

- Those that have been bitten by aliasing
- And those that will be

One of the quirks of imperative programming with mutable types

Adapted from: "There are two kinds of computer users"

- Those that have lost data
- And those that will

So, do make (offsite) backups!

Strings

- A string is a *sequence* of characters
- Python has no type for single characters. Use a string of length 1.
- Strings are **immutable**. Appending to a string is expensive (involves copying).

- Strings support standard sequence operations:
  - `len`
  - indexing and slicing
  - `in`
  - `.count(target)`

- String-specific operations:
  - `.lower()`, `.upper()`, `.capitalize()`
  - `.find(target)`, `.format(...)`, `.split(delimiters)`, `.join(list)`
  - Use **TAB for code completion** and **SHIFT TAB for documentation**

In [33]:
```
"abc".  # use TAB after . for code completion
```

```
  File "/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_19616/723778573.py", line 1
    "abc".  # use TAB after . for code completion
            ^
SyntaxError: invalid syntax
```

In [34]:
```
", ".join(subjects)
```

Out[34]:
```
'mathematics, informatics, programming, modeling'
```

Intermezzo: `join` versus `for`-loop

- `for`-loop to construct string is *inefficient*

In [35]:
```
result, separator = "", ""

for subject in subjects:
    result += separator + subject  # INEFFICIENT (because of copying)!
    separator = ", "

result
```

Out[35]:
```
'mathematics, informatics, programming, modeling'
```

In [36]:
```
def capitalize_words(words: str) -> str:
    """Capitalize each word in str, and compress whitespace.
    """
    return ' '.join([word.capitalize() for word in words.split()])
```

In [37]:
```
lines = ['abc def', 'hij   klm']
lines
```

Out[37]:
```
['abc def', 'hij   klm']
```

In [38]:
```
[capitalize_words(line) for line in lines]
```

Out[38]: ['Abc Def', 'Hij Klm']

Sequences and Iterables

- List, tuple, and string objects are *sequences*
    - They support `len`, indexing, slicing
- `map`, `filter`, and `reduce` objects are *not* sequences

Duck typing

> "If it walks like a duck and it quacks like a duck, then it must be a duck"

Dynamic typing: based on supported operations, rather than birth

Static typing: based on declaration

In [39]:
```
len(map(len, subjects))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last
)
/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_19616/579403691
.py in <module>
----> 1 len(map(len, subjects))

TypeError: object of type 'map' has no len()
```

Iterable

- An *iterable* is any object that can yield items one after another
    - ... is any object that can be used in a `for`-loop: `for item in iterable`
- Also see: [Python Like You Mean It: Iterables]
- Sequences are iterables
- `map` and `filter` objects are iterables

Recap: `for` statement, a.k.a. `for`-loop

Syntax:

```
for item in iterable:
    statement_suite
```

Semantics: Successively,

- assign each value yielded by the *iterable* to *item*,
- execute the statement suite.

```
In [40]: for item in map(len, subjects):
             print(item)
```

```
11
11
11
8
```

Notes for `for`-loops

- The iterable could be *empty*, in which case the statement suite is never executed.
- The iterable can be a string, tuple, or list.
- If the iterable is *finite*, then the `for` loop is guaranteed to *terminate*.
- **WARNING: NEVER change the iterable while iterating over it with a `for` loop!**
- You can use `break` and `continue` statements inside a `for` loop. Not recommended, however

```
In [41]: for letter in 'abstraction':
             if letter == 'i':
                 break
             print(letter.upper(), end=' ')
```

```
A B S T R A C T
```

Type hints for collections

- `List`, `Tuple`, `str`
- `Sequence`, `Iterable`

```
In [42]: from typing import List, Tuple, Sequence, Iterable, Any  # these will alwa
         ys be given

         Tuple[float, float]  # tuple with exactly two floats
         Tuple[str, ...]  # tuple with variable number of strings

         List[Any]  # list with objects of any type

         Sequence[int]  # sequence with integers
         Iterable[str]  # iterable yielding strings
```

```
Out[42]: typing.Iterable[str]
```

Advice: In function `def`, prefer *more general* types for parameters

- Preference order: `Iterable`; `Sequence`; `List`, `Tuple`, `str`

Advice: In function `def`, prefer *more concrete* type for return value

Ranges

- `range(n)` is an iterable yielding `n` integer values 0 through `n-1`. N.B. `n` is not in the range; rather, it is the length of the range.
- `range(m, n)` is iterable yielding integer values `m` through `n-1`.
- `range(m, n, s)` is an iterable with the integer values `m`, `m+s`, `m + 2*s`, ... through `n-1`. `s` is the *step size*.
- Compare to slicing: `lst[start:stop]`, `lst[start:stop:step]`

Property:

- If `a <= b <= c`, then `range(a, b)` followed by `range(b, c)` equals `range(a, c)`.

Advice: Use `range` in `for`-loops sparingly

- There are often better solutions
- Using `range` in a comprehension can be acceptable

In [43]:
```
# print first 10 squares, starting at 0

for n in range(10):
    print(n * n)
```

```
0
1
4
9
16
25
36
49
64
81
```

In [44]:
```
# print a triangle of letters "o"

for n in range(3, 0, -1):
    print(n * "o")
```

```
ooo
oo
o
```

More built-in iterables

- `enumerate(iterable)`: yield pairs `(index, item)`
- `zip(iterable1, iterable2, ...)`: yield tuples `(item1, item2, ...)`
- `reversed(sequence)`: yield items in reversed order

- `sorted(iterable)`: yield items in sorted order

In [45]:
```
word = 'Alphabet'

for index in range(len(word)):  # NOT RECOMMENDED!
    item = word[index]
    print(index, item)
```

```
0 A
1 l
2 p
3 h
4 a
5 b
6 e
7 t
```

In [46]:
```
for (index, item) in enumerate('Alphabet'):
    print(index, item)
```

```
0 A
1 l
2 p
3 h
4 a
5 b
6 e
7 t
```

In [47]:
```
# enumerate() yields tuples
# These tuples are "unpacked" into (index, item)
# Here, one can omit parentheses around tuple

for index, item in enumerate('Alphabet'):
    print(index, item)
```

```
0 A
1 l
2 p
3 h
4 a
5 b
6 e
7 t
```

`enumerate` also works for *iterables*, where indexing would be impossible

Note its *second* parameter, where numbering starts (default is 0):

In [48]:
```
for index, item in enumerate(map(lambda s: s.capitalize(), subjects), 1):
    print(index, item)
```

```
1 Mathematics
2 Informatics
3 Programming
```

`zip` to repack multiple iterables

In [49]:
```
for a, b, c in zip("ABCDE", "abcde", "12345"):
    print(a, b, c)
```

```
A a 1
B b 2
C c 3
D d 4
E e 5
```

`zip` as matrix transpose

In [50]:
```
matrix = [[1, 2, 3], [4, 5, 6]]
for row in matrix:
    print(row)
```

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

In [51]:
```
for row in zip(*matrix):   # * will be treated later
    print(row)
```

```
(1, 4)
(2, 5)
(3, 6)
```

Reducing iterables

- `sum(iterable)`
- `max(iterable)` `max(iterable, default)` : use `default` if `iterable` empty
- `min(iterable)` `min(iterable, default)` : use `default` if `iterable` empty
- `all(iterable)` : items interpreted as `bool` ("for all")
- `any(iterable)` : items interpreted as `bool` ("there exists")

In [52]:
```
word = "Mississippi"
min(word), max(word)
```

Out[52]:
```
('M', 's')
```

Reading and writing text files

- `open(file_path)` and `open(file_path, 'r')` open a text file for **reading**
- `open(file_path, 'a')` opens a text file for **appending**
- `open(file_path, 'w')` opens a text file for **writing WARNING: Writing is destructive !!!**

Use the `with` statement to work with files.

NOTE: The book *Think Python* doesn't do this, but it should have done so!

- `with` is a *context manager* that will properly close the file after using it, even when errors have occurred.

In [53]:
```
with open('2IS50-2223-Lecture-2-A.ipynb') as f:  # open file for reading
    lines = f.readlines()[8:20]  # extract some lines

lines
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last
)
/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_19616/618206233
.py in <module>
----> 1 with open('2IS50-2223-Lecture-2-A.ipynb') as f:  # open file for r
eading
      2     lines = f.readlines()[8:20]  # extract some lines
      3 
      4 lines

FileNotFoundError: [Errno 2] No such file or directory: '2IS50-2223-Lectur
e-2-A.ipynb'
```

A Jupyter notebook is basically a complex structured piece of data, encoded in a text file.

An file opened for reading is an *iterable*:

- it yields its lines as items
- N.B. these lines include the newline character `'\n'` at the end

In [54]:
```
with open('2IS50-2223-Lecture-2-A.ipynb') as f:  # open file for reading
    print(sum(1 for _ in f))  # count number of lines
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last
)
/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_19616/323499559
.py in <module>
----> 1 with open('2IS50-2223-Lecture-2-A.ipynb') as f:  # open file for r
eading
      2     print(sum(1 for _ in f))  # count number of lines

FileNotFoundError: [Errno 2] No such file or directory: '2IS50-2223-Lectur
e-2-A.ipynb'
```

NOTE: We used `_` (underscore) as 'anonymous' variable

Its value is used nowhere

Turtle Graphics

See Chapter 4 of *Think Python*.

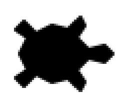
Also see: [The Beginner's Guide to Python Turtle] on *Real Python*

(A first encounter with recursive functions)

In [55]:
```
import turtle
```

In [56]:
```
t = turtle.Turtle()
t.shape("turtle")
```

A new window should have opened showing the following:

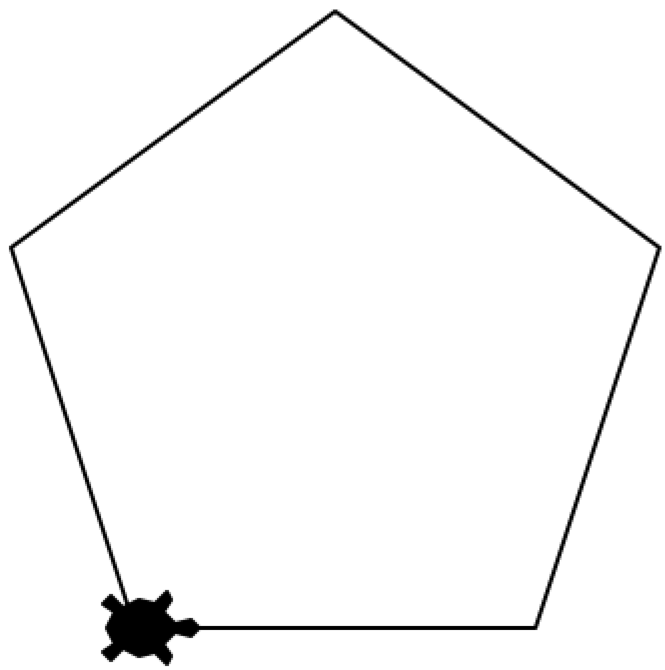


In [57]:
```
def n_gon(n: int, size: int=100) -> None:
    """Draw a regular n-gon with given side lengths.
    """
    i = n

    while i > 0:
        t.forward(size)
```

```
        t.left(360 / n)
        i = i - 1
```

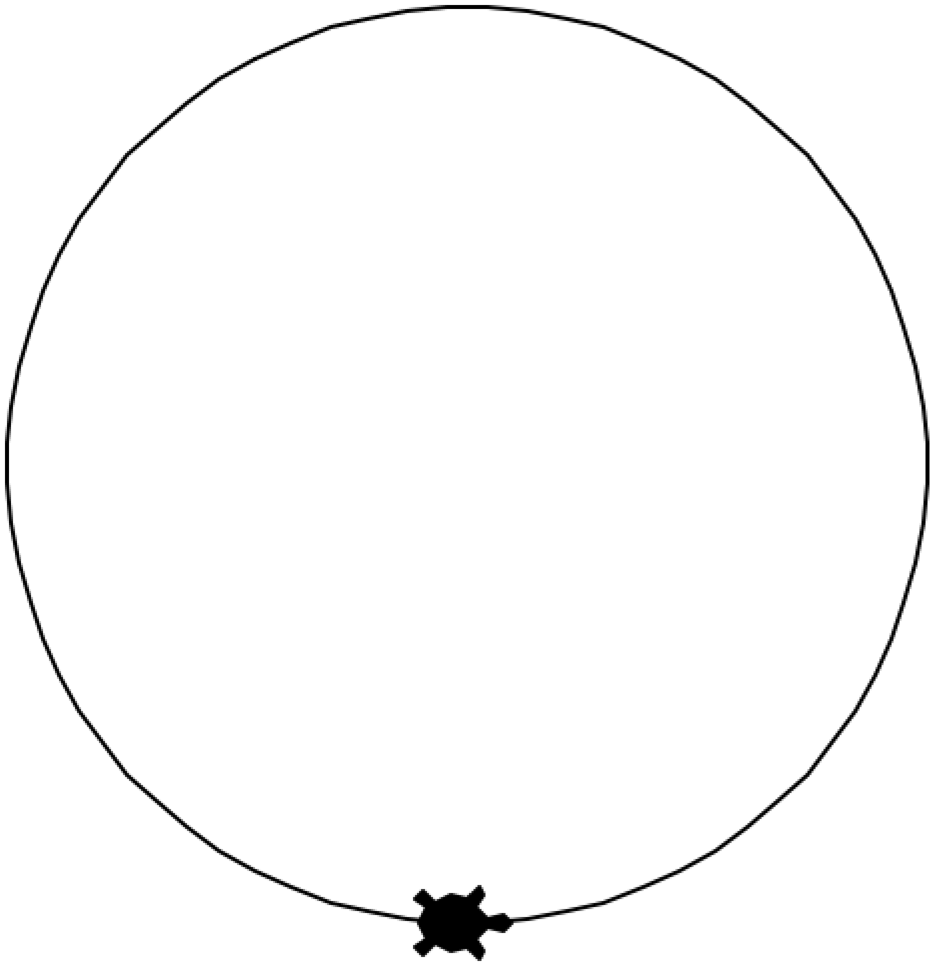
In [58]:
```
n_gon(5)
```

Expected output in turtle window:



In [59]:
```
t.reset()

n_gon(72, 10)
```

Expected output in turtle window (a 72-gon looks like a circle):

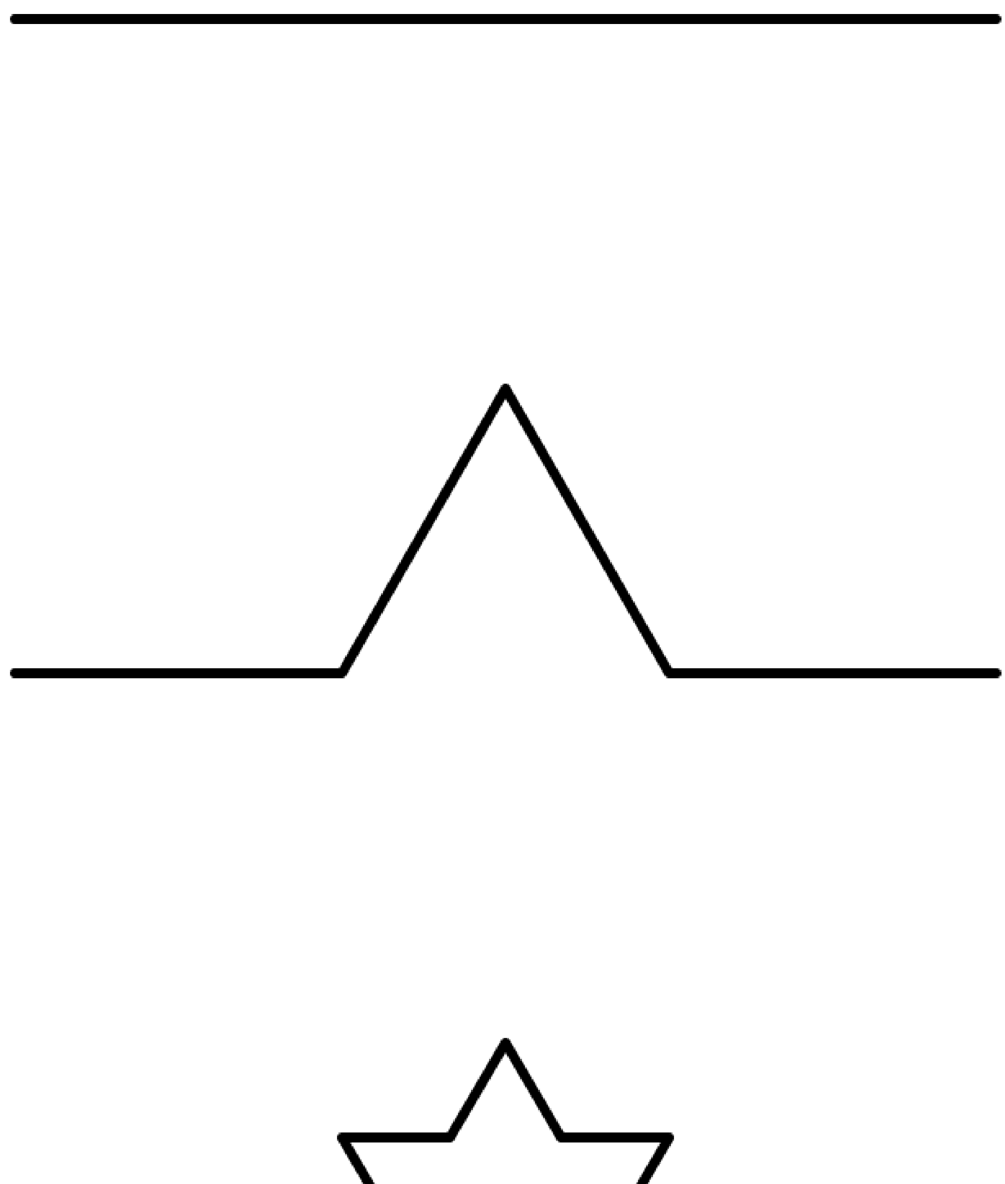


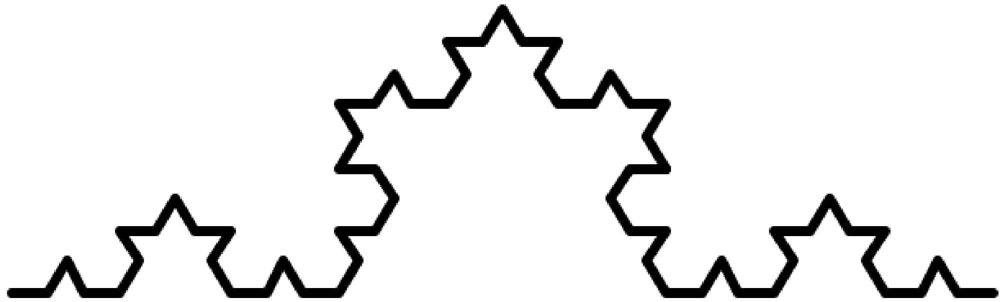
In [60]:
```
def koch(t: turtle.Turtle, n: int, size: float = 100) -> None:
    """Draw a Koch fractal with n generations of given size.
    """
    t.pendown()
    if n == 0:
        t.forward(size)
    else:
        koch(t, n - 1, size / 3)
        t.left(60)
        koch(t, n - 1, size / 3)
        t.right(120)
        koch(t, n - 1, size / 3)
        t.left(60)
        koch(t, n - 1, size / 3)
```

`koch` is a recursive function: its body contains calls to itself.

In [61]:
```
t.reset()
t.width(3)
t.penup()
t.goto(-turtle.screensize()[0] + 10, turtle.screensize()[1] - 10)
```

Expected output in turtle window:





In [62]:
```
for i in range(4):
    koch(t, i, 300)

    t.penup()
    t.right(90)
    t.forward(200)
    t.left(90)
    t.backward(300)
```

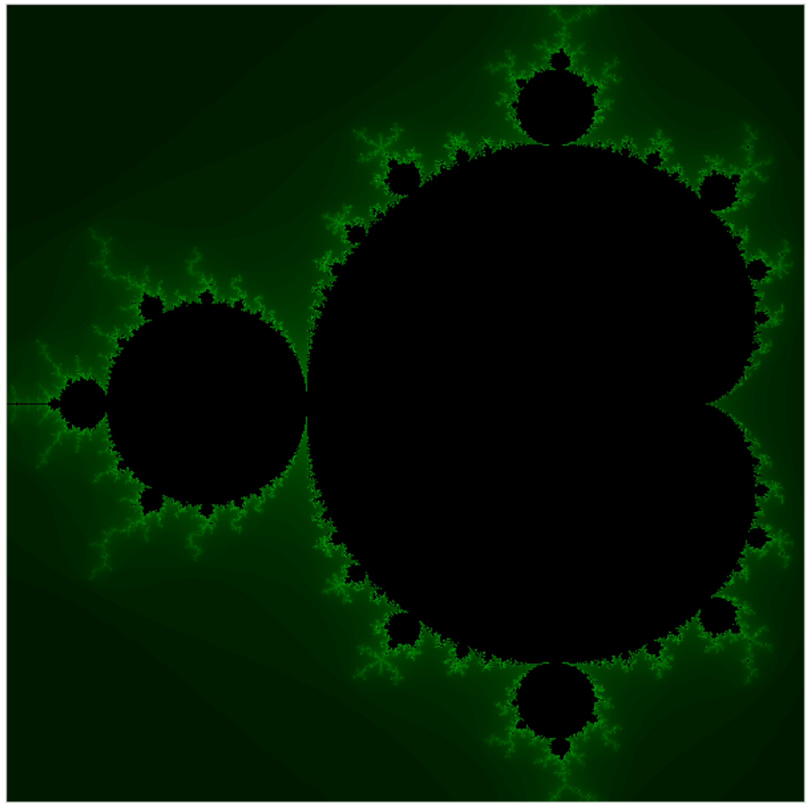
In [63]:
```
turtle.done()
# now close window with turtle canvas
```

(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]:
```
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]:
```
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]:
```
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]:
```
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]:
```
import random
from typing import List
```

In [12]:
```
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]:
```
# 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]:
```
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]:
```
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]:
```
doctest.run_docstring_examples(rps_choice, globals(), verbose=False, name=
'rps_choice')
```

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

In [20]:
```
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]:
```
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]:
```
beats_test = """
    >>> beats(2, 2)  # was missed above!
    False
"""
```

In [23]:
```
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]:
```
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*

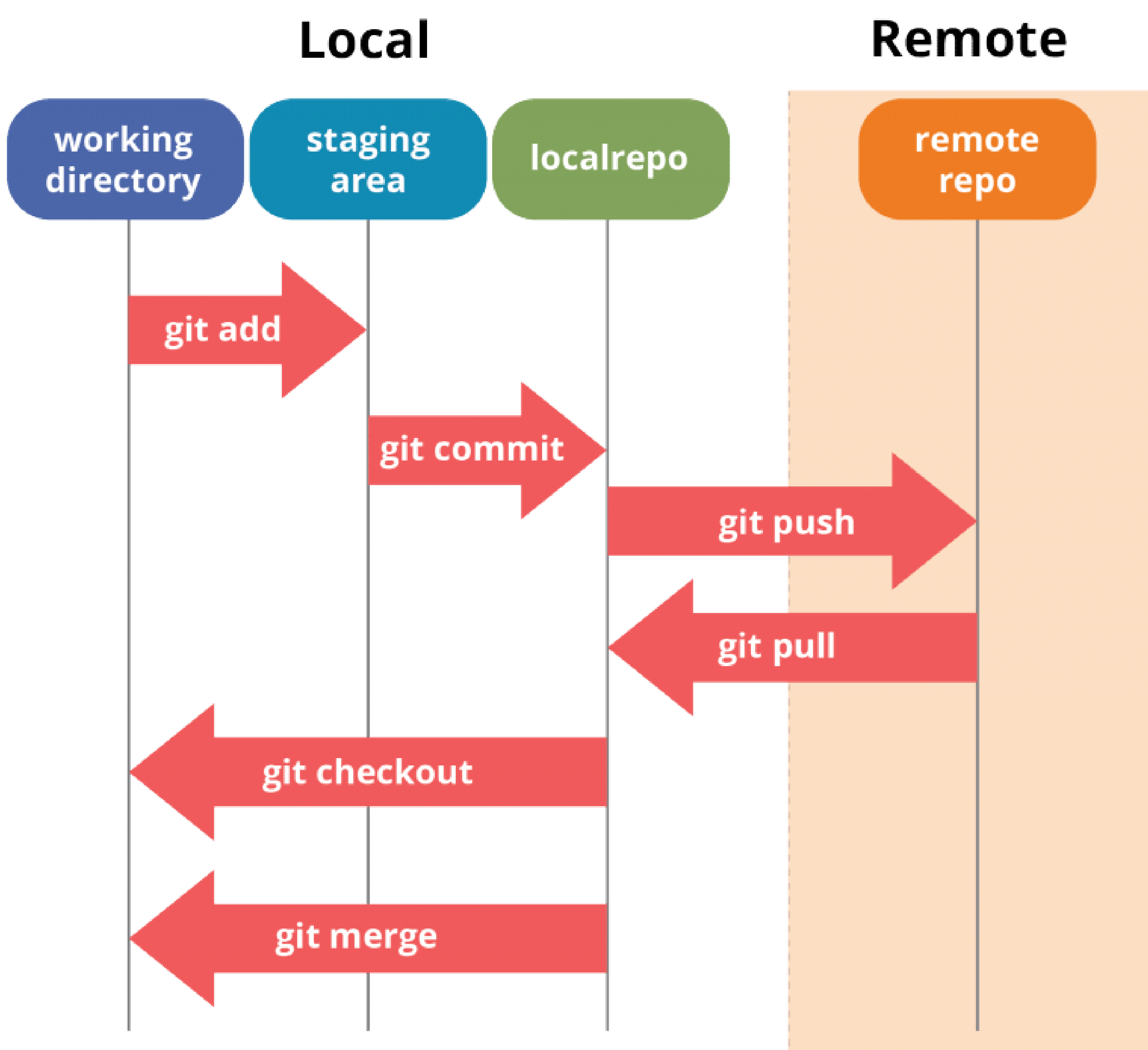
Version Control with Git

- 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)

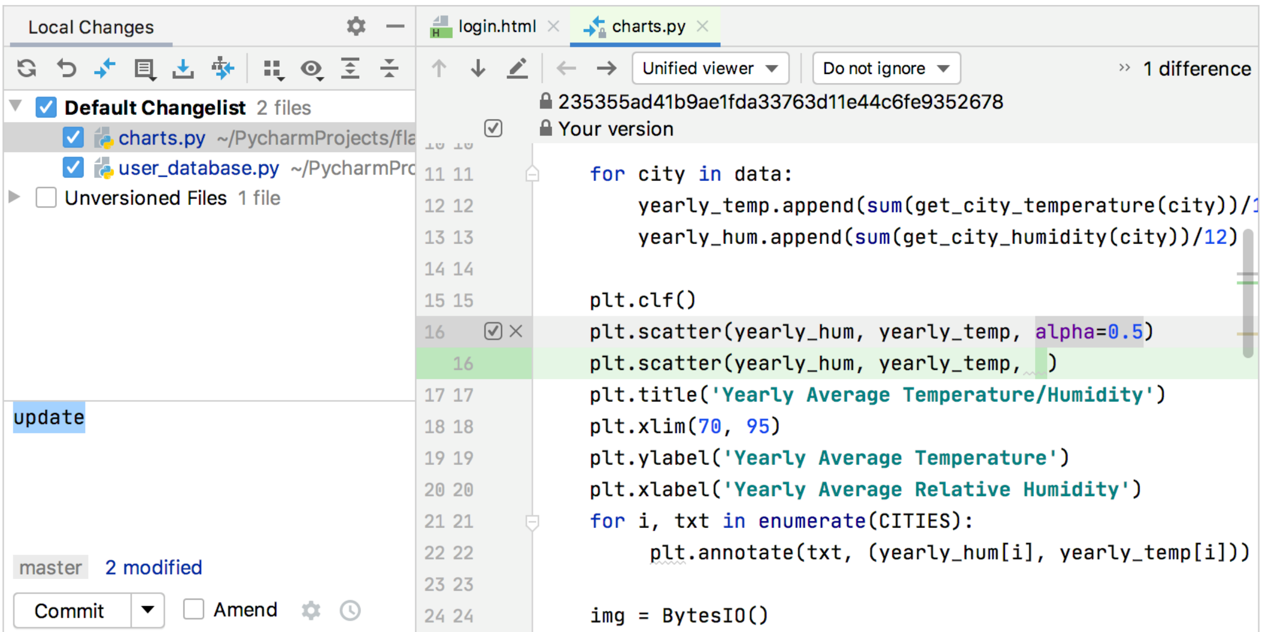


Source: https://www.edureka.co/blog/git-tutorial/

PyCharm manages the staging area

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

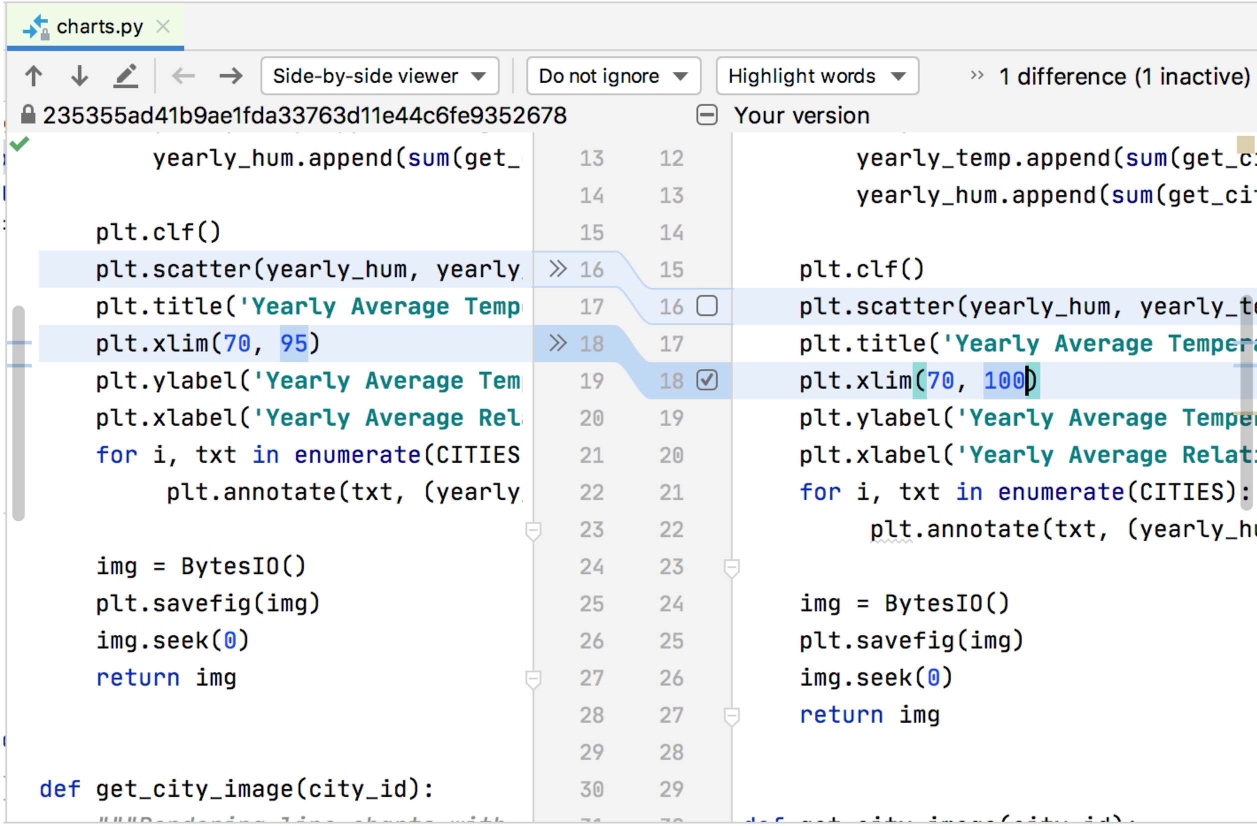
PyCharm: select files to commit in the *Git tool window*



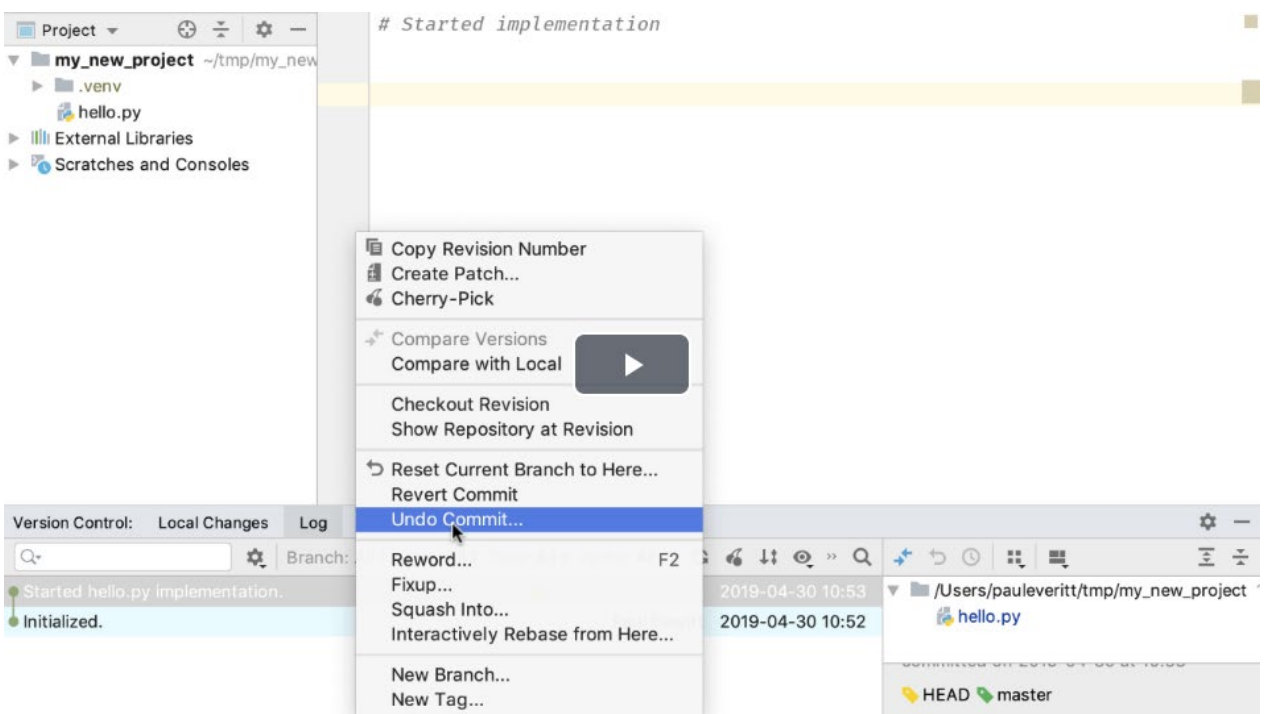
See: https://www.jetbrains.com/help/pycharm/commit-and-push-changes.html?section=Windows%20or%20Linux#

PyCharm: *Partial* commits

Select changes to commit per *chunk*



PyCharm: Undo last commit (if not yet pushed)



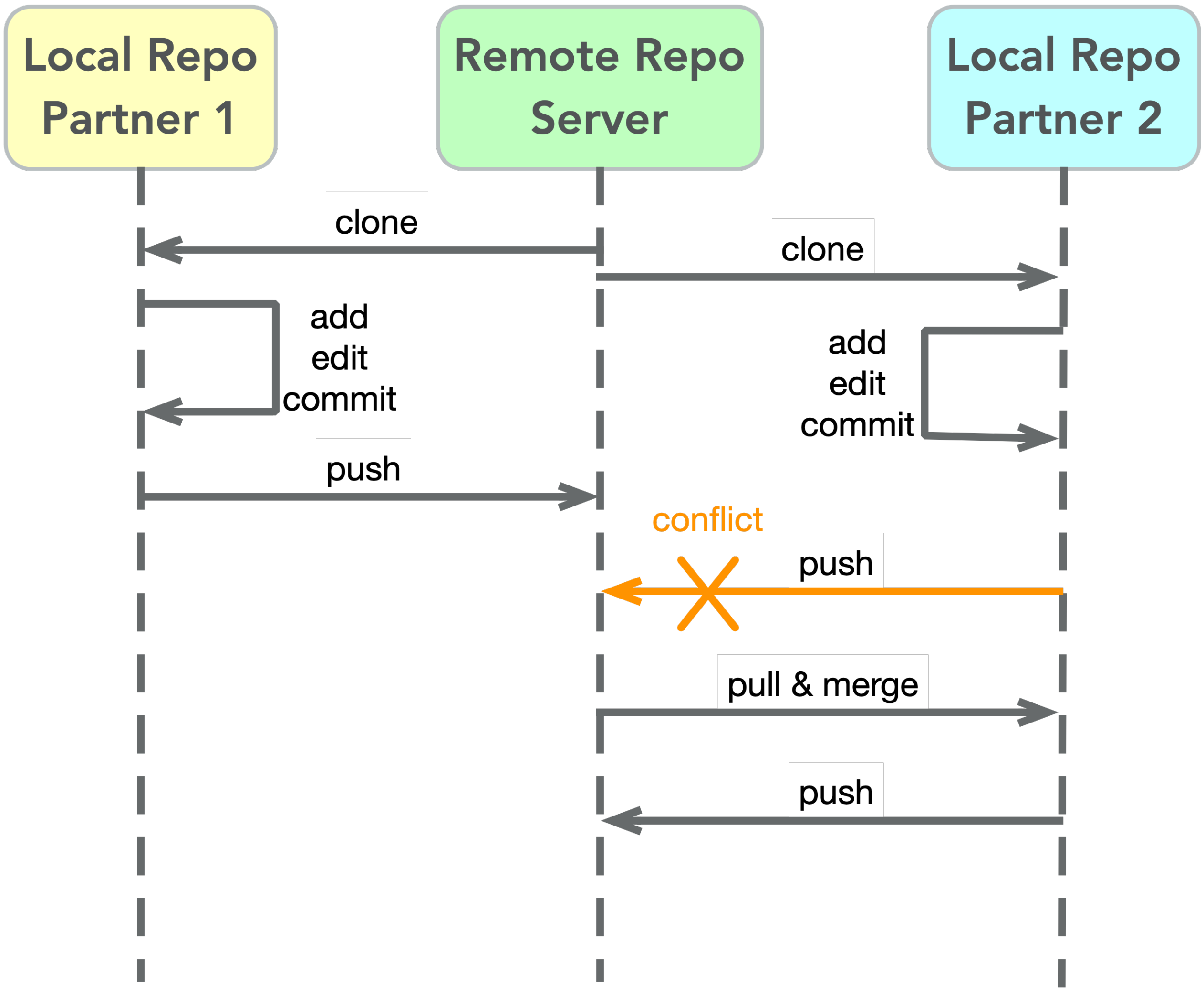
Video: https://www.jetbrains.com/pycharm/guide/tips/undo-last-commit/

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

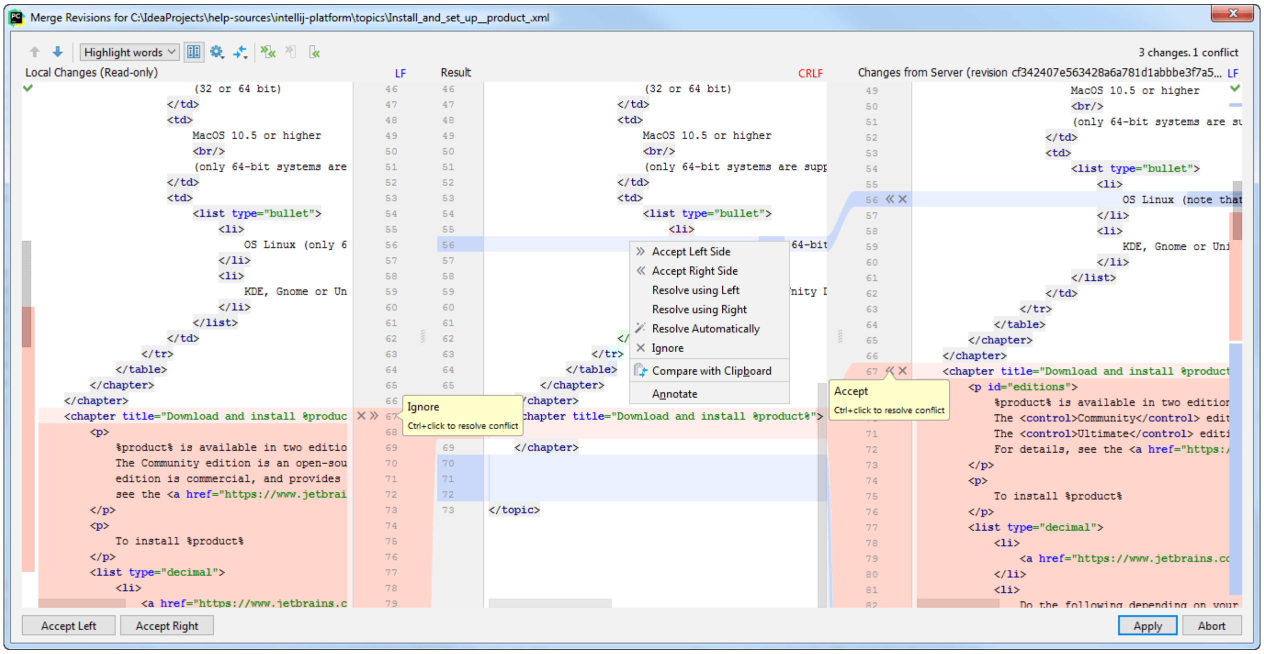
- Reason: Your local repo is out-of-sync with remote repo
- Someone else already pushed new changes that you miss

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)

2is50-2223-lecture-3-a

June 16, 2023

1 2IS50 – Software Development for Engineers – 2022-2023

1.1 Lecture 3.A (Python)

Lecturer: Tom Verhoeff

Also see the book ***Think Python*** (2e), by Allen Downey

1.1.1 Review of Lecture 2.A

- Organizing data: lists and tuples.
  - Sharing, aliasing
- Anonymous functions: `lambda` expressions
- *Sequences, Iterables,* and `for`-loops
- Reading from and writing to text files
- Turtle graphics

1.1.2 Preview of Lecture 3.A

- Algorithms and data structures
- Organizing data: sets (`set`) and dictionaries (`dict`)
  - other `collections`: `defaultdict`, `Counter`
- Avoid unnecessary computation and storage
  - Generator expressions, generator functions

1.2 Algorithms and Data Structures

Data related to the problem domain * must be *stored* efficiently, and

- *manipulated* efficiently
  - read, inspect, query
  - write, modify, update

Data structure: way of *organizing* data

- For a data type, it not only matters what its possible *values* are.
- It also matters what *operations* it supports,
  and how *efficient* it is (time and memory).

[ ]:
```python
from typing import Any, Tuple, List, Sequence, Iterable
```

Types `list` and `tuple` are very similar, but not the same:

Aspect Type	`tuple`	`list`
Values	sequence	sequence
Operations		
Length	Yes	Yes
Indexing, slicing	Yes	Yes
Iteration	Yes	Yes
…	…	…
Mutability	immutable	mutable
Speed	faster	slower
Memory	less	more

[ ]:
```python
from sys import getsizeof

getsizeof((1, 2, 3, 4, 5)), getsizeof([1, 2, 3, 4, 5])
```

[ ]:
```python
%%timeit -c

(1, 2, 3, 4, 5)
```

[ ]:
```python
%%timeit -c

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

Algorithm: way of using data structures to solve (computational) problems

More powerful data structures, means simpler algorithms

- This course does not focus on algorithm design
- Python offers powerful data structures
  - Choose and use wisely
  - Get to know them well

1.3 Sets

- Each value of the `set` type is a *set* of *values*
- Values in a set can have different types, including `tuple`
- Values in a set must be *hashable* (not everything can be in a set)
  - Hashable is roughly the same as immutable
- **Order** and **multiplicity** are *not* relevant
- Set literals:
  - `set()` (**empty set**; N.B. **cannot use `{}`**)
  - `{item_1, item_2, ...}` (set consisting of given items)
- Sets are **mutable** (`frozenset` is immutable)

[ ]:
```python
from typing import Set
```

[ ]:
```python
s: Set[int] = {1, 3, 1, 2}
s
```

1.3.1 Standard operations on set `s`

- `s` (test if non-empty: use it as boolean expression)
- `len(s)` (size of `s`)
- sets are *not* indexable
- `e in s` and `e not in s` (membership test)
- `for e in s` (iteration)
- `set(iterable)` (convert iterable to set)
- `{expr for i in iterable if condition}` (set comprehension)

[ ]:
```python
s.  # Hit TAB key for code completion; then SHIFT-TAB for documentation
```

1.3.2 Operations that don't modify sets

- `s.union(t)`, `s.intersection(t)`, `s.difference(t)`, `s.symmetric_difference(t)`
- `s.issubset(t)`, `s.issuperset(t)`, `s.isdisjoint(t)`
- Can also use `|`, `&`, `-`, `^`, `==`, `!=`, `<`, `<=`, `>`, `>=`

1.3.3 Operations that modify sets

- `s.add(e)`, `s.discard(e)`
- `s.update(t)`, `s.xxx_update(t)`
- `s.pop()` (take arbitrary element from *non-empty* set; removes it)

1.4 Dictionaries (Dicts)

- Each value of the `dict` type is a *mapping* from *keys* to *values*
  - like a labeled set: keys are elements, values are labels
  - like a mathematical function: a set of key-value pairs
  - a.k.a. *associative array*, or *association*
- Keys can be of any *hashable* type, like `int` and `str`
- **Order** and **multiplicity** of *keys* are *not* relevant
- Dictionary literals:
  - `{}` or `dict()` (**empty dictionary**)
  - `{key_1: value_1, key_2: value_2, ...}` (dictionary consisting of given key-value pairs)
- Dictionaries are **mutable**

[ ]:
```python
from typing import Dict
```

[ ]:
```python
d: Dict[str, int] = {'a': 1, 'b': 3, 'c': 1, 'b': 2}
d
```

1.4.1 Standard operations on dictionary `d`

- `d` (test if non-empty: use it as boolean expression)
- `len(d)` (size of `d`, i.e. number of keys)
- `key in d` and `key not in d` (key membership test)
- `d[key]` (get value for key; raises `KeyError` if not present)
- `for key in d` (iteration over *keys*.)
- `dict(args)` (convert args to dictionary, if applicable)
- `{key_expr: value_expr for i in iterable if condition}` (dictionary comprehension)

1.4.2 Counting things (1)

[ ]:
```
word = "MISSISSIPPI"

counts = {letter: word.count(letter) for letter in word}
counts
```

Note: The code above to count letters in a word is *inefficient*. Why?

- Run the code again with `10000 * MISSIPPI` (be prepared to wait almost 10 s)
- For every letter of the word, the entire word is scanned again.
  - Thus: runtime is **quadratic** in length of word
- The same letter is counted multiple times (previous count is overwritten)
- (Better solutions are presented later)

1.4.3 Counting things (2)

[ ]:
```
# avoid counting same letter multiple times

counts = {letter: word.count(letter) for letter in set(word)}
counts
```

Still not ideal: needs at least *two* passes * first, to create set * second, to collect all counts

The 'powerful' dictionary data structure *encapsulates* all kinds of loops

[ ]:
```
d.  # Hit TAB key for code completion; the SHIFT-TAB for documentation
```

1.4.4 Operations that don't modify dicts

- `d.get(key, default=None)` (get value for key; use default if not present)
- `d.keys()` - iterable 'view' for keys
- `d.values()` - iterable 'view' for values
- `d.items()` - iterable 'view' for items as key-value pairs `... for key, value in d.items() ...`

[ ]:
```
counts["A"]
```

[ ]:
```
counts.get("A", 0)
```

[ ]:
```
counts.keys()
```

[ ]:
```
sum(counts.values())
```

[ ]:
```
counts.items()
```

[ ]:
```
# set of letters that occur an even number of times in word

{letter for letter, count in counts.items() if count % 2 == 0}
```

Operation similar to `d.items()`: `enumerate(iterable)` * `enumerate` allows iteration over all pairs `(index, value)`

[ ]:
```
for index, letter in enumerate(word):
    # parentheses around index, letter are optional, here
    print(index, letter)
```

[ ]:
```
dict(enumerate(word, 1))
```

1.4.5 Operations that modify dicts

- `d[key] = value` (update or add key-value pair)
- `del d[key]` (delete key-value pair)
- `d.update(another_dict)` (overlay another dict on top of `d`)
- Also see Built-in Types - Dict

1.4.6 Counting things (3)

More efficient way to count occurences of letters in a text. * Traverse the text *once*, accumulating the counts in a dictionary.

[ ]:
```
def count_text(text: str) -> Dict[str, int]:
    """Return dictionary with count for each letter in text.
    """
    counts = {}  # letter frequencies for traversed part of text

    for letter in text:
#         if letter not in counts:
#             counts[letter] = 0
#         counts[letter] += 1  # won't work by itself! (why?)
        counts[letter] = counts.get(letter, 0) + 1

    return counts
```

[ ]:
```
count_text(word)
```

1.5 More collections

- `defaultdict`: special kind of dict, with default values
- `Counter`: special kind of dict, with int values

[ ]:
```
from collections import defaultdict, Counter
from typing import DefaultDict, Counter
```

[ ]:
```
import collections as co  # access the class (usually not needed)

(issubclass(defaultdict, dict),
 issubclass(co.Counter, dict)
)
```

1.5.1 defaultdict

Type `defaultdict` is a special type of `dict` (a subclass) * offers *default values* for *absent* keys

1.5.2 Counting things (4)

Alternative approach, using `defaultdict` (also see *Think Python* Section 19.7):

- `defaultdict(factory)` is like a dictionary, except that `factory()` is called to get a value when a key is absent
- E.g. `defaultdict(int)` will use `int()` (which equals 0) as default value for absent keys

[ ]:
```
def count_text(text: str) -> Dict[str, int]:
    """Return dictionary with count for each letter in text.
    """
    counts = defaultdict(int)  # use 0 when key is not present

    for letter in text:
        counts[letter] += 1  # now, this works!

    return counts
```

[ ]:
```
count_text(word)
```

1.5.3 Counter

Type `Counter` is also a subclass of `dict` * It has key-`int` key-values * Can be used as a *bag*, also known as *multiset* * Order is irrelvant, but multiplicity is relevant * `int`-value is multiplicity

[ ]:
```
letter_bag: Counter[str] = Counter(P=2, S=1, Y=1)
letter_bag
```

1.5.4 Counting things (5)

[ ]:
```python
bag: Counter[str] = Counter("MISSISSIPPI")
bag
```

Some special operations on a `Counter cnt`: * `+`
Add two counters * `cnt.most_common(n: int = None)`
List the `n` most common elements and their counts from the most common to the least.
If `n` is `None`, then list all element counts in decreasing order. * `cnt.elements()`
Iterator over elements, repeating each as many times as its count.

[ ]:
```python
bag + letter_bag
```

[ ]:
```python
bag - letter_bag
```

[ ]:
```python
bag.most_common(2)
```

[ ]:
```python
for c in bag.elements():
    print(c, end=' ')
```

1.6 Comprehensions and Generators

- *generator expressions*
- *generator functions*, using `yield` and `yield from` instead of `return`

Lazy expressions, computed *on-demand*

[ ]:
```python
[i ** 3 for i in range(10)]  # cubes
```

[ ]:
```python
# square of the sum = sum of the cubes

n = 100  # try different n

sum(range(n)) ** 2, sum([i ** 3 for i in range(n)])
```

- First, all cubes are computed and stored
- Next, they are summed

How can we see that?

[ ]:
```python
def trail(obj: Any, pebble: str) -> Any:
    """Print pebble and return obj.
    """
    print(pebble, end="")
    return obj
```

[ ]:
```python
# Comprehension is completely evaluated and stored before use

for cube in [trail(i ** 3, '.') for i in range(10)]:
```

```
    print(cube)
```

[ ]:
```
# Generator expression is only evaluated as needed

for cube in (trail(i ** 3, '.') for i in range(10)):
#     if cube > 100:
#         break
    print(cube)
```

1.6.1 Generator expressions

Syntax:

`(E(v) for v in iterable if C(v))`

Semantics: 1. Take items from an ***iterable***: `python    for v in iterable` 1. ***select*** items based on a condition: `python    if C(v)` 1. ***transform*** the selected items using an expression: `python E(v)` 1. and *yield* items one-by-one, *as needed*

Note the order: first select, then transform
(even though you write the transformation first, and the selection last).

- A generator doesn't construct a list to store all items.
- A generator is **lazy**: it will not be computed completely in advance.
  (In fact, a generator can be endless/infinite.)
- Instead, a generator is only evaluated to the extent that its values are needed.
  The evaluation of a generator is **demand driven**.

A generator is not a list, but it is itself again an *iterable*. In fact, a generator is an *iterator*.
(A list is also an iterable, but a list is completely stored in memory.)

[ ]:
```
# square of the sum = sum of the cubes

n = 100  # try different n

sum(range(n)) ** 2, sum(i ** 3 for i in range(n))  # less memory used
```

Note the omission of parentheses in `sum(i ** 3 for i in range(n))` * This is short for `sum( (i ** 3 for i in range(n)) )`

[ ]:
```
from typing import Optional

def first(iterable: Iterable[Any]) -> Optional[Any]:
    """Return first item from iterable.
    """
    for item in iterable:
        return item  # and ignore everything else
```

[ ]:
```
print( first( [trail(i ** 3, '.') for i in range(10)] ) )
```

[ ]:
```
print( first( trail(i ** 3, '.') for i in range(10) ) )
```

1.6.2 Warning about generator expressions

- Generator expressions can be used only *once*

[ ]:
```
cubes_10 = [n ** 3 for n in range(10)]   # comprehension

for cube in cubes_10:
    print(cube)

print(5 * '-')

for cube in cubes_10:
    print(cube)

print(5 * '-')
```

[ ]:
```
cubes_10 = (n ** 3 for n in range(10))   # generator

for cube in cubes_10:
    print(cube)

print(5 * '-')

for cube in cubes_10:
    print(cube)

print(5 * '-')
```

[ ]:
```
cubes_10 = (n ** 3 for n in range(10))   # generator

for cube in cubes_10:
    print(cube)
    if cube > 10:
        break

print(5 * '-')

for cube in cubes_10:
    print(cube)

print(5 * '-')
```

1.6.3 Generator functions

- Generator function = function that returns a generator

- It is a *generator factory*

```
[ ]: type((i ** 3 for i in range(10)))
```

```
[ ]: from typing import Iterator
```

```
[ ]: def cubes(n: int) -> Iterator[int]:  # previously Generator[int, None, None]:
         """Return generator for first n cubes.
         """
         return (i ** 3 for i in range(n))
```

```
[ ]: for cube in cubes(10):
         print(cube)

     print(5 * '-')

     for cube in cubes(10):
         print(cube)

     print(5 * '-')
```

1.6.4 yield statement in generator function

- Using `yield` instead of `return` makes a function a *generator function*

```
[ ]: # Advanced generator factory

     def gen_cubes(n: int) -> Iterator[int]:  # Generator[int, None, None]:
         """Yield cubes < n.
         """
         i, cube = 0, 0  # cube == i ** 3

         while cube < n:
             yield cube
             i += 1
             cube = i ** 3
```

```
[ ]: type(gen_cubes(100))
```

```
[ ]: for cube in gen_cubes(100):
         print(cube)
```

1.6.5 Nested generator expressions

```
[ ]: # Nested generators: no storage wasted

     all( sum(range(n)) ** 2 == sum(cubes(n))
```

```
    for n in range(1000)
  )
```

Note: The above expression does *recompute* many cubes and sums

[ ]:
```
n, s, c = 0, 0, 0  # s = sum(range(n)); c = sum(cubes(n))

while n < 1000:
    if s ** 2 != c:
        print(False)
        break
    n, s, c = n + 1, s + n, c + n ** 3

else:
    print(True)
```

1.6.6 For more details

- Separate notebook: *Comprehensions and Generators* (in Handouts)

1.7 What Next?

- This conclude the coverage of the Python core
- Next look more at programming as problem solving
  - Does a given list contain duplicates?
  - Which?

[ ]:
```
# Determine duplicate lines in the file with this notebook

with open('2IS50-2223-Lecture-3-A.ipynb') as f:
    # len(f) does not work; f is an iterator
    n = sum(1 for _ in f)  # number of lines in f
    # f is now 'exhausted', and must be opened again

with open('2IS50-2223-Lecture-3-A.ipynb') as f:
    unique = len(set(f))  # number of unique lines in f

n, unique
```

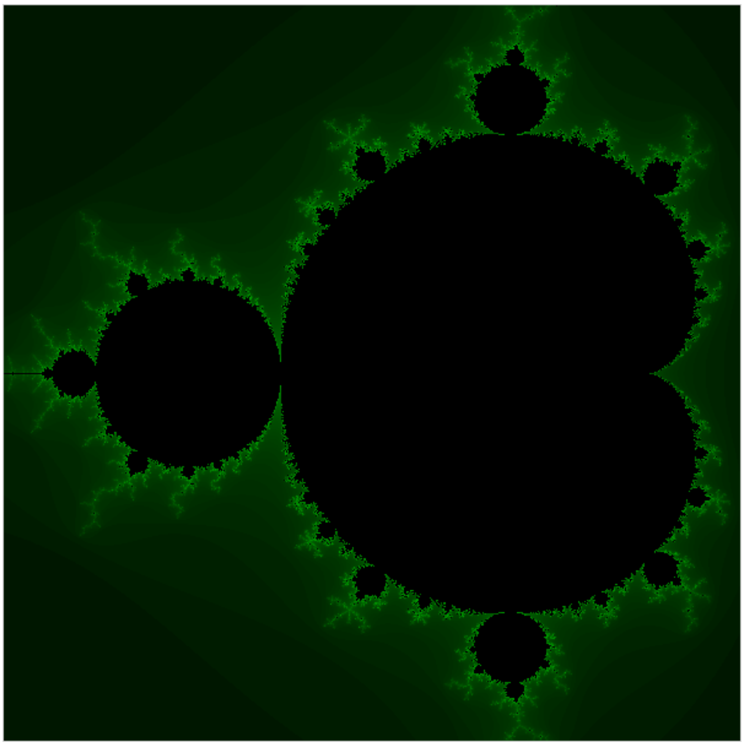
[ ]:
```
with open('2IS50-2223-Lecture-3-A.ipynb') as f:
    for line, count in Counter(f).most_common():
        if count > 1:
            print(f"{count:3} - {line.rstrip()}")
```

1.8 (End of Notebook)

In [ ]:

```
# 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 [ ]:

```
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 [ ]:

```
%%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:**

```
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 [ ]:

```
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

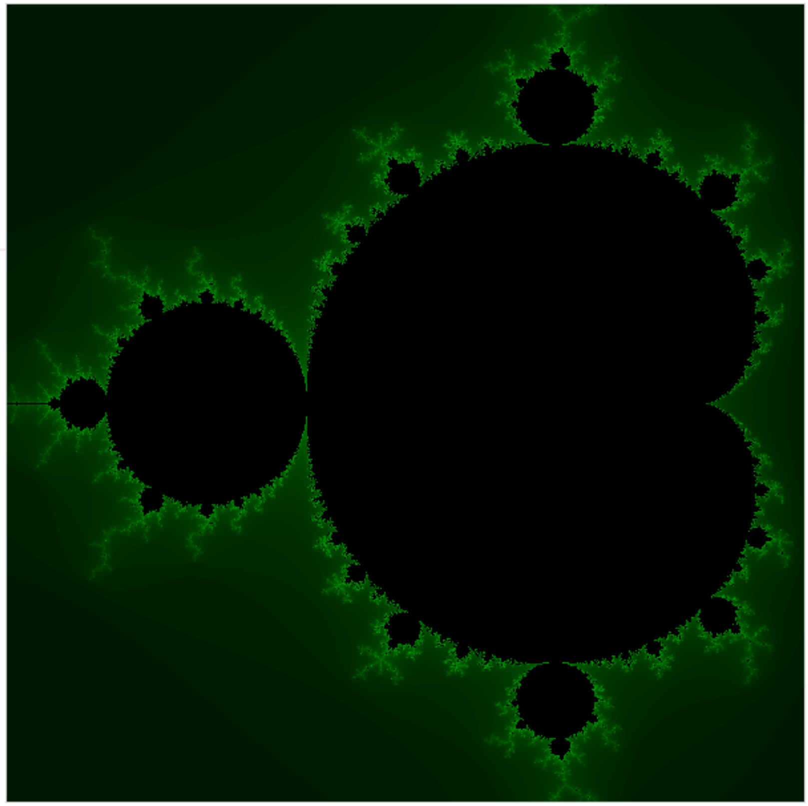
- Each issue has a unique number
- Refer to issue by prefixing its number with `#`
  - in issue descriptions
  - in commit messages
- Turned into a link to that issue

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.A (Python)

Lecturer: Tom Verhoeff

Also see the book *Think Python* (2e), by Allen Downey

Review of Lecture 3.A

- Algorithms and data structures
- Organizing data: sets and dictionaries
  - other `collections`: `defaultdict`, `Counter`
- Avoid unnecessary computation and storage
  - Generator expressions, generator functions

Preview of Lecture 4.A

- Standard algorithms
  - Sorting and searching
  - `key` argument in `sorted`, `min`, `max`
- Object-oriented programming (OOP)

In [1]:
```
# Preliminaries

from collections import defaultdict, Counter
```

```python
from typing import Tuple, List, Dict, DefaultDict, Counter
from typing import Any, Optional, Sequence, Mapping, Iterable, TypeVar
import math
import doctest
```

Standard Algorithms

- There are many recurring computational problems
  - Searching
  - Sorting (to improve searching)
  - ...
- There are many solutions for these problems
- There are many trade-offs
  - Depending on input characteristics
  - Depending on goals: less time, less memory

Standard algorithms are often described in *general terms*

- Not using a specific programming language
- Ignoring (language/machine-specific) details
- Focus on *correctness*

Algorithm Description

What to provide when describing an algorithm:

- **Name**
  - E.g.: ArgMax
- **Inputs** and **assumptions** (constraints)
  - E.g.: a non-empty sequence s of integers
- **Outputs**
  - E.g.: integer i
- **Intended relation between input and output**
  - E.g.: $\max(s)$ occurs in s at index i
  - *Output need not be uniquely determined by input*
- **Performance characteristics** (cost)
  - E.g. Runtime linear in length of sequence
- Its **computational steps** (recipe)
  - E.g. in *pseudo code*

We will use Python

Searching

- Given an input *collection*
- *Find* an item in it having some specified *property*

Problem: ArgMax

In [2]:
```
def arg_max(s: Sequence[int]) -> int:
    """Find index in s where maximum of s occurs.

    Traverse s once.
    Assumption: s is non-empty

    >>> arg_max([13, 42, 17, 42]) in {1, 3}
    True
    """
```

In [3]:
```
doctest.run_docstring_examples(arg_max, globs=globals(), name='arg_max')
```

```
**********************************************************************
File "__main__", line 7, in arg_max
Failed example:
    arg_max([13, 42, 17, 42]) in {1, 3}
Expected:
    True
Got:
    False
```

- The following 'solution' may seem acceptable

In [4]:
```
def arg_max(s: Sequence[int]) -> int:
    """Find index in s where maximum of s occurs.

    Traverse s once.
    Assumption: s is non-empty

    >>> arg_max([13, 42, 17, 42]) in {1, 3}
    True
    """
    return s.index(max(s))
```

In [5]:
```
doctest.run_docstring_examples(arg_max, globs=globals(), name='arg_max')
```

It works.

Why not acceptable?

- Sequence is traversed *twice*:

  1. When determining maximum
  2. When finding its index

In [6]:
```
def arg_max(s: Sequence[int]) -> int:
    """Find index in s where maximum of s occurs.
```

```
    Traverse s once.
    Assumption: s is non-empty

    >>> arg_max([13, 42, 17, 42]) in {1, 3}
    True
    """
    m = - math.inf  # invariant: m == maximum seen so far
    i = None  # invariant: i == index where m was seen

    for index, number in enumerate(s):
        if number > m:
            i, m = index, number

    return i
```

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

Note the use of `enumerate`

Could have done

- `index = 0` before loop
- `index += 1` inside loop (at end)

ArgMax alternative solutions

- Solution above actually finds *smallest* index where maximum occurs
  - This *could* have been required for the algorithm (but *wasn't*)
  - N.B. *Stronger* requirement may preclude efficient solutions
- How to find *largest* index?

```
    if number >= m:
```

or traverse sequence in reverse order

- Solution using *comparison of tuples* and built-in `max`
- Solution using built-in function `max` with `key` parameter
- Solution using *Numpy* (`import numpy as np` and `np.argmax`)

Solution using comparison of tuples and built-in `max`

In [8]:
```
def arg_max(s: Sequence[int]) -> int:
    """Find index in s where maximum of s occurs.

    Traverse s once.
    Assumption: s is non-empty
```

```
    >>> arg_max([13, 42, 17, 42]) in {1, 3}
    True
    """
    m, i = max((number, index) for index, number in enumerate(s))
    # m is maximum, and it occurs at index i
    return i
```

In [9]:
```
doctest.run_docstring_examples(arg_max, globs=globals(), name='arg_max')
```

Notes:

- Tuples are compared according to *lexicographic order*
  - $(a_0, a_1, \ldots) < (b_0, b_1, \ldots)$ if and only if\ there exists an index i with $a[0:i) = b[0:i)$ and $a[i] < b[i]$
  - I.e., find *smallest* index where they differ, and compare there
- We used *tuple unpacking*
  - Since `m` is not used, we usually prefer `_, i = max(...)`
  - Could have avoided this:

    ```
        return max((number, index) for ...)[1]
    ```

- This program finds the *largest* index where the maximum occurs (why?)

In [10]:
```
arg_max([13, 42, 17, 42])
```

Out[10]: 3

- Using `map` and `reversed` (not recommended, because less readable)
  - N.B. `reversed` and `enumerate` are *lazy* (demand driven)

In [11]:
```
def arg_max(s: Sequence[int]) -> int:
    """Find index in s where maximum of s occurs.

    Traverse s once.
    Assumption: s is non-empty

    >>> arg_max([13, 42, 17, 42]) in {1, 3}
    True
    """
    _, i = max(map(lambda t: tuple(reversed(t)), enumerate(s)))
    return i
```

In [12]:
```
doctest.run_docstring_examples(arg_max, globs=globals(), name='arg_max')
```

Solution using built-in `max` with `key` parameter

- `max(iterable, key=f)` returns first `i in iterable` where `f(i)` is maximal

In [13]:
```
def arg_max(s: Sequence[int]) -> int:
```

```python
    """Find index in s where maximum of s occurs.

    Traverse s once.
    Assumption: s is non-empty

    >>> arg_max([13, 42, 17, 42]) in {1, 3}
    True
    """
    return max(enumerate(s), key=lambda t: t[1])[0]
```

In [14]:
```python
doctest.run_docstring_examples(arg_max, globs=globals(), name='arg_max')
```

Analyze `max(enumerate(s), key=lambda t: t[1])[0]`

- items in the iterable (first argument of `max`) have the shape `(i, s[i])`
- Key-function `f(t) == t[1]`; thus, `f((i, s[i])) == s[i]`
- So, `max` returns `(i, s[i])` where `s[i]` is maximal
- Of this returned pair, the *first* item is taken: `max(...)[0]`

It returns the *first* occurrence of the maximum (with least index)

In [15]:
```python
arg_max([13, 42, 17, 42])
```

Out[15]: 1

Simplify this approach:

In [16]:
```python
def arg_max(s: Sequence[int]) -> int:
    """Find index in s where maximum of s occurs.

    Traverse s once.
    Assumption: s is non-empty

    >>> arg_max([13, 42, 17, 42]) in {1, 3}
    True
    """
    return max(range(len(s)), key=lambda i: s[i])
```

In [17]:
```python
doctest.run_docstring_examples(arg_max, globs=globals(), name='arg_max')
```

Analyze `max(range(len(s)), key=lambda i: s[i])`

- items in the iterable are indices `0`, `1`, ...
- Key-function `f(i) == s[i]`
- So, `max` returns smallest `i` where `s[i]` is maximal

Variants of ArgMax Problem

- Find *all* positions where maximum occurs (Exercise)
- Count *number of times* that maximum occurs

- The parameter could be `Iterable[int]`, instead of `Sequence[int]`

CountMax Problem

- **Name**: CountMax
- **Inputs** and **constraints** (assumptions):
  - An iterable s of integers
- **Outputs**: integer c
- **Intended relation between input and output**:
  - $c = $ how often $\max(s)$ occurs in s (0 if s empty)
  - Input *uniquely determines* output
- **Performance characteristics** (cost)
  - Runtime linear in length of sequence, no extra storage

In [18]:
```
def count_max(s: Iterable[int]) -> int:
    """Count how often maximum of s occurs.

    Traverse s once. Don't store all numbers.

    >>> count_max([])
    0
    >>> count_max(iter([13, 42, 17, 42]))
    2
    """
```

In [19]:
```
doctest.run_docstring_examples(count_max, globs=globals(), name='count_max
')
```

```
**********************************************************************
File "__main__", line 6, in count_max
Failed example:
    count_max([])
Expected:
    0
Got nothing
**********************************************************************
File "__main__", line 8, in count_max
Failed example:
    count_max(iter([13, 42, 17, 42]))
Expected:
    2
Got nothing
```

Notes:

- `s` is an iterable: for all you know, its items can only be visited once
  - There is no guarantee that multiple iterations work
  - `count_max(int(item) for item in "13 42 17 42".split())`
  - `count_max(item for item in [13, 42, 17, 42])`
  - `count_max(iter([13, 42, 17, 42]))`

- So, the following 'solution' is not acceptable (for 2 reasons)

In [20]:
```
def count_max(s: Iterable[int]) -> int:
    """Count how often maximum of s occurs.

    Traverse s once. Don't store all numbers.

    >>> count_max([])
    0
    >>> count_max(iter([13, 42, 17, 42]))
    2
    """
    return s.count(max(s, default=0))
```

In [21]:
```
doctest.run_docstring_examples(count_max, globs=globals(), name='count_max
')
```

```
**********************************************************************
File "__main__", line 8, in count_max
Failed example:
    count_max(iter([13, 42, 17, 42]))
Exception raised:
    Traceback (most recent call last):
      File "/Users/wstomv/opt/anaconda3/lib/python3.9/doctest.py", line 13
36, in __run
        exec(compile(example.source, filename, "single",
      File "<doctest count_max[1]>", line 1, in <module>
        count_max(iter([13, 42, 17, 42]))
      File "/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_112
94/1433346288.py", line 11, in count_max
        return s.count(max(s, default=0))
    AttributeError: 'list_iterator' object has no attribute 'count'
```

N.B. `count` is also not defined for a `set`

- `set` is iterable, but not indexable
- `set` not so interesting input for `count_max`: no duplicates

The following is also not acceptable

In [22]:
```
def count_max(s: Iterable[int]) -> int:
    """Count how often maximum of s occurs.

    Traverse s once. Don't store all numbers.

    >>> count_max([])
    0
    >>> count_max(iter([13, 42, 17, 42]))
    2
    """
    items = list(s)
    return items.count(max(items, default=0))
```

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

It works.

Why not acceptable?

- All numbers are stored (temporarily)
- `s` is traversed once, but `items` is traversed *twice*:
  1. When determining maximum
  2. When counting how often it occurs

In [24]:
```
def count_max(s: Iterable[int]) -> int:
    """Count how often maximum of s occurs.

    Traverse s once. Don't store all numbers.

    >>> count_max([])
    0
    >>> count_max(iter([13, 42, 17, 42]))
    2
    """
    m = - math.inf  # invariant: m is maximum seen so far
    c = 0  # invariant: m occurs c times among values seen so far

    for number in s:
        if number == m:
            c += 1
        elif number > m:
            m, c = number, 1

    return c
```

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

Sorting

- Needs items that can be compared for ordering (using *less than* relation)
- Goals:
  - Organized output: same values are grouped together
  - Improve further operations: faster searching
- Many problem variations
  - Duplicates allowed in input or not
  - *Stable* (equal items remain in original order), or not
  - Few different values in input, or many
  - Almost sorted input, or not
  - Speed versus memory usage

Many sorting algorithms

- Slow, but *in place*
  - **Bubble sort**
  - **Selection sort**
  - **Insertion sort** (fast if input almost sorted)
- Generally fast, *in place*
  - **Quick sort**
- Always fast
  - **Merge sort** (extra memory)
  - **Heap sort**
- Special cases
  - **Counting sort**
  - **Radix sort**

[Visualize sorting algorithms]

Sorting advice

- Use built-in functions, unless ...
- Also see https://docs.python.org/3/howto/sorting.html

Sorting on a key

Example:

- Sort table of name-birthday pairs on name,\ or on birthday
- What you sort on is called the *sort key*

Built-in function `sorted` can take `key` parameter

- `key` parameter is function of *one argument* that returns the *sort key*
- `sorted(iterable, key=f)` returns `new` list of items from iterable, such that `map(f, result)` is in ascending order
- guaranteed to be *stable*

In [26]:
```
names = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday
", "Sunday"]

sorted(names, key=len)
```

Out[26]:
```
['Monday', 'Friday', 'Sunday', 'Tuesday', 'Thursday', 'Saturday', 'Wednesd
ay']
```

In [27]:
```
table = [("Amalia", (2023, 5, 23)),
         ("Juliana", (1909, 4, 30)),
         ("Beatrix", (1938, 1, 31)),
         ("Willem-Alexander", (1967, 4, 27)),
         ("Amalia", (2003, 12, 7))
```

```
        ]
```

In [28]:
```
# sort items lexicographically
# * first on name
# * then on birthday

sorted(table)
```

Out[28]:
```
[('Amalia', (2003, 12, 7)),
 ('Amalia', (2023, 5, 23)),
 ('Beatrix', (1938, 1, 31)),
 ('Juliana', (1909, 4, 30)),
 ('Willem-Alexander', (1967, 4, 27))]
```

In [29]:
```
# sort items on name (note difference)

sorted(table, key=lambda t: t[0])
```

Out[29]:
```
[('Amalia', (2023, 5, 23)),
 ('Amalia', (2003, 12, 7)),
 ('Beatrix', (1938, 1, 31)),
 ('Juliana', (1909, 4, 30)),
 ('Willem-Alexander', (1967, 4, 27))]
```

In [30]:
```
# sort items on birthday

sorted(table, key=lambda t: t[1])
```

Out[30]:
```
[('Juliana', (1909, 4, 30)),
 ('Beatrix', (1938, 1, 31)),
 ('Willem-Alexander', (1967, 4, 27)),
 ('Amalia', (2003, 12, 7)),
 ('Amalia', (2023, 5, 23))]
```

In [31]:
```
# sort items on birth month

sorted(table, key=lambda t: t[1][1])
```

Out[31]:
```
[('Beatrix', (1938, 1, 31)),
 ('Juliana', (1909, 4, 30)),
 ('Willem-Alexander', (1967, 4, 27)),
 ('Amalia', (2023, 5, 23)),
 ('Amalia', (2003, 12, 7))]
```

In [32]:
```
# sort items on birth month, then day

sorted(table, key=lambda t: t[1][1:])
```

Out[32]:
```
[('Beatrix', (1938, 1, 31)),
 ('Willem-Alexander', (1967, 4, 27)),
 ('Juliana', (1909, 4, 30)),
 ('Amalia', (2023, 5, 23)),
 ('Amalia', (2003, 12, 7))]
```

In [33]:
```
# sort items on birth month, and then sort on day
# relies on stability of sorting algorithm
```

```python
sorted(sorted(table, key=lambda t: t[1][1]), key=lambda t: t[1][2])
```

Out[33]:
```
[('Amalia', (2003, 12, 7)),
 ('Amalia', (2023, 5, 23)),
 ('Willem-Alexander', (1967, 4, 27)),
 ('Juliana', (1909, 4, 30)),
 ('Beatrix', (1938, 1, 31))]
```

In [34]:
```python
# sort items on day of birth, and then sort on month
# relies on stability of sorting algorithm

sorted(sorted(table, key=lambda t: t[1][2]), key=lambda t: t[1][1])
```

Out[34]:
```
[('Beatrix', (1938, 1, 31)),
 ('Willem-Alexander', (1967, 4, 27)),
 ('Juliana', (1909, 4, 30)),
 ('Amalia', (2023, 5, 23)),
 ('Amalia', (2003, 12, 7))]
```

In [35]:
```python
# Example: sort powers of 7 on last digit

sorted((7 ** i for i in range(10)), key=lambda n: n % 10)
```

Out[35]:
```
[1, 2401, 5764801, 343, 823543, 7, 16807, 40353607, 49, 117649]
```

Stable sorting

A sorting algorithm is called *stable* when

> Items with the same sort key remain in *original order*

- Built-in `sorted` and `list.sort` are stable
- Advantage: easy to sort on multiple keys in multiple calls

In [36]:
```python
items = "yb xa ya xb".split()

items2 = sorted(items, key=lambda item: item[-1], reverse=True)

items2, sorted(items2, key=lambda item: item[0])
```

Out[36]:
```
(['yb', 'xb', 'xa', 'ya'], ['xb', 'xa', 'yb', 'ya'])
```

- First reverse sorts on last column, then sorts on first column
- Result is sorted on first column, and if equal then on last column in reverse!

Could be done in one call, exploiting lexicographic order of tuples (N.B. use of `-ord(...)`):

In [37]:
```python
sorted(items, key=lambda item: (item[0], -ord(item[-1])))
```

Out[37]:
```
['xb', 'xa', 'yb', 'ya']
```

Searching in Sorted Sequence

- Built-in method `list.index` searches *linearly* from left to right
  - Works for any sequence
- *Binary search* searches *logarithmically* by repeated halving
  - Works for *sorted* sequences
  - Can use `bisect.bisect` from Python standard library

In [38]:
```
T = TypeVar('T')  # values in T must comparable

def binary_search(s: Sequence[T], x: T) -> int:
    """Find index i in s such that s[i] <= x < s[i + 1].

    Pretend s[-1] == - math.inf and s[len(s)] == math.inf

    >>> binary_search(list("bdfhjln"), "i")
    3
    >>> binary_search(list("bdfhjln"), "h")
    3
    >>> binary_search(list("bdfhjln"), "a")
    -1
    >>> binary_search(list("bdfhjln"), "o")
    6
    """
    lo, hi = -1, len(s)
    # invariant: -1 <= lo < hi <= len(s) and s[lo] <= x < s[hi]

    while hi - lo != 1:
        m = (lo + hi) // 2  # lo < m < hi, hence 0 <= m < len(s)
        if s[m] <= x:
            lo = m
        else:
            hi = m
        # hi - lo is roughly halved

    # lo + 1 == hi, hence s[lo] <= x < s[lo + 1]
    return lo
```

In [39]:
```
doctest.run_docstring_examples(binary_search, globs=globals(), name='binar
y_search')
```

Notes:

- We used a so-called *type variable* to enforce that
  - type of `x` equals type of items in `s`
- `binary_search` *doesn't* require `s` to be sorted
- If sorted, then output uniquely determined by input
  - otherwise, not necessarily:
  - consider: `binary_search(list("MISSISSIPPI"), "I")`
- If `s` is sorted, then
  - `x in s` holds if and only if `x == s[binary_search(s, x)]`

Object-Oriented Programming

- *Think Python* (2e), Chapter 15-18
- *Real Python*: [Object-Oriented Programming (OOP) in Python 3]()

- 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

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`
  - Attributes whose names start and end with `__` are **magic attributes**
  - `t.__repr__()`: `repr(t)` returns a precise string representation of `t`
  - `t.__str__()`: `str(t)` returns human readable string (default: same as `repr(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 [40]:
```
bag: Counter = Counter('Mississippi')
```

In [41]:
```
print(bag.__doc__)
```

```
Dict subclass for counting hashable items.  Sometimes called a bag
    or multiset.  Elements are stored as dictionary keys and their counts
    are stored as dictionary values.

    >>> c = Counter('abcdeabcdabcaba')  # count elements from a string

    >>> c.most_common(3)                # three most common elements
    [('a', 5), ('b', 4), ('c', 3)]
    >>> sorted(c)                       # list all unique elements
    ['a', 'b', 'c', 'd', 'e']
    >>> ''.join(sorted(c.elements()))   # list elements with repetitions
    'aaaaabbbbcccdde'
    >>> sum(c.values())                 # total of all counts
    15

    >>> c['a']                          # count of letter 'a'
    5
    >>> for elem in 'shazam':           # update counts from an iterable
    ...     c[elem] += 1                # by adding 1 to each element's co
unt
```

```
    >>> c['a']                          # now there are seven 'a'
    7
    >>> del c['b']                      # remove all 'b'
    >>> c['b']                          # now there are zero 'b'
    0

    >>> d = Counter('simsalabim')        # make another counter
    >>> c.update(d)                     # add in the second counter
    >>> c['a']                          # now there are nine 'a'
    9

    >>> c.clear()                       # empty the counter
    >>> c
    Counter()

    Note:  If a count is set to zero or reduced to zero, it will remain
    in the counter until the entry is deleted or the counter is cleared:

    >>> c = Counter('aaabbc')
    >>> c['b'] -= 2                     # reduce the count of 'b' by two
    >>> c.most_common()                 # 'b' is still in, but its count i
s zero
    [('a', 3), ('c', 1), ('b', 0)]
```

In [42]:
```python
bag.most_common
```

Out[42]:
```
<bound method Counter.most_common of Counter({'i': 4, 's': 4, 'p': 2, 'M':
 1})>
```

In [43]:
```python
bag.most_common(1)   # bag is an implicit argument for most_common()
```

Out[43]:
```
[('i', 4)]
```

In [44]:
```python
help(Counter.most_common)   # look at the parameters
```

```
Help on function most_common in module collections:

most_common(self, n=None)
    List the n most common elements and their counts from the most
    common to the least.  If n is None, then list all element counts.

    >>> Counter('abracadabra').most_common(3)
    [('a', 5), ('b', 2), ('r', 2)]
```

Defining your own type: `class`

In [45]:
```python
class Card:
    """A mutable card with an up and down side (non-empty strings).

    >>> Card('', 'O')
    Traceback (most recent call last):
        ...
```

```
    AssertionError: up and down must not be empty
    >>> card = Card('#', 'O')
    >>> card
    Card('#', 'O')
    >>> card.flip()
    >>> print(card)
    O (#)
    """

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

In [46]:

```
doctest.run_docstring_examples(Card, globals(), verbose=True, name="Card")
  # with details
```

```
Finding tests in Card
Trying:
    Card('', 'O')
Expecting:
    Traceback (most recent call last):
        ...
    AssertionError: up and down must not be empty
ok
Trying:
    card = Card('#', 'O')
Expecting nothing
ok
Trying:
    card
Expecting:
    Card('#', 'O')
ok
Trying:
    card.flip()
Expecting nothing
ok
Trying:
    print(card)
Expecting:
    O (#)
```

```
ok
```

In [47]:
```
help(Card)
```

```
Help on class Card in module __main__:

class Card(builtins.object)
 |  Card(up: str, down: str)
 |
 |  A mutable card with an up and down side (non-empty strings).
 |
 |  >>> Card('', 'O')
 |  Traceback (most recent call last):
 |      ...
 |  AssertionError: up and down must not be empty
 |  >>> card = Card('#', 'O')
 |  >>> card
 |  Card('#', 'O')
 |  >>> card.flip()
 |  >>> print(card)
 |  O (#)
 |
 |  Methods defined here:
 |
 |  __init__(self, up: str, down: str)
 |      Create card with given state.
 |
 |  __repr__(self) -> str
 |      Return repr(self).
 |
 |  __str__(self) -> str
 |      Return str(self).
 |
 |  flip(self) -> None
 |      Flip over this card.
 |
 |      Modifies: self
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Magic methods

Magic method: its name starts and ends with *two underscores*

- `__init__`: Initialize an object (automatically called after creation)
- `__repr__`: Return machine-processible string representation of current state
- `__str__`: Return human-readable string representation of current state.

- If a class does not implement `__str__()`, then instead `__repr__()` will be used
- `__repr__` and `__str__` don't need docstring (it is always the same)

Class instantiation

- Create an object: use class name *as function*
  - `card = Card('#', 'O')`
- Also known as *constructor* of class
- Constructor also *initializes* the object, using constructor arguments

Instance variables (attributes)

- Each object has its own *state*
  - State is determined by *instance variables*
  - `card.up` and `card.down`

Instance methods

- *Methods* can inspect and modify the state
  - Objects can be *mutable*
- `card.flip()`
- Methods can access instance variables via `self.name`
- `self` is implicit first argument of methods
  - `card.flip()` is the same as `Card.flip(card)`
  - `self` needs no type hint; `self: Card` is obvious

In [48]:
```
card = Card('#', 'O')
card.up, card.down
```

Out[48]:
```
('#', 'O')
```

In [49]:
```
card.flip()
card
```

Out[49]:
```
Card('O', '#')
```

In [50]:
```
Card.flip(card)
card
```

Out[50]:
```
Card('#', 'O')
```

In [51]:
```
"{:5.2f}".format(math.pi)
```

Out[51]:
```
' 3.14'
```

```
In [52]: str.format("{:5.2f}", math.pi)
```

```
Out[52]: ' 3.14'
```

Another example

A type for quadratic polynomials as objects:

```
In [53]: class QuadPoly:
    """A quadratic polynomial is given by three coefficients a, b, c:
    a x^2 + b x + c, with a != 0.

    >>> q = QuadPoly(1, -8, 12)
    >>> q
    QuadPoly(1, -8, 12)
    >>> print(q)
    1 x^2 + -8 x + 12
    >>> q.eval(0)
    12
    >>> q.eval(2)
    0
    >>> sorted(q.solve())
    [2.0, 6.0]
    >>> QuadPoly(1, 2, 1).solve()
    {-1.0}
    >>> QuadPoly(1, 0, 1).solve()
    set()
    """

    def __init__(self, a: float, b: float, c: float):
        """Create quadratic equation with given coefficients.
        """
        self.a, self.b, self.c = a, b, c

    def __repr__(self) -> str:
        return f"QuadPoly({self.a}, {self.b}, {self.c})"

    def __str__(self) -> str:
        return f"{self.a} x^2 + {self.b} x + {self.c}"

    def eval(self, x) -> float:
        """Evaluate quadratic polynomial in point x.
        """
        return self.a * x ** 2 + self.b * x + self.c

    def solve(self) -> None:
        """Compute approximate solutions of a * x ** 2 + b * x + c == 0.
        """
        p, q = -self.b / (2 * self.a), self.c / self.a
        # a * x ** 2 + b * x + c == 0  <==>  x ** 2 - 2 * p * x + q == 0

        discriminant = p ** 2 - q

        if discriminant >= 0:
            s = math.sqrt(discriminant)
```

```
            return {p + s, p - s}
        else:
            return set()
```

In [54]:

```
doctest.run_docstring_examples(QuadPoly, globals(), verbose=True, name="QuadPoly")  # with details
```

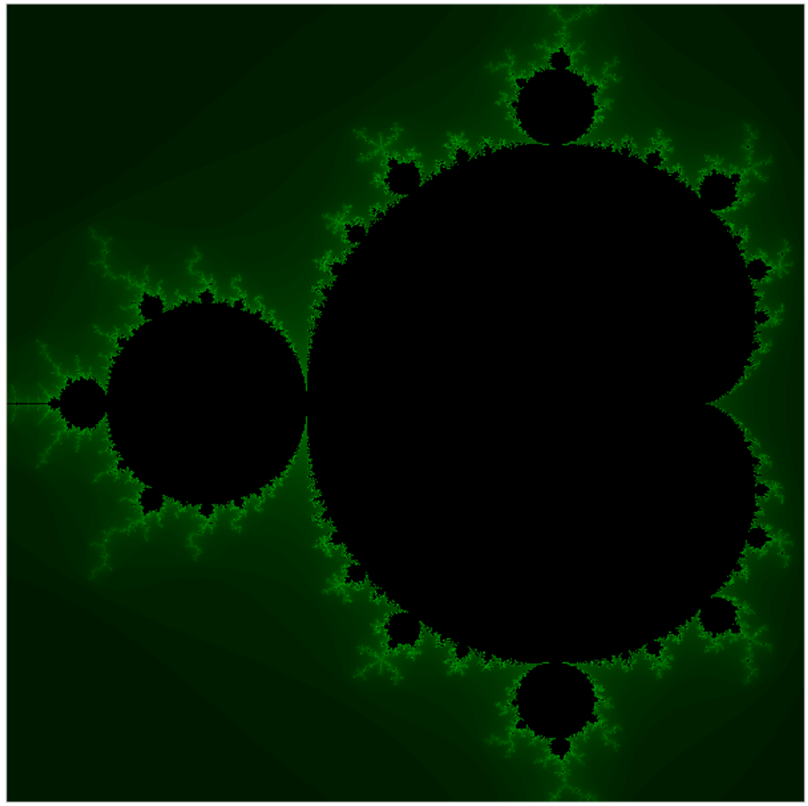
```
Finding tests in QuadPoly
Trying:
    q = QuadPoly(1, -8, 12)
Expecting nothing
ok
Trying:
    q
Expecting:
    QuadPoly(1, -8, 12)
ok
Trying:
    print(q)
Expecting:
    1 x^2 + -8 x + 12
ok
Trying:
    q.eval(0)
Expecting:
    12
ok
Trying:
    q.eval(2)
Expecting:
    0
ok
Trying:
    sorted(q.solve())
Expecting:
    [2.0, 6.0]
ok
Trying:
    QuadPoly(1, 2, 1).solve()
Expecting:
    {-1.0}
ok
Trying:
    QuadPoly(1, 0, 1).solve()
Expecting:
    set()
ok
```

(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]:
```
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='jud
ge_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 == '#'
```

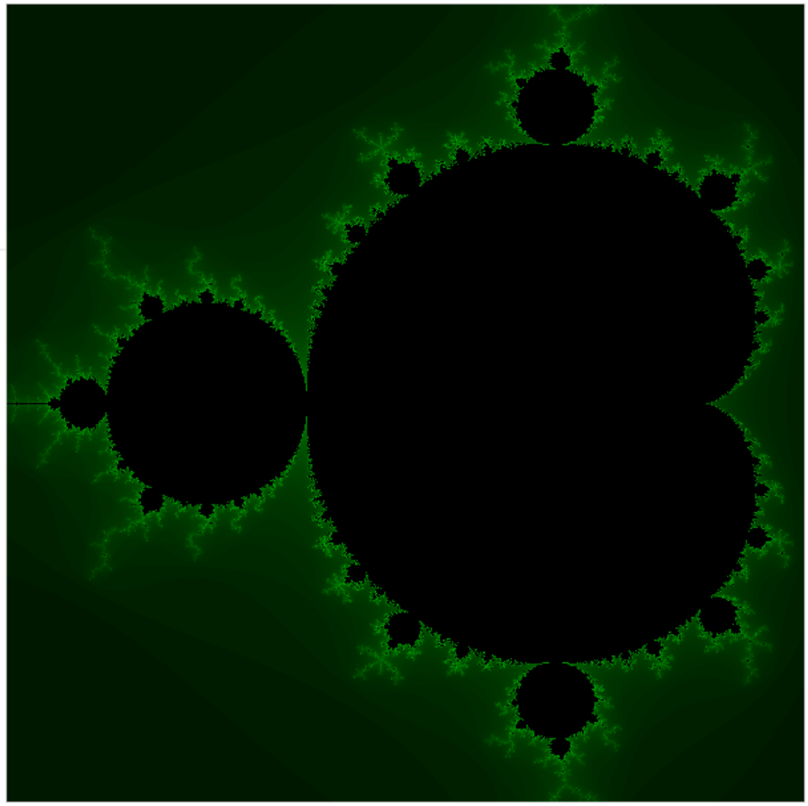
```
.
Set up Card('#', 'O').Tear down Card('#', 'O')
Set up Card('#', 'O').Tear down Card('#', 'O')
Set up Card('#', 'O').Tear down Card('#', 'O')
Set up Card('#', 'O').Tear down Card('O', '#')
```

More information on testing in Python

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

(End of Notebook)

2IS50 – Software Development for Engineers – 2022-2023



Lecture 5.A (Python)

Lecturer: Tom Verhoeff

Also see the book *Think Python* (2e), by Allen Downey

Review of Lecture 4.A

- Standard algorithms
  - Sorting and searching
  - `key` argument in `sorted`, `min`, `max`
- Object-oriented programming (OOP)

Preview of Lecture 5.A

- Robustness
  - Exceptions, `try: ... except: ... finally: ...`, `raise`
- Iterators and iterables
- Object-oriented programming (OOP)
  - Composition of classes
  - Define your own collection type

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]:
```
# Preliminaries

import collections as co
from typing import Tuple, List, Set, Dict, DefaultDict, Counter
from typing import Any, Sequence, Mapping, MutableMapping, Iterable, Iterator
from typing import NewType, TypeVar
import math
import random
from pprint import pprint
import itertools as it
import doctest
```

Program Robustness

A program is called **robust** when

- it works reliably under *unexpected* circumstances

Exceptions

- Exceptional situations are reported by **raising an exception**
- Built-in exceptions, used by built-in operations:
    - index out of bounds
    - key not found
    - division by zero
    - file does not exist

- A Python exception is an *object* holding information such as:
    - location in program: incl. *traceback*
    - nature of the event (exception's type)
    - a message

In [3]:
```
1 / 0
```

```
---------------------------------------------------------------------------
-
ZeroDivisionError                         Traceback (most recent call last
)
/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_40311/145566970
4.py in <module>
```

```
----> 1 1 / 0

ZeroDivisionError: division by zero
```

`try`-`except`

Without program intervention, a raised exception *aborts execution*

- Program can **catch** exception using a `try`-`except` statement

See *Think Python*, Section 14.5

Syntax:

```
try:
    statement_suite_1
except:
    statement_suite_2
```

or variants thereof (see examples)

Semantics:

1. Execute `statement_suite_1`
2. If exception occurs, then
   A. abort that execution
   B. execute `statement_suite_2`

In [4]:
```python
try:
    print(1 / 0)
    print("Further work")
except:
    print("Something went wrong")
```

```
Something went wrong
```

In [5]:
```python
try:
    print(1 / 0)
except ZeroDivisionError:
    print("+inf")
except:
    print("Something else went wrong")
```

```
+inf
```

In [6]:
```python
try:
    print([][0])
except ZeroDivisionError:
    print("+inf")
except Exception as exc:
    print(f"Something else went wrong: {exc}")
```

```
Something else went wrong: list index out of range
```

`assert` and `raise`

Program can also **raise exception** using

- `assert` statement or
- `raise` statement

See *Think Python*, Sections 11.4 and 16.5

In [7]:
```
try:
  assert False, "Should not happen, but it does"
except Exception as exc:
  print(f"Something went wrong: {type(exc).__name__} ({exc})")
```

```
Something went wrong: AssertionError (Should not happen, but it does)
```

In [8]:
```
raise ValueError("Square root argument is < 0")
print("Further work")
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call last
)
/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_40311/368781210
1.py in <module>
----> 1 raise ValueError("Square root argument is < 0")
      2 print("Further work")

ValueError: Square root argument is < 0
```

- `ValueError` is a *class*
- `ValueError` is a *subclass* of `Exception`
- `ValueError("Root argument is < 0")` is a *constructor* call

In [9]:
```
# Find purpose of ValueError

# help(ValueError)  # gives lots of additional information
ValueError.__doc__ # can also use Shift-Tab
```

Out[9]:
```
'Inappropriate argument value (of correct type).'
```

`finally`

There can also be a `finally` clause in `try`-statement:

- this is *always* executed
- Purpose: to do clean-up (close file, etc.)

Syntax:

```
try:
    statement_suite_1
except:
    statement_suite_2
finally:
    statement_suite_3
```

or variants thereof (see examples)

Semantics:

1. Execute `statement_suite_1`
2. If exception occurs, then
   A. abort that execution
   B. execute `statement_suite_2`
3. Always execute `statement_suite_3`

In [10]:
```
try:
    print("Try this")
except Exception:
    print("Exception")
finally:
    print("Finally")
```

```
Try this
Finally
```

In [11]:
```
try:
    print("Try this")
    1 / 0
    print("Should not get here")
except Exception:
    print("Exception")
finally:
    print("Finally")
```

```
Try this
Exception
Finally
```

Notes about exceptions

- Exceptions and their handling add *overhead*
  - But in Python not so much as other languages
- There are hairy details:
  - What if exception occurs when handling an exception?
  - What if statement suite contains `return`?

Iterators and iterables

Iterable = (virtual) collection that can be iterated over

- using `for` construct
- in *loop* or *comprehension* or *generator expression*

Examples of iterables:

- `tuple`, `list`, `set`, `dict`, generator
- result of `range`, `map`, `filter`, `zip`, `enumerate`

Some iterables allow only one iteration

- generator expression, result of `map`, etc.

In [12]:
```python
squares = map(lambda n: n ** 2, range(10))
# squares = (n ** 2 for n in range(10))

list(squares), list(squares)
```

Out[12]:
```
([0, 1, 4, 9, 16, 25, 36, 49, 64, 81], [])
```

Each **iteration** is controlled by its own **iterator**

- iterator holds *administration* of that specific iteration

In other languages, e.g. Java, administration is explicit:

```java
for (i = 0, i < 10, i += 1) {
    // do something with control variable i
    name = names[i]
}
```

Python iterator object 'knows':

- where to *start*
- how to determine *when done*
- how to step to *next item*

In [13]:
```python
beatles = "John Paul George Ringo".split()

for beatle in beatles:
    print(f"Do something with {beatle}")

# beatle is _not_ a control variable
```

```
Do something with John
Do something with Paul
Do something with George
Do something with Ringo
```

Intermezzo on bad iteration style

In [14]:
```
cars = [Counter(car) for car in "McFarri Tipsla Nissota".split()]
cars
```

Out[14]:
```
[Counter({'M': 1, 'c': 1, 'F': 1, 'a': 1, 'r': 2, 'i': 1}),
 Counter({'T': 1, 'i': 1, 'p': 1, 's': 1, 'l': 1, 'a': 1}),
 Counter({'N': 1, 'i': 1, 's': 2, 'o': 1, 't': 1, 'a': 1})]
```

In [15]:
```
# BAD

i = 0

while i < len(cars):
    print(cars[i].most_common(1))
    i += 1 # easy to forget
```

```
[('r', 2)]
[('T', 1)]
[('s', 2)]
```

In [16]:
```
# better, but still BAD

for i in range(len(cars)):
    print(cars[i].most_common(1))
```

```
[('r', 2)]
[('T', 1)]
[('s', 2)]
```

In [17]:
```
# Pythonic

for car in cars: # can also take a slice: cars[start:stop:step]
    print(car.most_common(1))
```

```
[('r', 2)]
[('T', 1)]
[('s', 2)]
```

In [18]:
```
# if you need index as well

for index, car in enumerate(cars):
    print(index, car.most_common(1))
```

```
0 [('r', 2)]
1 [('T', 1)]
2 [('s', 2)]
```

Multiple iterators on same collection

Multiple iterators can be active *concurrently* on same collection:

In [19]:
```
s = ('a', 'b', 'c') # shorter: tuple('abc')
```

```
# t = []

for c in s:
   print(c)
#      t.append(c)
   for d in s: # for d in t:
      print(f'{c.upper()}{d}', end=" ")
   print()
```

```
a
Aa Ab Ac
b
Ba Bb Bc
c
Ca Cb Cc
```

How many iterables and iterators are involved during execution?

This nested `for`-loop involves

- *one* iterable (`s`) and
- *four* iterators:
  - *one* iterator controls the outer loop and
  - *three* the inner loop

Each item in list `s` visited by outer loop

- causes a fresh execution of the inner loop
- with its own iterator

Iterator can be used for *single iteration* only.

Built-in function `iter()`

Obtain iterator from iterable via built-in function `iter()`

In [20]:
```
itr = iter('abc')
print(next(itr)) # get next item for this iteration

for c in itr:
   print(c, end='')

for c in itr:
   print(c, end='*') # not executed

print('.')
next(itr)
```

```
a
bc.
```

```
-
StopIteration                             Traceback (most recent call last
)
/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_40311/214224542
2.py in <module>
      9
     10 print('.')
---> 11 next(itr)

StopIteration: 
```

Type	Supported methods	Purpose
`Iterable`	`__iter__`	Implement `iter(self)`
`Iterator`	`__next__`	Return next item or `raise StopIteration`

It can be confusing that *iterator* can be used as *iterable*

- Iterator object also supports `__iter__` and returns itself

In [21]:
```
itr = iter('abc')
iter(itr) is itr
```
Out[21]: `True`

In [22]:
```
squares = map(lambda n: n ** 2, range(10))

iter(squares) is squares
```
Out[22]: `True`

So, *iterables* that can be iterated over once

- are actually *iterators*

Put differently, if you want to know whether `s` can be iterated over more than once, then check that it *doesn't* support `__next__`:

In [23]:
```
hasattr(s, '__next__'), hasattr(itr, '__next__'), hasattr(squares, '__next__')
```
Out[23]: `(False, True, True)`

Module `itertools`

[`itertools` - Functions creating iterators for efficient looping]

- `itertools.permutations`: iterator for all permutations
- `itertools.combinations`: iterator for all combinations of given size
- `itertools.zip_longest`: like `zip`, but over longest

More information

- [The Iterator Protocol: How "For Loops" Work in Python]
- [How to make an iterator in Python]
- [Python Like You Mean It: Iterables]

Object-Oriented Programming

- Class serves as *type*
- Values of type are *objects*, instantiated from the class

In [24]:
```
class Card:
  """A mutable card with an up and down side (non-empty strings).

    >>> Card('', 'O')
    Traceback (most recent call last):
        ...
    AssertionError: up and down must not be empty
    >>> card = Card('#', 'O')
    >>> card
    Card('#', 'O')
    >>> card.flip()
    >>> print(card)
    O (#)
    """

  def __init__(self, up: str, down: str) -> None:
    """Create card with given state.
        """
    assert up and down, "up and down must not be empty"
    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
```

In [25]:
```
doctest.run_docstring_examples(Card, globals(), name="Card")  # without details
```

Class instantiation, object creation

- Create an object: use class name *as function*

In [26]:
```
card = Card(' Q♡', '#')
type(card), card
```

Out[26]:
```
(__main__.Card, Card(' Q♡', '#'))
```

- A.k.a. *constructor* of class
- Constructor creates object and *initializes* it (using `__init__`), using constructor arguments

- Object *destruction* is automatic in Python
  - When an object becomes *unreachable*, its memory *can* be recycled
  - Known as *garbage collection*

Instance variables (attributes)

- Each object has its own *state*
  - State is determined by *instance variables*
  - `card.up` and `card.down`

Instance methods

- *Methods* can inspect and modify the state
  - Objects can be *mutable*
- `card.flip()`
- Methods can access instance variables via `self.name`
- `self` is implicit first argument of methods
  - `card.flip()` is the same as `Card.flip(card)`
  - `self` needs no type hint; `self: Card` is implied

Class Composition

- Function composition
  - define new function in terms of existing function(s)
- Class composition
  - Define class in terms of existing class(es)

In [27]:
```
#: The card suits
SUITS = '  ♡   '

#: The card ranks
RANKS = 'A 2 3 4 5 6 7 8 9 10 J Q K'.split()

list(SUITS), RANKS
```

Out[27]:
```
(['  ', '♡', '  ', '  '],
 ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K'])
```

In [28]:
```python
class Deck:
    """A deck of (regular playing) cards.
    """

    def __init__(self, cards: Iterable[Card] = None) -> None:
        """Constructs a deck for cards if given,
           otherwise of regular playing cards in sorted order, top to bottom.
           """
        if cards:
            self.cards = list(cards)
        else:
            self.cards = [Card(f"{rank}{suit}", "#")
                          for suit in SUITS
                          for rank in RANKS]

    def __repr__(self) -> str:
        return f"Deck({self.cards!r})"

    def __str__(self) -> str:
        return ''.join(str(card) for card in self.cards)

    def __len__(self) -> int:
        """Return len(self).
           """
        return len(self.cards)

    def __iter__(self) -> Iterator[Card]:
        """Implement iter(self).
           """
        return iter(self.cards)

    def shuffle(self) -> None:
        """Shuffle the deck.

           Modifies: self
           """
        random.shuffle(self.cards)

    # NOTE the strings in `Tuple['Deck', 'Deck']
    # Deck is not yet defined
    def cut(self, n: int) -> Tuple['Deck', 'Deck']:
        """Cut the deck into two decks, the first having n cards.

           Assumption: 0 <= n <= len(self)
           """
        return Deck(self.cards[:n]), Deck(self.cards[n:])

    def rotate(self, n: int) -> None:
        """Cut the deck, taking n cards from the top,
           and putting them underneath.

           Assumption: 0 <= n <= len(self)

           Modifies: self
           """
```

```
        top, bottom = self.cut(n)
        self.cards = bottom.cards + top.cards

        # in-place rotation
#         self.cards.extend(self.cards[:n])
#         del self.cards[:n]

    def show(self, up: bool = True) -> str:
        """Show up or down sides of all cards.
        """
        return ''.join(str(card.up if up else card.down) for card in self)

    def turnover(self) -> None:
        """Turn over the deck, by flipping all cards and reversing their order.

        Modifies: self
        """
        for card in self:
            card.flip()
        self.cards.reverse()

    def riffle(self, in_shuffle: bool = True) -> None:
        """Riffle shuffle the deck,
        taking two halves and merging cards in alternating order.
        If in_shuffle, then top card ends up in second place,
        else on top.

        See: https://www.whydomath.org/Reading_Room_Material/ian_stewart/s
huffle/shuffle.html

        Modifies: self
        """
        half = (len(self) + int(not in_shuffle)) // 2
        top, bottom = self.cut(half)
        if in_shuffle:
            top, bottom = bottom, top
        # merge top and bottom, starting with top
        self.cards = [card
                      for pair in it.zip_longest(top, bottom)
                      for card in pair
                      if card]
```

In [29]:

```
deck = Deck()
print(deck)
deck
```

```
A    (#) 2    (#) 3    (#) 4    (#) 5    (#) 6    (#) 7    (#) 8    (#) 9    (#) 10    (#) J    (#) Q    (#) K    (#) A♡ (#) 2♡ (#)
 3♡ (#) 4♡ (#) 5♡ (#) 6♡ (#) 7♡ (#) 8♡ (#) 9♡ (#) 10♡ (#) J♡ (#) Q♡ (#) K♡ (#) A    (#) 2    (#) 3    (#) 4    (#) 5
   (#) 6    (#) 7    (#) 8    (#) 9    (#) 10    (#) J    (#) Q    (#) K    (#) A    (#) 2    (#) 3    (#) 4    (#) 5    (#) 6    (#)
7    (#) 8    (#) 9    (#) 10    (#) J    (#) Q    (#) K    (#)
```

Out[29]:

```
Deck([Card('A   ', '#'), Card('2   ', '#'), Card('3   ', '#'), Card('4   ', '#'), Card('5   ', '#'), Card('6   ', '#'), Card('7   ', '#'
), Card('8   ', '#'), Card('9   ', '#'), Card('10   ', '#'), Card('J   ', '#'), Card('Q   ', '#'), Card('K   ', '#'), Card('A♡', '#'),
Card('2♡', '#'), Card('3♡', '#'), Card('4♡', '#'), Card('5♡', '#'), Card('6♡', '#'), Card('7♡', '#'), Card('8♡', '#'), Card('
9♡', '#'), Card('10♡', '#'), Card('J♡', '#'), Card('Q♡', '#'), Card('K♡', '#'), Card('A   ', '#'), Card('2   ', '#'), Card('3   '
, '#'), Card('4   ', '#'), Card('5   ', '#'), Card('6   ', '#'), Card('7   ', '#'), Card('8   ', '#'), Card('9   ', '#'), Card('10   ', '#')
, Card('J   ', '#'), Card('Q   ', '#'), Card('K   ', '#'), Card('A   ', '#'), Card('2   ', '#'), Card('3   ', '#'), Card('4   ', '#'), Ca
```

rd('5 ', '#'), Card('6 ', '#'), Card('7 ', '#'), Card('8 ', '#'), Card('9 ', '#'), Card('10 ', '#'), Card('J ', '#'), Card('Q ', '#'), Card('K ', '#')])

In [30]:
```
len(deck)
```
Out[30]:
```
52
```

In [31]:
```
deck.shuffle()
print(deck)
```
```
5   (#) A♡ (#) 3   (#) J   (#) 3   (#) 7   (#) 7   (#) 4   (#) 10♡ (#) 8♡ (#) K   (#) 7♡ (#) 8   (#) 3♡ (#) 10   (#)
 Q   (#) 3   (#) K♡ (#) 5   (#) 6   (#) J   (#) 8   (#) 4   (#) 5♡ (#) 6♡ (#) 2♡ (#) 9♡ (#) A   (#) 6   (#) 10   (#)
Q♡ (#) 4   (#) 2   (#) 9   (#) A   (#) Q   (#) 7   (#) 10   (#) 9   (#) 9   (#) 2   (#) Q   (#) K   (#) J   (#) 5   (#
) A   (#) J♡ (#) 6   (#) 2   (#) 4♡ (#) K   (#) 8   (#)
```

In [32]:
```
print("{}\n\n{}".format(*deck.cut(5)))
```
```
5   (#) A♡ (#) 3   (#) J   (#) 3   (#)

7   (#) 7   (#) 4   (#) 10♡ (#) 8♡ (#) K   (#) 7♡ (#) 8   (#) 3♡ (#) 10   (#) Q   (#) 3   (#) K♡ (#) 5   (#) 6   (#
) J   (#) 8   (#) 4   (#) 5♡ (#) 6♡ (#) 2♡ (#) 9♡ (#) A   (#) 6   (#) 10   (#) Q♡ (#) 4   (#) 2   (#) 9   (#) A   (#
) Q   (#) 7   (#) 10   (#) 9   (#) 9   (#) 2   (#) Q   (#) K   (#) J   (#) 5   (#) A   (#) J♡ (#) 6   (#) 2   (#) 4♡ (
#) K   (#) 8   (#)
```

In [33]:
```
deck.rotate(5)
print(deck)
```
```
7   (#) 7   (#) 4   (#) 10♡ (#) 8♡ (#) K   (#) 7♡ (#) 8   (#) 3♡ (#) 10   (#) Q   (#) 3   (#) K♡ (#) 5   (#) 6   (#
) J   (#) 8   (#) 4   (#) 5♡ (#) 6♡ (#) 2♡ (#) 9♡ (#) A   (#) 6   (#) 10   (#) Q♡ (#) 4   (#) 2   (#) 9   (#) A   (#
) Q   (#) 7   (#) 10   (#) 9   (#) 9   (#) 2   (#) Q   (#) K   (#) J   (#) 5   (#) A   (#) J♡ (#) 6   (#) 2   (#) 4♡ (
#) K   (#) 8   (#) 5   (#) A♡ (#) 3   (#) J   (#) 3   (#)
```

In [34]:
```
# Deck is iterable

for card in deck:
    print(card.up, end=' ')
```
```
7   7   4   10♡ 8♡ K   7♡ 8   3♡ 10   Q   3   K♡ 5   6   J   8   4   5♡ 6♡ 2♡ 9♡ A   6   10   Q♡ 4   2   9
  A   Q   7   10   9   9   2   Q   K   J   5   A   J♡ 6   2   4♡ K   8   5   A♡ 3   J   3
```

In [35]:
```
deck.turnover()

print(deck)
```
```
# (3   ) # (J   ) # (3   ) # (A♡) # (5   ) # (8   ) # (K   ) # (4♡) # (2   ) # (6   ) # (J♡) # (A   ) # (5   ) # (J   ) # (K   ) #
 (Q   ) # (2   ) # (9   ) # (9   ) # (10   ) # (7   ) # (Q   ) # (A   ) # (9   ) # (2   ) # (4   ) # (Q♡) # (10   ) # (6   ) # (A
  ) # (9♡) # (2♡) # (6♡) # (5♡) # (4   ) # (8   ) # (J   ) # (6   ) # (5   ) # (K♡) # (3   ) # (Q   ) # (10   ) # (3♡) # (8
  ) # (7♡) # (K   ) # (8♡) # (10♡) # (4   ) # (7   ) # (7   )
```

In [36]:
```
deck.turnover()

print(deck)
```
```
7   (#) 7   (#) 4   (#) 10♡ (#) 8♡ (#) K   (#) 7♡ (#) 8   (#) 3♡ (#) 10   (#) Q   (#) 3   (#) K♡ (#) 5   (#) 6   (#
) J   (#) 8   (#) 4   (#) 5♡ (#) 6♡ (#) 2♡ (#) 9♡ (#) A   (#) 6   (#) 10   (#) Q♡ (#) 4   (#) 2   (#) 9   (#) A   (#
) Q   (#) 7   (#) 10   (#) 9   (#) 9   (#) 2   (#) Q   (#) K   (#) J   (#) 5   (#) A   (#) J♡ (#) 6   (#) 2   (#) 4♡ (
```

#) K (#) 8 (#) 5 (#) A♡ (#) 3 (#) J (#) 3 (#)

In [37]:
```
deck = Deck()

print(deck.show())
```

A 2 3 4 5 6 7 8 9 10 J Q K A♡ 2♡ 3♡ 4♡ 5♡ 6♡ 7♡ 8♡ 9♡ 10♡ J♡ Q♡ K♡ A 2 3 4 5 6 7 8 9 10 J Q K A 2 3 4 5 6 7 8 9 10 J Q K

In [38]:
```
hand, _ = deck.cut(10)

print(hand.show())
```

A 2 3 4 5 6 7 8 9 10

In [39]:
```
hand.riffle()

print(hand.show())
```

6 A 7 2 8 3 9 4 10 5

In [40]:
```
for _ in range(9):
    hand.riffle()

    print(hand.show())

# Get back original order
```

```
3 6 9 A 4 7 10 2 5 8
7 3 10 6 2 9 5 A 8 4
9 7 5 3 A 10 8 6 4 2
10 9 8 7 6 5 4 3 2 A
5 10 4 9 3 8 2 7 A 6
8 5 2 10 7 4 A 9 6 3
4 8 A 5 9 2 6 10 3 7
2 4 6 8 10 A 3 5 7 9
A 2 3 4 5 6 7 8 9 10
```

(End of Notebook)

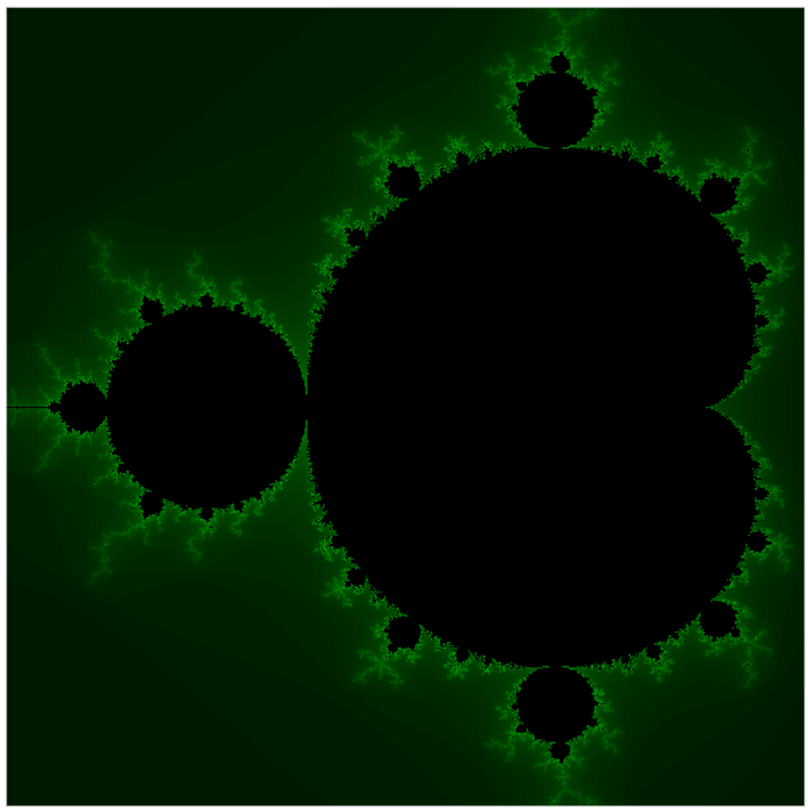
In [1]:
```python
# enable mypy type checking
try:
    %load_ext nb_mypy
except ModuleNotFoundError:
    print("Type checking facility (Nb Mypy) is not installed.")
    print("To use this facility, install Nb Mypy by executing (in a cell):
")
    print("  !python3 -m pip install nb_mypy")
```

```
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

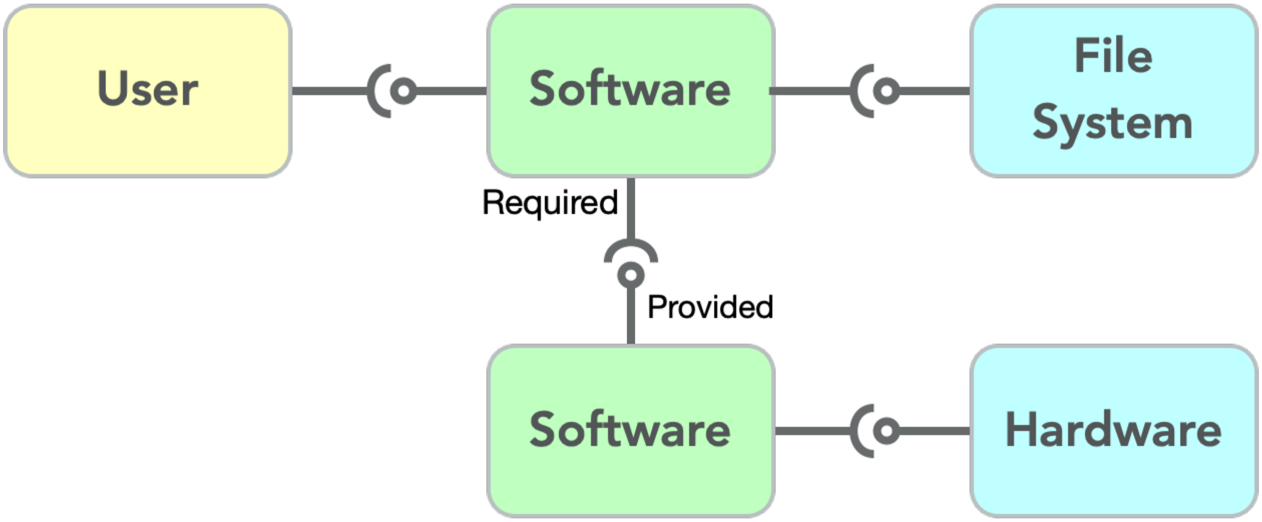
Advice on Documentation

- 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`

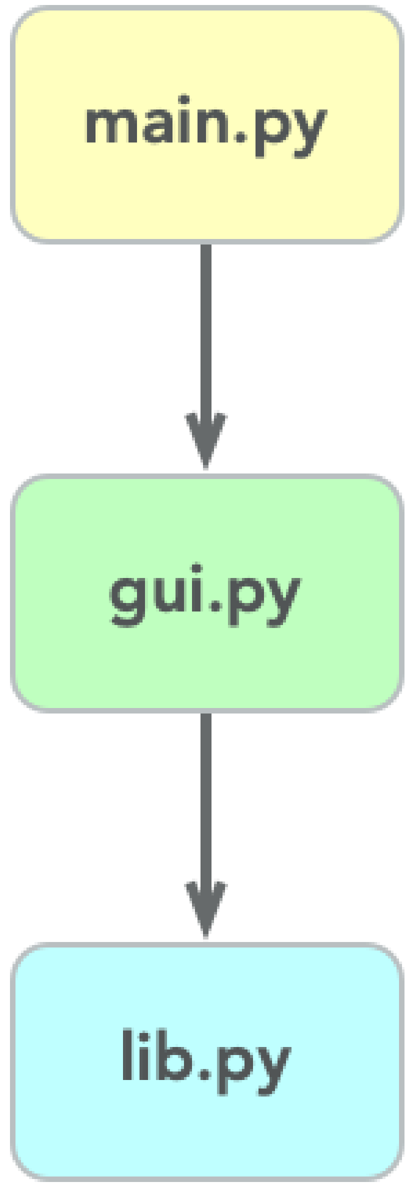
In case you really want/have to go there:

- [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:

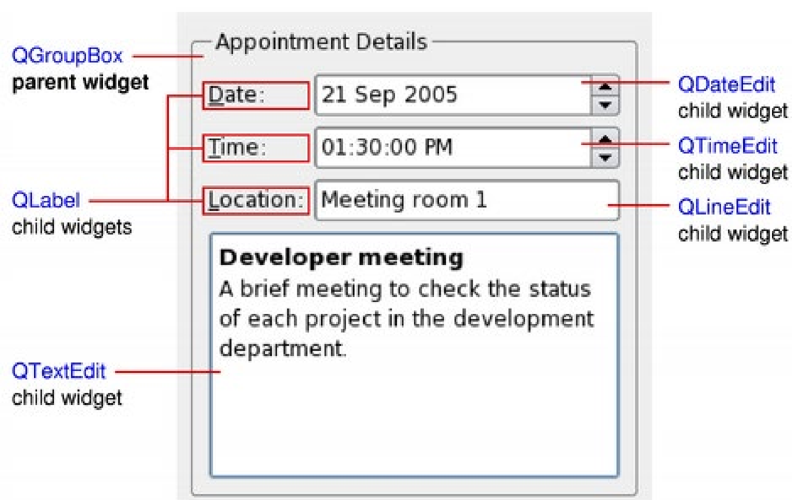
- *DelftStack* Tutorial
- Official Qt5 docs Very good & complete
- Official PyQt5 docs Unfortunately not so complete, better stick to the QT5 documentation!
- PyQt5 Tutorial, Create GUI Applications with Python & Qt — Martin Fitzpatrick
- PyQt5 YouTube Tutorial

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*

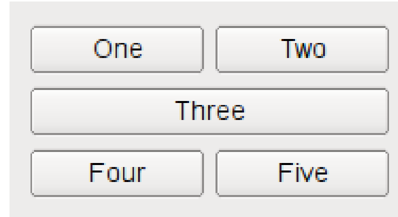
QWidgets

- **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)



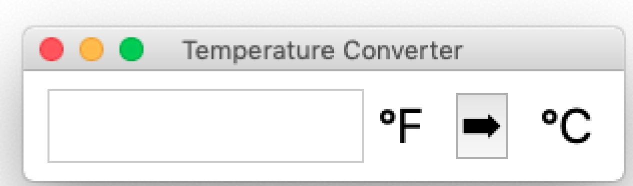
In [11]:
```
import sys
from PyQt5 import QtGui, QtWidgets

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

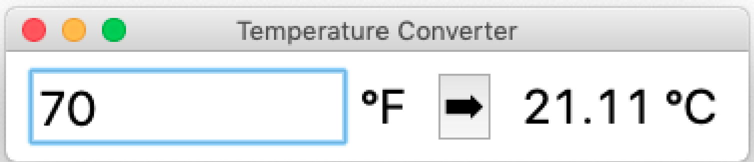
In [12]:
```
class Window(QtWidgets.QWidget):
    def __init__(self) -> None:
        super().__init__()
        self.grid_layout = QtWidgets.QGridLayout()
        self.setLayout(self.grid_layout)

        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)
```

```python
        # 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*

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

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

```python
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.

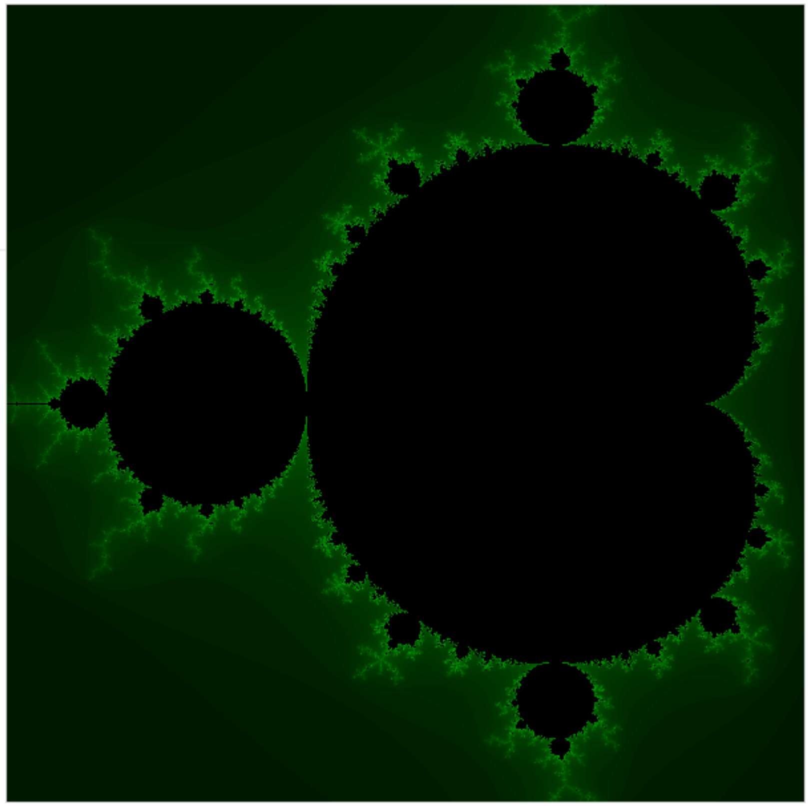
- Runs the game:
  - gets bets from `Player`,
  - spins `Wheel`,
  - collects losing bets,
  - pays winning bets.
- This encapsulates the basic sequence of play into a single class.

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.A (Python)

Lecturer: Tom Verhoeff

Also see the book *Think Python* (2e), by Allen Downey

Review of Lecture 5.A

- Robustness
  - Exceptions, `try: ... except: ... finally: ...`, `raise`
- Iterators and iterables
- Object-oriented programming (OOP)
  - Composition of classes
  - Define your own collection

Preview of Lecture 6.A

- Object-Oriented Programming
  - Inheritance, subclass, superclass
  - Polymorphism
- Argument gathering
- Recursion

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]:
```
# Preliminaries

import collections as co
from typing import Tuple, List, Set, Dict, DefaultDict, Counter
from typing import Any, Optional
from typing import Sequence, Mapping, MutableMapping, Iterable, Iterator,
Callable
from typing import NewType, TypeVar
import math
import random
from pprint import pprint
import itertools as it
import doctest
```

More Rock-Paper-Scissors

- Lecture 4-B showed a monolithic RPS program:
  - Random opponent with given distribution
  - Try all my options against all permutations of opponent
  - Collect outcomes and determine best and guessed option\ (Guess: beat the opponent's most frequent option)

In [3]:
```
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: 552 - 448
My choice: 1
  win - lose: 158 - 842
My choice: 2
  win - lose: 799 - 201
My best choice: 2
Guessed choice: 0

Opponent's probability distribution: 0.40, 0.10, 0.50
My choice: 0
  win - lose: 832 - 168
My choice: 1
  win - lose: 447 - 553
My choice: 2
  win - lose: 186 - 814
My best choice: 0
Guessed choice: 0

Opponent's probability distribution: 0.40, 0.50, 0.10
My choice: 0
  win - lose: 169 - 831
My choice: 1
  win - lose: 811 - 189
My choice: 2
  win - lose: 532 - 468
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: 843 - 157
My choice: 2
  win - lose: 412 - 588
My best choice: 1
```

```
Guessed choice: 1

Opponent's probability distribution: 0.50, 0.10, 0.40
My choice: 0
  win - lose: 817 - 183
My choice: 1
  win - lose: 529 - 471
My choice: 2
  win - lose: 157 - 843
My best choice: 0
Guessed choice: 1

Opponent's probability distribution: 0.10, 0.50, 0.40
My choice: 0
  win - lose: 447 - 553
My choice: 1
  win - lose: 209 - 791
My choice: 2
  win - lose: 819 - 181
My best choice: 2
Guessed choice: 2
```

- Applying *functional* decomposition can yield (differs from decomposition in 4-B):

In [4]:

```
# Compacted code (WARNING: violates Python Coding Standard; how so?)

Option = NewType('Option', int)
ROCK, PAPER, SCISSORS = Option(0), Option(1), Option(2)
OPTIONS: List[Option] = [ROCK, PAPER, SCISSORS]
option_str = {ROCK: "ROCK", PAPER: "PAPER", SCISSORS: "SCISSORS"}

Outcome = NewType('Outcome', int)
TIE = Outcome(0)
outcome_str = {TIE: "TIE", 1: "1 wins", 2: "2 wins"}

def judge_encounter(choice_1: Option, choice_2: Option) -> Outcome:
    return Outcome((choice_1 - choice_2) % len(OPTIONS))

Distribution = Sequence[float]

def choose_random(distr: Distribution) -> Option:
    return Option(random.choices(OPTIONS, weights=distr, k=1)[0])

ChoiceFunction = Callable[[], Option]

def play_game(choice_1: ChoiceFunction, choice_2: ChoiceFunction) -> Outco
me:
    result = TIE
    while result == TIE:
        result = judge_encounter(choice_1(), choice_2())
    return result

def play_games(n: int, choice_1: ChoiceFunction, choice_2: ChoiceFunction)
```

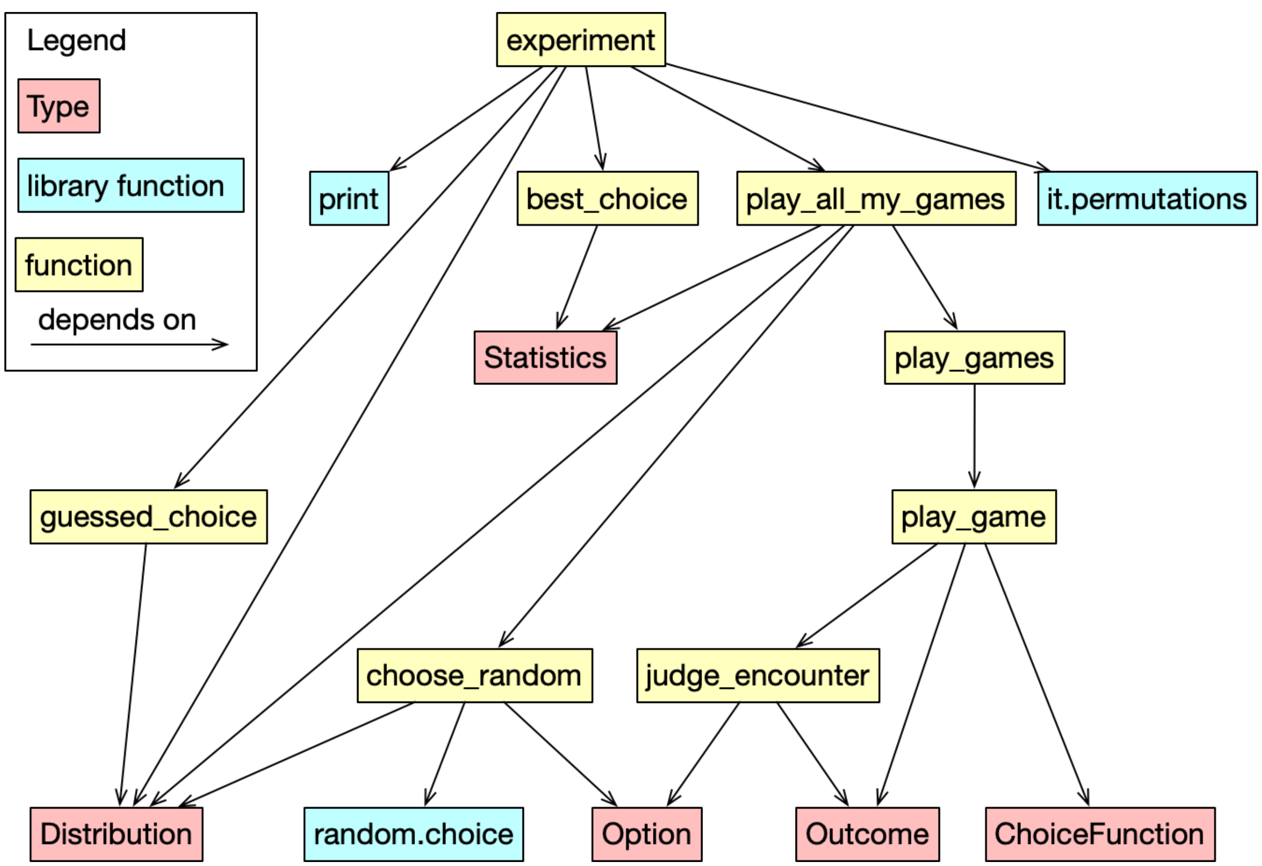
```python
 -> int:
    return sum(play_game(choice_1, choice_2) % 2 for _ in range(n))

Statistics = Mapping[Option, int]

def play_all_my_games(n: int, distr_2: Distribution) -> Statistics:
    return {choice_me: play_games(n, lambda: choice_me, lambda: choose_ran
dom(distr_2))
            for choice_me in OPTIONS}

def best_choice(wins: Statistics) -> Option:
    return max(OPTIONS, key=lambda x: wins[x])

def guessed_choice(distr: Distribution) -> Option:
    return Option((max(OPTIONS, key=lambda x: distr[x]) + 1) % len(OPTIONS
))

def experiment(distr_2: Distribution, n: int) -> None:
    for distr in it.permutations(distr_2):
        print("Opponent's probability distribution: {:1.2f}, {:1.2f}, {:1.
2f}".format(*distr))
        wins = play_all_my_games(n, distr)
        for choice_me, win in wins.items():
            print(f"I choose {option_str[choice_me]:10}: win - lose: {win}
 - {n - win}")
        print(f"My best and guessed choices: {option_str[best_choice(wins)
]} - {option_str[guessed_choice(distr)]}\n")

experiment([0.1, 0.4, 0.5], 1000)
```

```
Opponent's probability distribution: 0.10, 0.40, 0.50
I choose ROCK      : win - lose: 546 - 454
I choose PAPER     : win - lose: 189 - 811
I choose SCISSORS  : win - lose: 787 - 213
My best and guessed choices: SCISSORS - ROCK

Opponent's probability distribution: 0.10, 0.50, 0.40
I choose ROCK      : win - lose: 436 - 564
I choose PAPER     : win - lose: 216 - 784
I choose SCISSORS  : win - lose: 847 - 153
My best and guessed choices: SCISSORS - SCISSORS

Opponent's probability distribution: 0.40, 0.10, 0.50
I choose ROCK      : win - lose: 812 - 188
I choose PAPER     : win - lose: 441 - 559
I choose SCISSORS  : win - lose: 190 - 810
My best and guessed choices: ROCK - ROCK

Opponent's probability distribution: 0.40, 0.50, 0.10
I choose ROCK      : win - lose: 184 - 816
I choose PAPER     : win - lose: 803 - 197
I choose SCISSORS  : win - lose: 560 - 440
My best and guessed choices: PAPER - SCISSORS

Opponent's probability distribution: 0.50, 0.10, 0.40
I choose ROCK      : win - lose: 802 - 198
I choose PAPER     : win - lose: 553 - 447
```

```
I choose SCISSORS  : win - lose: 173 - 827
My best and guessed choices: ROCK - PAPER

Opponent's probability distribution: 0.50, 0.40, 0.10
I choose ROCK      : win - lose: 201 - 799
I choose PAPER     : win - lose: 846 - 154
I choose SCISSORS  : win - lose: 454 - 546
My best and guessed choices: PAPER - PAPER
```

- Do you understand `play_games`, using *generator expression*?
- Function `play_all_my_games` was decomposed further (SRP)
  - play `n` games for each option and return `Statistics` using a *dictionary comprehension*
  - functions `best_choice` and `guess_choice`
- All printing is now done in `experiment`
  - number of games is easier to change
  - the output is more compact

Functional Decompsition for RPS Experiment



- Compare top-down and bottom-up views
- This is only one design, of many
- Consequences for testing

Dependence diagrams for functions (like above) not used often

- But good designers 'see them' in their mind
- Can help
  - understanding
  - pinpoint 'hotspots' (big *fan out*)
  - decide on order of testing (*bottom up*)

`play_all_my_games()` depends on `Distribution` and `choose_random()`

- Can be avoided
- *Generalize* `play_all_my_games()`
- Instead, pass a `ChoiceFunction` for opponent

- One goal of functional decomposition:
  - facilitate reuse
- Let's try

New RPS Experiment

We want to compare

- a so-called *Markov Chain* Player:
  - never repeats previous choice
  - chooses uniformly between remaining options
  - put differently: probabilities depend on previous choice
- a player who chooses to beat opponent's previous choice

- `ChoiceFunction` is no longer applicable
- Choice depends on something else:
  - own previous choice, or
  - opponent's previous choice

- This could be solved by introducing two extra parameters in `ChoiceFunction`
  - own previous choice
  - opponent's previous choice

  But what if ... it depends on previous *two* choices, etc.?
- It can also be solved by using a *class*
- *Instance variables* keep track of 'past'
- Choice is returned by *method* using instance variables

In [5]:
```
class MarkovChainChoice:
    """A player who chooses uniformly
```

```
    among options different from previous choice.

    First time, chooses uniformly among all options.

    >>> player = MarkovChainChoice()
    >>> choice = player.choose()
    >>> choice in OPTIONS
    True
    >>> player.choose() != choice
    True
    """

    def __init__(self) -> None:
        """Initialize the player.
        """
        self.previous: Optional[Option] = None

    def choose(self) -> Option:
        """Return player's choice.
        """
        options = set(OPTIONS)
        if self.previous is not None:
            options.discard(self.previous)
        choice = random.choice(list(options))
        self.previous = choice
        return choice
```

In [6]:
```
doctest.run_docstring_examples(MarkovChainChoice, globs=globals(), name='MarkovChainChoice')
```

In [7]:
```
player = MarkovChainChoice()

[player.choose() for _ in range(20)]
```

Out[7]:
```
[1, 2, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2, 1, 0, 2, 1, 2, 0, 1, 2]
```

(How would you beat this player?)

In [8]:
```
class BeatPreviousChoice:
    """A player who chooses to beat
    opponent's previous choice.

    First time, chooses uniformly among all options.

    Usage:

    1. ``__init__()`` (via constructor)
    2. ``choose()``
    3. ``inform(...)``, repeat from 2

    >>> player = BeatPreviousChoice()
    >>> player.choose() in OPTIONS
    True
    >>> player.inform(ROCK)
    >>> player.choose() == PAPER
```

```
    True
    """

    def __init__(self):
        """Initalize the player.
        """
        self.opponent_previous: Optional[Option] = None

    def choose(self) -> Option:
        """Return player's choice.
        """
        if self.opponent_previous is None:
            choice = random.choice(OPTIONS)
        else:
            choice = Option((self.opponent_previous + 1) % len(OPTIONS))
            # could define a separate function for this

        return choice

    def inform(self, opponent_previous: Option) -> None:
        """Inform player of opponent's previous choice.
        """
        self.opponent_previous = opponent_previous
```

In [9]:
```
doctest.run_docstring_examples(BeatPreviousChoice, globs=globals(), name='
BeatPreviousChoice')
```

Function to play n games between these players

In [10]:
```
def play_games_(n: int,
                player_1: MarkovChainChoice,
                player_2: BeatPreviousChoice
               ) -> Counter[Outcome]:
    """Play n games between MarkovChainChoice player
    and BeatPreviousChoice player,
    returning outcome statistics.
    """
    result: Counter[Outcome] = Counter()

    for _ in range(n):
        choice_1 = player_1.choose()
        choice_2 = player_2.choose()
        outcome = judge_encounter(choice_1, choice_2)
        result[outcome] += 1
        player_2.inform(choice_1)
        # alternative (not recommended): player_2.opponent_previous = choi
ce_1

    return result
```

In [11]:
```
play_games_(1000, MarkovChainChoice(), BeatPreviousChoice()).most_common()
```

Out[11]:
```
[(0, 512), (1, 488)]
```

- Conclusion?

- How can you beat `BeatPreviousChoice` player even more?
- How can you beat `MarkovChainChoice` player?

Generalization

- Can `play_games` and `play_games_` be *unified*?
  - One function that generalizes both?

What every player needs:

- Ability to choose
- Ability to receive previous choice of opponent
  - can be ignored, if not needed

OOP: Subclasses and inheritance

Inheritance is OO mechanism to create *subclass*

- Subclass *inherits* all methods with definitions from *superclass*
- Subclass can *override* method inherited definitions
- Subclass can *introduce* other instance variables
- Subclass can *introduce* other methods

No copy-paste-edit involved; so, DRY (Don't Repeat Yourself)!

Abstract RPS player

Abstract superclass `Player`

- Not intended for instantiation
- Misses (some) method *definitions*
- Intended to be *subclassed*
- Each *concrete* player class inherits from `Player`
- Each *concrete* player object is also of type `Player`

In [12]:
```
class Player:
    """An abstract named player for Rock-Paper-Scissors.

    Usage:

    1. ``__init__()`` (via constructor)
    2. ``choose()``
    3. ``inform(...)``, repeat from 2
    """

    def __init__(self, name: str) -> None:
        """Initialize player with given name.
```

```
        """
        self.name = name

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

    def __str__(self) -> str:
        return self.name

    def choose(self) -> Option:
        """Choose from OPTIONS for this turn.
        """
        raise NotImplementedError("method is abstract")

    def inform(self, opponent_previous: Option) -> None:
        """Inform player of opponent's previous choice.
        """
        pass  # default behavior: ignore
```

General function to play RPS game

Define function to play a game between two players

- Definitions of `choose()` and `inform()` are not needed
- Only their type signatures matter

  N.B. Here we decided to play only a single encounter (ties will show up in statistics)

In [13]:
```
def play_encounter(player_1: Player, player_2: Player) -> Outcome:
    """Play one encounter between two players, returning outcome.
    """
    choice_1, choice_2 = player_1.choose(), player_2.choose()

    player_1.inform(choice_2)
    player_2.inform(choice_1)

    return judge_encounter(choice_1, choice_2)
```

In [14]:
```
play_encounter(Player("A"), Player("B"))  # fails
```

```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last
)
/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_80041/195765251
2.py in <module>
----> 1 play_encounter(Player("A"), Player("B"))  # fails

/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_80041/121251152
7.py in play_encounter(player_1, player_2)
      2     """Play one encounter between two players, returning outcome.
      3     """
----> 4     choice_1, choice_2 = player_1.choose(), player_2.choose()
      5
```

```
      6     player_1.inform(choice_2)

/var/folders/dq/bbdqxxcx30zdyfdd26t6vj4c0000gq/T/ipykernel_80041/188595021
.py in choose(self)
     23         """Choose from OPTIONS for this turn.
     24         """
---> 25         raise NotImplementedError("method is abstract")
     26 
     27     def inform(self, opponent_previous: Option) -> None:

NotImplementedError: method is abstract
```

In [15]:
```python
def play_games_OO(n: int,
                  player_1: Player,
                  player_2: Player
                 ) -> Counter[Outcome]:
    """Play n encounters between two players,
    returning outcome statistics.
    """
    return Counter(play_encounter(player_1, player_2)
                       for _ in range(n))
```

How to Define Subclass

Syntax:

```python
class SubClass(SuperClass):
     ...
```

Implement concrete players in subclass

- `ConstPlayer` (me, in first experiment)
- `RandomPlayer` (according to distribution)
- `MarkovPlayer` (accoording to Markov Chain Model)
- `BeatPreviousPlayer` (beat opponent's previous choice)

`ConstPlayer`

In [16]:
```python
class ConstPlayer(Player):
    """A constant Player who always chooses the same option.

    >>> player = ConstPlayer("Test", ROCK)
    >>> player
    ConstPlayer('Test', ROCK)
    >>> player.choose()
    0
    """

    def __init__(self, name: str, option: Option) -> None:
        """Initialize player for given option.
```

```
        """
        super().__init__(name)
        self.option = option

    def __repr__(self) -> str:
        return (f"{super().__repr__()}"[:-1] +
                f", {option_str[self.option]})"
               )

    # Overrides superclass definition
    def choose(self) -> Option:
        """Choose given option.
        """
        return self.option
```

In [17]:
```
doctest.run_docstring_examples(ConstPlayer, globs=globals(), name='ConstPl
ayer')
```

`RandomPlayer`

In [18]:
```
class RandomPlayer(Player):
    """A memoryless random Player, with given distribution.
    """

    def __init__(self, name: str, distr: Distribution) -> None:
        """Initialize player for given distribution.
        """
        super().__init__(name)
        self.distr = distr

    def __repr__(self) -> str:
        return (f"{super().__repr__()}"[:-1] +
                f", {self.distr!r})"
               )

    # Overrides superclass definition
    def choose(self) -> Option:
        """Make random choice according to given distribution.
        """
        return random.choices(OPTIONS, weights=self.distr, k=1)[0]
```

In [19]:
```
# manual smoke test
distr = [0.1, 0.4, 0.5]
ran_dom = RandomPlayer("Ran Dom", distr)

Counter(ran_dom.choose() for _ in range(1000)).most_common()
```

Out[19]:
```
[(2, 508), (1, 392), (0, 100)]
```

In [20]:
```
me = ConstPlayer("Me", guessed_choice(distr))
print(f"{me!r} vs {ran_dom!r}")
play_games_OO(1000, me, ran_dom).most_common()
```

```
ConstPlayer('Me', ROCK) vs RandomPlayer('Ran Dom', [0.1, 0.4, 0.5])
```

Out[20]: [(1, 514), (2, 383), (0, 103)]

Let's play all options for me:

In [21]:
```
stats = {option_str[option]: play_games_OO(1000,
                                            ConstPlayer("Me", option),
                                            ran_dom)
         for option in OPTIONS
        }
stats
```

Out[21]:
```
{'ROCK': Counter({2: 375, 1: 523, 0: 102}),
 'PAPER': Counter({1: 100, 0: 392, 2: 508}),
 'SCISSORS': Counter({0: 505, 1: 386, 2: 109})}
```

Now determine ratios of my wins against my losses:

In [22]:
```
{option: round(counts[Outcome(1)] / counts[Outcome(2)], 2)
 for option, counts in stats.items()
}
```

Out[22]: {'ROCK': 1.39, 'PAPER': 0.2, 'SCISSORS': 3.54}

So, apparently my best choice is 2 (`SCISSORS`), not 0 (`ROCK`)

In [23]:
```
# reverse sort on win/loss ratio (best at top)
sorted(((option_str[option], play_games_OO(1000,
                                            ConstPlayer("Me", option),
                                            ran_dom)
        )
        for option in OPTIONS
       ),
        key=lambda t: t[1][Outcome(1)] / t[1][Outcome(2)],
        reverse=True
       )
```

Out[23]:
```
[('SCISSORS', Counter({0: 474, 1: 417, 2: 109})),
 ('ROCK', Counter({2: 394, 1: 504, 0: 102})),
 ('PAPER', Counter({1: 87, 2: 497, 0: 416}))]
```

`MarkovPlayer`

Given: a distribution for each previous choice (by this player)

- That is, for each previous choice, there is a separate probability distribution for next choice
- A.k.a. *Markov Model of order 1*

In [24]:
```
MarkovModel_1 = Mapping[Option, Distribution]
```

In [25]:
```
class MarkovPlayer(Player):
    """A player who chooses according
```

```python
    to given order-1 Markov model.

    First time, chooses uniformly among all options.
    """

    def __init__(self, name: str, mm: MarkovModel_1) -> None:
        """Initialize the player for given Markov model.
        """
        super().__init__(name)
        self.mm = mm
        self.previous: Optional[Option] = None

    def choose(self) -> Option:
        """Return player's choice.
        """
        distr: Distribution  # (needed for type checking)
        if self.previous is None:
            distr = [1, 1, 1]
        else:
            distr = self.mm[self.previous]
        choice = random.choices(OPTIONS, weights=distr, k=1)[0]
        self.previous = choice
        return choice
```

In [26]:
```python
mm = {ROCK:     [0.0, 0.5, 0.5],
      PAPER:    [0.5, 0.0, 0.5],
      SCISSORS: [0.5, 0.5, 0.0],
     }
markov = MarkovPlayer("Mark Ov", mm)

[markov.choose() for _ in range(20)]
```

Out[26]:
```
[2, 1, 0, 2, 1, 0, 1, 0, 1, 0, 1, 2, 0, 1, 2, 1, 2, 0, 1, 2]
```

In [27]:
```python
# Overall statistics

Counter(markov.choose() for _ in range(10000))
```

Out[27]:
```
Counter({0: 3333, 2: 3342, 1: 3325})
```

Intermezzo: Uniform versus Independent

Two ways in which RPS choices can be 'bad':

- Not *uniformly* distributed
- Not *independently* distributed

- `RandomPlayer("Ran Dom", [0.1, 0.4, 0.5])`
  - *not* uniform
  - but all choices are independent (no memory effect)
- `MarkovPlayer("Mark Ov", mm)`
  - *not* independent (memory effect)
  - but choices are uniform (overall)

Both can be exploited

`BeatPreviousPlayer`

In [28]:
```
class BeatPreviousPlayer(Player):
    """A player who chooses to beat
    opponent's previous choice.

    First time, chooses uniformly among all options.

    >>> player = BeatPreviousPlayer("Test")
    >>> player.choose() in OPTIONS
    True
    >>> player.inform(ROCK)
    >>> player.choose() == PAPER
    True
    """

    def __init__(self, name: str):
        """Initalize the player.
        """
        super().__init__(name)
        self.opponent_previous: Optional[Option] = None

    def choose(self) -> Option:
        if self.opponent_previous is None:
            choice = random.choice(OPTIONS)
        else:
            choice = Option((self.opponent_previous + 1) % len(OPTIONS))

        return choice

    def inform(self, opponent_previous: Option) -> None:
        self.opponent_previous = opponent_previous
```

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

In [30]:
```
beat_previous = BeatPreviousPlayer("Beat Prev")

print(f"{markov!r} vs {beat_previous!r}")
play_games_OO(1000, markov, beat_previous)
```

```
MarkovPlayer('Mark Ov') vs BeatPreviousPlayer('Beat Prev')
```

Out[30]:
```
Counter({1: 498, 0: 502})
```

Notes about players

- `ConstPlayer` and `RandomPlayer` are *immutable*
- `MarkovPlayer` and `BeatPreviousPlayer` are *mutable*

- Many other strategies imaginable
- Track statistics of opponent's choices and create a Markov model to predict its behavior.

How would you play?

Can you imagine a way of playing an RPS *tournament*?

- All kinds of players play against each other

Polymorphism

Observe that `play_game_OO` needed no changes when introducing new types of players.

The parameters `player_1` and `player_2` of type `Player` accepted objects of *subclasses* of `Player` as well.

Parameters `player_1` and `player_2` are **polymorphic**.

Polymorphism ("taking on multiple forms"):

- the actual *run-time type* can be a *subclass* of the *declared type*

In [31]:
```
issubclass(ConstPlayer, Player)
```
Out[31]:
```
True
```

In [32]:
```
issubclass(ConstPlayer, RandomPlayer)
```
Out[32]:
```
False
```

In [33]:
```
type(me), isinstance(me, ConstPlayer), isinstance(me, Player)
```
Out[33]:
```
(__main__.ConstPlayer, True, True)
```

Notes about inheritance

- Use sparingly (prefer *composition*).
  - Only in case of 'is-a' relationship
  - A `RandomPlayer` is a (kind of) `Player`
- Inheritance can help avoid *code duplication* (DRY).

  Attributes and methods are inherited from *superclass* without copying code.

- Inheritance can be used to *add* attributes/methods.

- Inheritance can be used to *change* (**override**) method behavior.

  Use `super()` to invoke behavior of superclass.

More RPS Classes (via composition)

We can do further data decomposition

- `OutcomeStats`
  - holds count per outcome (composition)
  - can print nicely
  - can compute win fraction
- `Referee`
  - holds two players (composition)
  - can play one or more encounters

`OutcomeStats`

In [34]:
```python
WIN = Outcome(1)
LOSS = Outcome(2)

class OutcomeStats:
    """A count per outcome (mutable).

    >>> stats = OutcomeStats()
    >>> stats
    OutcomeStats(Counter())
    >>> print(stats)
    0 wins - 0 ties - 0 losses
    >>> stats.win_fraction()
    Traceback (most recent call last):
        ...
    AssertionError: No wins and losses
    >>> stats.update([WIN])
    >>> stats.win_fraction()
    1.0
    >>> stats.update([LOSS, TIE, LOSS, TIE, LOSS])
    >>> stats.win_fraction()
    0.25
    >>> print(stats)
    1 wins - 2 ties - 3 losses
    """

    def __init__(self, counts: Counter[Outcome] = None) -> None:
        self.counts: Counter[Outcome]  # (needed for type checking)
        if counts is None:
            self.counts = Counter()
        else:
            self.counts = counts

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

    def __str__(self) -> str:
        return ' - '.join([f"{self.counts[WIN]} wins",
                           f"{self.counts[TIE]} ties",
```

```python
                              f"{self.counts[LOSS]} losses",
                             ])

    def update(self, iterable: Iterable[Outcome]) -> None:
        """Update with all given outcomes.
        """
        self.counts.update(iterable)

    def win_fraction(self) -> float:
        """Return fraction win / (win + loss).
        """
        win_loss = self.counts[WIN] + self.counts[LOSS]
        assert  win_loss != 0, 'No wins and losses'
        return self.counts[WIN] / win_loss
```

In [35]:
```python
doctest.run_docstring_examples(OutcomeStats, globs=globals(), name='OutcomeStats')
```

`Referee`

In [36]:
```python
class Referee:
    """A referee for RPS games between two given players
    (immutable).
    """

    def __init__(self, player_1: Player, player_2: Player) -> None:
        """Initialize referee with two given players.
        """
        self.player_1 = player_1
        self.player_2 = player_2

    def play_encounter(self) -> Outcome:
        """Play one encounter between the two players,
        returning outcome.
        """
        choice_1 = self.player_1.choose()
        choice_2 = self.player_2.choose()

        self.player_1.inform(choice_2)
        self.player_2.inform(choice_1)

        return judge_encounter(choice_1, choice_2)

    def play_games(self, n: int, verbose=False) -> OutcomeStats:
        """Play n encounters between the two players,
        returning outcome statistics.
        """
        stats = OutcomeStats()
        stats.update(self.play_encounter()
                     for _ in range(n))
        if verbose:
            print(f"{self.player_1!r} vs {self.player_2!r}")
            print(stats)
            print(f"win fraction: {stats.win_fraction():1.2f}")
```

```python
        return stats
```

In [37]:
```python
referee = Referee(markov, beat_previous)

referee.play_games(1000, True)
```

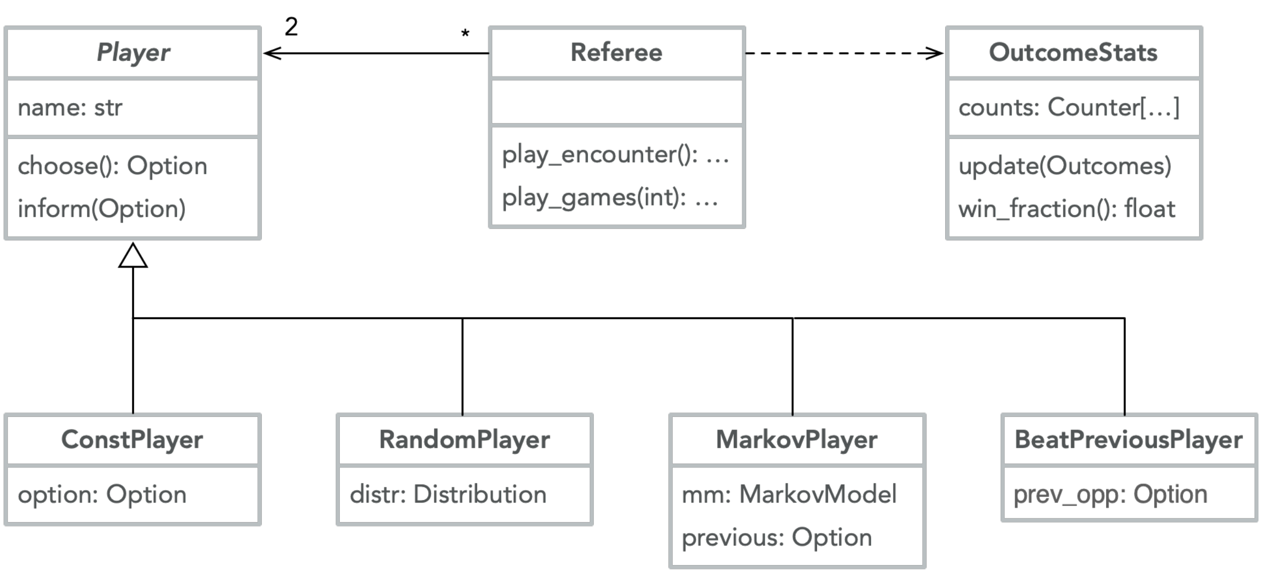
```
MarkovPlayer('Mark Ov') vs BeatPreviousPlayer('Beat Prev')
477 wins - 523 ties - 0 losses
win fraction: 1.00
```

Out[37]:
```
OutcomeStats(Counter({0: 523, 1: 477}))
```

In [38]:
```python
for option in OPTIONS:
    referee = Referee(ConstPlayer("Me", option),
                      RandomPlayer("Ran Dom", [0.1, 0.4, 0.5])
                     )
    referee.play_games(1000, True)
    print()
```

```
ConstPlayer('Me', ROCK) vs RandomPlayer('Ran Dom', [0.1, 0.4, 0.5])
501 wins - 92 ties - 407 losses
win fraction: 0.55

ConstPlayer('Me', PAPER) vs RandomPlayer('Ran Dom', [0.1, 0.4, 0.5])
101 wins - 393 ties - 506 losses
win fraction: 0.17

ConstPlayer('Me', SCISSORS) vs RandomPlayer('Ran Dom', [0.1, 0.4, 0.5])
387 wins - 508 ties - 105 losses
win fraction: 0.79
```

Data Decompsition for RPS Experiments



Dependence diagrams for classes (like above) often used

- Diagram notation: *Unified Modeling Language* (UML)

- Note how all experiments were supported
- Recommendation: *encapsulate* `Distribution` and `MarkovModel` in a class

Also see: [Inheritance and Composition: A Python OOP Guide]

Argument Gathering

Goals:

- Call a function having multiple parameters, such that *each argument is taken from a sequence or dictionary*.
- Define a function with a *variable number of arguments*.

Argument unpacking for positional arguments

Example: [Multiply-accumulate operation]

- `a += b * c`

In [39]:
```
def mac(a: float, b: float, c: float) -> float:
    """Multiply accumulate operation.

    >>> mac(0, 1, 2)
    2
    >>> mac(1, 2, 3)
    7
    """
    return a + b * c
```

In [40]:
```
doctest.run_docstring_examples(mac, globals(), True, 'mac')
```

```
Finding tests in mac
Trying:
    mac(0, 1, 2)
Expecting:
    2
ok
Trying:
    mac(1, 2, 3)
Expecting:
    7
ok
```

In [41]:
```
args = [2, 3, 4]

# we want to do mac(args[0], args[1], args[2])
mac(*args)  # NOTE the *
```

Out[41]:
```
14
```

In [42]:
```
args = (mac, globals(), True, 'mac')
```

```
doctest.run_docstring_examples(*args)   # NOTE the *
```

```
Finding tests in mac
Trying:
    mac(0, 1, 2)
Expecting:
    2
ok
Trying:
    mac(1, 2, 3)
Expecting:
    7
ok
```

Argument unpacking for keyword arguments

In [43]:
```
kwargs: Mapping[str, Any] = {
    'f': mac,
    'globs': globals(),
    'verbose': True,
    'name': 'mac'
}

doctest.run_docstring_examples(**kwargs)   # NOTE the **
```

```
Finding tests in mac
Trying:
    mac(0, 1, 2)
Expecting:
    2
ok
Trying:
    mac(1, 2, 3)
Expecting:
    7
ok
```

Can also be mixed.

Positional arguments always precede keyword arguments:

In [44]:
```
args = (mac, globals())
kwargs = {'verbose': True, 'name': 'mac'}

doctest.run_docstring_examples(*args, **kwargs)   # NOTE the * and **
```

```
Finding tests in mac
Trying:
    mac(0, 1, 2)
Expecting:
    2
ok
Trying:
    mac(1, 2, 3)
```

```
Expecting:
    7
ok
```

Functions with variable number of arguments

In [45]:

```python
def print_args(*args: Any, **kwargs: Any) -> None:
    """Print each positional and keyword argument.

    >>> print_args('a', 'b')
    position 0 = 'a'
    position 1 = 'b'
    >>> print_args(42, a=1)
    position 0 = 42
    a = 1
    >>> print_args(first=1, second=2)
    first = 1
    second = 2
    """
    for index, arg in enumerate(args):
        print(f"position {index} = {arg!r}")
    for kw, val in kwargs.items():
        print(f"{kw} = {val!r}")
```

In [46]:

```python
args = (print_args, globals())
kwargs = {'verbose': True, 'name': 'print_args'}

doctest.run_docstring_examples(*args, **kwargs)
```

```
Finding tests in print_args
Trying:
    print_args('a', 'b')
Expecting:
    position 0 = 'a'
    position 1 = 'b'
ok
Trying:
    print_args(42, a=1)
Expecting:
    position 0 = 42
    a = 1
ok
Trying:
    print_args(first=1, second=2)
Expecting:
    first = 1
    second = 2
ok
```

Recursion

Recursive function: defined (directly or indirectly) in terms of itself

Image source: https://en.wikipedia.org/wiki/Droste_effect

In [47]:
```
def print_triangle_recursive(n: int) -> None:
    """Print an o-triangle with base n.

    Assumption: n >= 0
```

```
    >>> print_triangle_recursive(3)
    ooo
    oo
    o
    """
    if n == 0:
        pass  # done
    else:
        print(n * "o")
        print_triangle_recursive(n - 1)
```

In [48]:
```
print_triangle_recursive(5)
```

```
ooooo
oooo
ooo
oo
o
```

Leap of faith

To understand this definition:

- Don't try to play out the possible executions of all the recursive calls.
- Rather, make a **leap of faith** (cf. *Think Python*, Section 6.6):
  - Understand that the function works correctly, *under the assumption* that the function already works for 'smaller' values of the parameters
- Compare this to a **proof by induction**:
  - the assumption serves as **induction hypothesis**

Notes

- Recursive definitions are like `while` loops:
  - They can lead to *infinite* computations;
  - You have to ensure **termination**

Recursion for variable number of nested loops

All 4-bit binary numbers can be generated using 4 nested loops:

In [49]:
```
BIT = range(2)

def binary_numbers_4() -> Iterator[Tuple[int, int, int, int]]:
    """Yield all 4-bit tuples, in lexicographic order.
    """
    for b1 in BIT:
        for b2 in BIT:
            for b3 in BIT:
```

```
                for b4 in BIT:
                    yield b1, b2, b3, b4
```

In [50]:
```
for t in binary_numbers_4():
    print(t)
```

```
(0, 0, 0, 0)
(0, 0, 0, 1)
(0, 0, 1, 0)
(0, 0, 1, 1)
(0, 1, 0, 0)
(0, 1, 0, 1)
(0, 1, 1, 0)
(0, 1, 1, 1)
(1, 0, 0, 0)
(1, 0, 0, 1)
(1, 0, 1, 0)
(1, 0, 1, 1)
(1, 1, 0, 0)
(1, 1, 0, 1)
(1, 1, 1, 0)
(1, 1, 1, 1)
```

In [51]:
```
print(*binary_numbers_4(), sep='\n')
```

```
(0, 0, 0, 0)
(0, 0, 0, 1)
(0, 0, 1, 0)
(0, 0, 1, 1)
(0, 1, 0, 0)
(0, 1, 0, 1)
(0, 1, 1, 0)
(0, 1, 1, 1)
(1, 0, 0, 0)
(1, 0, 0, 1)
(1, 0, 1, 0)
(1, 0, 1, 1)
(1, 1, 0, 0)
(1, 1, 0, 1)
(1, 1, 1, 0)
(1, 1, 1, 1)
```

Generate all n-bit binary tuples

We want to define the following function:

In [52]:
```
def binary_numbers(n: int) -> Iterator[Tuple[int, ...]]:
    """Yield all n-bit binary tuples in lexicographic order.

    Assumptions:

    * n >= 0

    >>> list(binary_numbers(0))
```

```
    [()]
    >>> binary_numbers(2)
    [(0, 0), (0, 1), (1, 0), (1, 1)]
    """
```

In a way, we need a *variable number* of nested `for`-loops

Can be achieved via:

- Recursion

Note the *recursive pattern* in the desired output

In [53]:
```python
for index, t in enumerate(binary_numbers_4()):
    print("{}  {}{}{}".format(*t),
          end='\n\n' if index == 7 else '\n')
```

```
0  000
0  001
0  010
0  011
0  100
0  101
0  110
0  111

1  000
1  001
1  010
1  011
1  100
1  101
1  110
1  111
```

In [54]:
```python
def binary_numbers(n: int) -> Iterator[Tuple[int, ...]]:
    """Yield all n-bit binary tuples in lexicographic order.

    Assumptions:

    * n >= 0

    >>> list(binary_numbers(0))
    [()]
    >>> list(binary_numbers(2))
    [(0, 0), (0, 1), (1, 0), (1, 1)]
    """
    if n == 0:
        # base case
        yield ()
    else:
        # inductive step
        for b in BIT:
            for t in binary_numbers(n - 1):
```

```
                yield (b, ) + t
```

In [55]:
```
doctest.run_docstring_examples(binary_numbers, globs=globals(), name='binary_numbers')
```

In [56]:
```
for u in binary_numbers(4):
    print(*u)
```

```
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
```

Branching recursion

- Each call, except base case,
  - results in *two* recursive calls
- *Exponential growth*
- Jargon: *backtracking*, *exhaustive search*

In [57]:
```
def binary_numbers_(n: int, indent='') -> Iterator[Tuple[int, ...]]:
    """Yield all n-bit binary tuples in lexicographic order.

    Assumptions: n >= 0
    """
    print(indent, f"binary_numbers_({n})", sep='')
    if n == 0:
        # base case
        yield ()
    else:
        # inductive step
        for b in BIT:
            for t in binary_numbers_(n - 1, indent + 4 * ' '):
                yield (b, ) + t
```

In [58]:
```
for u in binary_numbers_(4):
    pass
```

```
binary_numbers_(4)
    binary_numbers_(3)
        binary_numbers_(2)
```

```
            binary_numbers_(1)
                binary_numbers_(0)
                binary_numbers_(0)
            binary_numbers_(1)
                binary_numbers_(0)
                binary_numbers_(0)
        binary_numbers_(2)
            binary_numbers_(1)
                binary_numbers_(0)
                binary_numbers_(0)
            binary_numbers_(1)
                binary_numbers_(0)
                binary_numbers_(0)
    binary_numbers_(3)
        binary_numbers_(2)
            binary_numbers_(1)
                binary_numbers_(0)
                binary_numbers_(0)
            binary_numbers_(1)
                binary_numbers_(0)
                binary_numbers_(0)
        binary_numbers_(2)
            binary_numbers_(1)
                binary_numbers_(0)
                binary_numbers_(0)
            binary_numbers_(1)
                binary_numbers_(0)
                binary_numbers_(0)
```

Another *recursive pattern*

In [59]:

```
for index, t in enumerate(binary_numbers_4()):
    print("{}{}{}  {}".format(*t),
          end='\n\n' if index % 2 == 1 else '\n')
```

```
000  0
000  1

001  0
001  1

010  0
010  1

011  0
011  1

100  0
100  1

101  0
101  1

110  0
110  1
```

```
111  0
111  1
```

In [60]:
```python
def binary_numbers_2(n: int) -> Iterator[Tuple[int, ...]]:
    """Yield all n-bit binary tuples in lexicographic order.

    Assumptions:

    * n >= 0

    >>> list(binary_numbers_2(0))
    [()]
    >>> list(binary_numbers_2(2))
    [(0, 0), (0, 1), (1, 0), (1, 1)]
    """
    if n == 0:
        # base case
        yield ()
    else:
        # inductive step
        for t in binary_numbers_2(n - 1):
            for b in BIT:
                yield t + (b, )
```

In [61]:
```python
doctest.run_docstring_examples(binary_numbers_2, globs=globals(), name='bi
nary_numbers_2')
```

In [62]:
```python
for u in binary_numbers_2(4):
    print(*u)
```

```
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
```

Generalized problem

Observe another pattern:

In [63]:
```python
for index, t in enumerate(binary_numbers_4()):
    print("{}{}  {}{}".format(*t),
```

```
            end='\n\n' if index % 4 == 3 else '\n')
```

```
00  00
00  01
00  10
00  11

01  00
01  01
01  10
01  11

10  00
10  01
10  10
10  11

11  00
11  01
11  10
11  11
```

- Function `binary_numbers_gen(n, t)`
  - generates all binary tuples in increasing order
  - that *extend* given tuple `t` with `n` bits,
- Assumption: `n >= 0`

Function `binary_numbers_gen` *generalizes* `binary_numbers`:

- To get original problem, take `t == ()`
- `binary_numbers_gen(n, ())` yields the same as `binary_numbers(n)`
- `binary_numbers` is *special case* of `binary_numbers_gen`

Solution:

- Do induction on `n`
- Base case: `n == 0`, then only `t` generated
- Inductive step: `n > 0`
  - Induction hypothesis: recursive call with smaller `n` 'works' as expected
  - Extend `t` in all possible ways with *one* bit `b`: `t + (b, )`
  - Call function with `n - 1` and `t + (b, )`

In [64]:
```
def binary_numbers_gen(n: int, t: Tuple[int, ...] = ()) -> Iterator[Tuple[
int, ...]]:
    """Yield all binary numbers that extend t with n bits,
    in increasing order.

    Assumptions:

    * n >= 0
```

```
    >>> list(binary_numbers_gen(0))
    [()]
    >>> list(binary_numbers_gen(2, (0, 1)))
    [(0, 1, 0, 0), (0, 1, 0, 1), (0, 1, 1, 0), (0, 1, 1, 1)]
    """
    if n == 0:
        # base case
        yield t
    else:
        # inductive step
        for b in BIT:
            yield from binary_numbers_gen(n - 1, t + (b, ))
```

In [65]:
```
doctest.run_docstring_examples(binary_numbers_gen, globals(), verbose=False, name='binary_numbers_gen')
```

Syntax:

```
yield from iterable
```

Semantics:

```
for item in iterable
    yield item
```

In [66]:
```
for u in binary_numbers_gen(4):
    print(*u)
```

```
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
```

Solution from Python Standard Library

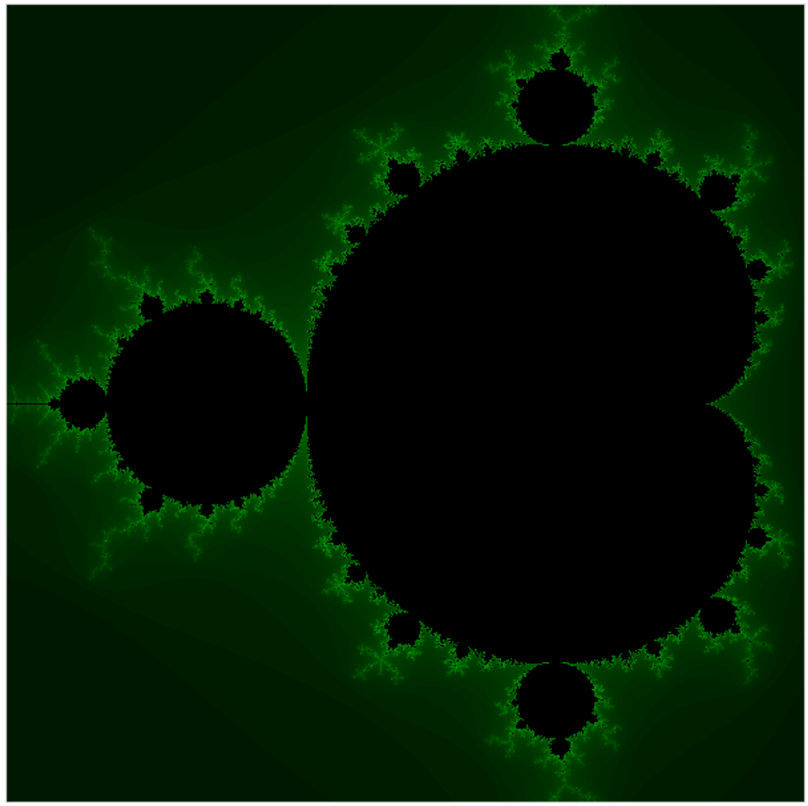
In [67]:
```
for u in it.product(BIT, repeat=4):
    print(*u)
```

```
0 0 0 0
0 0 0 1
```

```
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
```

(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

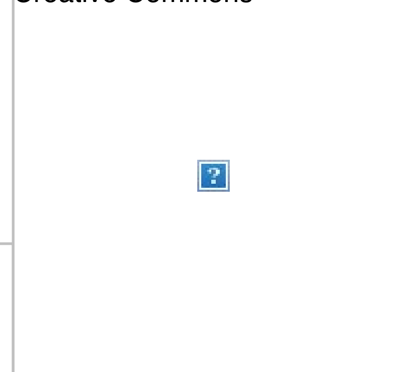
choose appropriate license
    - must agree with licenses of third-party software you use
- as user
    - read license of software you use
    - adhere to license of software you use

Differences across the globe:

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

Open-Source Software Licenses

Creative Commons



- Many flavors
- Subtle differences
- Public Domain
- Creative Commons, several 'levels'
- Permissive licenses
    - BSD, Apache, MIT, ...

Copyleft

- GPL and LGPL
    - GNU General Public License
    - GNU Library/Lesser GPL
    - **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]:
```python
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)."""
```

```
        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]:
```
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 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]:
```
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]:
```
print("".join(char for char in it.islice(mm_en_5, 200)))
```

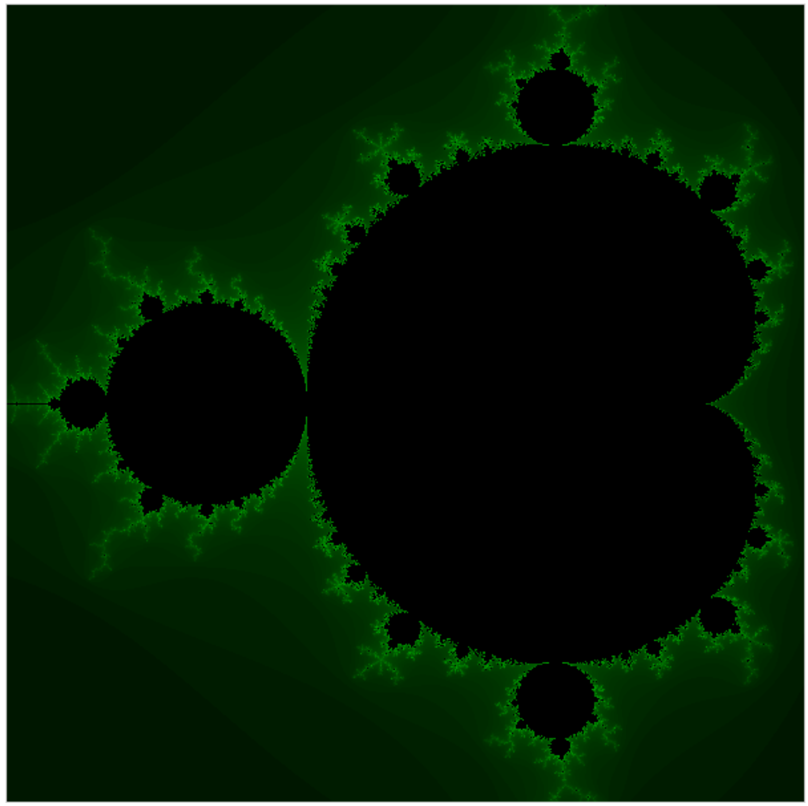
```
tch."

"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)

2IS50 – Software Development for Engineers – 2022-2023



Lecture 7 (Extra)

Lecturer: Tom Verhoeff

Preview of Lecture 7 (Extra)

- Course summary
- Persisting data with `pickle`
- Floating-point concerns
- Difficulty of computational problems
- Bonus

Course Summary

- Programming: concepts, terminology
- Python: syntax, semantics, pragmatics
  - See official [Python documentation]
  - Python Standard Library
- Write *clean* code: **Coding Standard**
- Organize your code
  - Variables, expresssions, assignment, `if`, `for`, `while`
  - Functional decomposition, `def`

    - Data decomposition, `class`
- Avoid **code duplication** and **recomputation**
    - Introduce auxiliary variables and functions
- **Document** your code
    - type hints, docstrings, `doctest` examples
- **Test** your code
- Some algorithms, efficiency, recursion

- [Repository with study material]()
- Study the book *Think Python* (2e), by Allen Downey

In [1]:
```
# 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")
```

```
nb-mypy.py not installed
```

In [2]:
```
# Preliminaries

import collections as co
from typing import Tuple, List, Set, Dict, DefaultDict, Counter
from typing import Any, Optional
from typing import Sequence, Mapping, MutableMapping, Iterable, Iterator,
Callable
from typing import NewType, TypeVar
import math
import random
from pprint import pprint
import itertools as it
import doctest
```

Persisting Data via `pickle`

- Data in variables gets lost when you close a notebook/program.

- To persist data, save it to a file or database.

- There are many formats:
    - Custom format in text file: `open`, `read`, `write`
    - Pickle (Python Object Serialization): `import pickle`
    - CSV (Comma-Separated Valued): `import csv`
    - JSON (JavaScript Object Notation): `import json`
    - ...

In [3]:
```
import pickle
```

Define some data:

In [4]:
```
data = ["test", 42, math.pi]
data
```

Out[4]:
```
['test', 42, 3.141592653589793]
```

Open *binary* file for writing and **pickle** `data` :

In [5]:
```
with open('test.pk', 'wb') as f:
    pickle.dump(data, f)
```

Open *binary* file for reading and **unpickle** its content:

In [6]:
```
with open('test.pk', 'rb') as f:
    data2 = pickle.load(f)

data2
```

Out[6]:
```
['test', 42, 3.141592653589793]
```

In [7]:
```
data == data2
```

Out[7]:
```
True
```

Floating-Point Concerns

- Floating-point arithmetic *approximates* real-number arithmetic.
- They are not the same; not even *homomorphic*.

IEEE-754 Standard for Floating-Point Arithmetic

- Open standard
- Includes positive and negative *infinite* values
- Distinguishes positive and negative zero
- Has rules to calculate with these values consistently

You can get this in Python through the **Numpy** library

Binary notation

Floating point numbers are stored in *binary notation*

Python can print them in *hexadecimal* with `float.hex()`

- base 16, groups of 4 bits
- `...p...` stands for $\fbox{$\ldots \times 2^{ \ldots}$}$
  - shifts binary point to the right

inverse is `float.fromhex(...)`

In [8]:
```
for i in range(0, 33+1):
    print(f"{i:2} {i:6b}", float(i).hex())
```

```
 0      0 0x0.0p+0
 1      1 0x1.0000000000000p+0
 2     10 0x1.0000000000000p+1
 3     11 0x1.8000000000000p+1
 4    100 0x1.0000000000000p+2
 5    101 0x1.4000000000000p+2
 6    110 0x1.8000000000000p+2
 7    111 0x1.c000000000000p+2
 8   1000 0x1.0000000000000p+3
 9   1001 0x1.2000000000000p+3
10   1010 0x1.4000000000000p+3
11   1011 0x1.6000000000000p+3
12   1100 0x1.8000000000000p+3
13   1101 0x1.a000000000000p+3
14   1110 0x1.c000000000000p+3
15   1111 0x1.e000000000000p+3
16  10000 0x1.0000000000000p+4
17  10001 0x1.1000000000000p+4
18  10010 0x1.2000000000000p+4
19  10011 0x1.3000000000000p+4
20  10100 0x1.4000000000000p+4
21  10101 0x1.5000000000000p+4
22  10110 0x1.6000000000000p+4
23  10111 0x1.7000000000000p+4
24  11000 0x1.8000000000000p+4
25  11001 0x1.9000000000000p+4
26  11010 0x1.a000000000000p+4
27  11011 0x1.b000000000000p+4
28  11100 0x1.c000000000000p+4
29  11101 0x1.d000000000000p+4
30  11110 0x1.e000000000000p+4
31  11111 0x1.f000000000000p+4
32 100000 0x1.0000000000000p+5
33 100001 0x1.0800000000000p+5
```

In [9]:
```
for e in range(0, 3+1):
    x = 1 / 2 ** e
    print(f"1/{2 ** e} {x:5.3f}", float(x).hex())
```

```
1/1 1.000 0x1.0000000000000p+0
1/2 0.500 0x1.0000000000000p-1
1/4 0.250 0x1.0000000000000p-2
1/8 0.125 0x1.0000000000000p-3
```

Smallest *positive* `float`

Making it smaller yields \$0\$ (underflow)

In [10]:
```
min_float, e = 1.0, 0  # invariant: min_float = 2 ** e
```

```
while min_float / 2 != 0.0:
    min_float /= 2
    e -= 1

print(min_float, min_float.hex(), e)
print(min_float / 2)   # underflow
```

```
5e-324 0x0.0000000000001p-1022 -1074
0.0
```

Largest `float`

However you try to make it larger

- either it stays the *same*
- or it becomes *infinity* (overflow)

In [11]:

```
max_float, e = 1.0, 0   # invariant: max_float = 2 ** e

while max_float * 2 != math.inf:
    max_float *= 2
    if max_float + 1 - 1 == max_float:
        max_float += 1
    e += 1

print(max_float, max_float.hex(), e)
print(2 * max_float)   # overflow
```

```
1.7976931348623157e+308 0x1.fffffffffffffp+1023 1023
inf
```

Range of `float` from small to large:

- spans over 600 *orders of magnitude*
- more than enough for *science* and *engineering*

Machine Precision

Smallest positive `float` ε such that `1.0 +`ε`> 1.0`

- smallest relative *step size* between `float` numbers
- difference between 1.0 and next `float` > 1.0

In [12]:

```
mach_prec, e = 1.0, 0   # invariant: mach_prec = 2 ** e

while 1.0 + mach_prec / 2 != 1.0:
    mach_prec /= 2
    e -= 1

print(1.0 + mach_prec, (1.0 + mach_prec).hex())
```

```
print(mach_prec, mach_prec.hex(), e)
```

```
1.0000000000000002 0x1.0000000000001p+0
2.220446049250313e-16 0x1.0000000000000p-52 -52
```

Machine precision: roughly *one nanosecond* (10^{-9}s) on scale of *one year* (3×10^{7}s) $$\begin{array}{lll} \text{Factor} & \text{Name} \\ \hline 10^{-3} & \text{milli} \\ 10^{-6} & \text{micro} \\ 10^{-9} & \text{nano} \\ 10^{-12} & \text{pico} \\ 10^{-15} & \text{femto} \end{array}$$

- In Python >= 3.9: see `math.nextafter()` and `math.ulp()`
- In Numpy: see `numpy.nextafter()`

In [13]:
```
import numpy as np

np.nextafter(1.0, np.inf), np.nextafter(1.0, np.inf) - 1.0
```

Out[13]:
```
(1.0000000000000002, 2.220446049250313e-16)
```

Largest `float` that cannot be incremented by 1

`x: int` such that `float(x) + 1 = float(x)`

In [14]:
```
max_int, e = 1.0, 0  # invariant: max_int = 2 ** e

while max_int + 1 != max_int:
    max_int *= 2
    e += 1

print(max_int, f"{max_int:e}", max_int.hex(), e)
print(max_int + 1)  # disappears after rounding
```

```
9007199254740992.0 9.007199e+15 0x1.0000000000000p+53 53
9007199254740992.0
```

In [15]:
```
np.nextafter(max_int, np.inf)
```

Out[15]:
```
9007199254740994.0
```

Rounding

Float value `0.1` is not *exactly* $\displaystyle\frac{1}{10}$:

In [16]:
```
TENTH = 0.1

f"{TENTH:.20e}"
```

Out[16]:
```
'1.00000000000000005551e-01'
```

The representation of `0.1` in *hexadecimal* (base 16)

In [17]:
```
TENTH.hex()
```
Out[17]:
```
'0x1.999999999999ap-4'
```

In binary: $0.0001\,1001\,1001\,1001\,\cdots\,1001\,1010$

Hexadecimal representation ends in `a` (1010 in binary)

- it was rounded *up* (from `9`)

`0.1` converts to a binary floating-point number

- that is a tad *larger* than $\frac{1}{10}$,

When you add ten copies, the result is a tad *smaller* than 1.0!

See if you can understand why.

Here are the ten intermediate results

- in *decimal* (approximate: conversion from internal floating-point format to decimal for printing)
- in hexadecimal (exact):

In [18]:
```
r = 0.0

for _ in range(10):
    r += TENTH
    print(f"{r:.20f} {r.hex():22}")
```
```
0.10000000000000000555 0x1.999999999999ap-4
0.20000000000000001110 0x1.999999999999ap-3
0.30000000000000004441 0x1.3333333333334p-2
0.40000000000000002220 0x1.999999999999ap-2
0.50000000000000000000 0x1.0000000000000p-1
0.59999999999999997780 0x1.3333333333333p-1
0.69999999999999995559 0x1.6666666666666p-1
0.79999999999999993339 0x1.9999999999999p-1
0.89999999999999991118 0x1.cccccccccccccp-1
0.99999999999999988898 0x1.fffffffffffffp-1
```

- Floating-point operations are *not exact*
  - but involve *rounding*

Cancelation

In [19]:
```
for e in range(6+1):
    x = 10.0 ** e
```

```
    y = (x + TENTH) - x

    print(f"{x:9} {y:1.18f} {y == TENTH}")
```

```
      1.0 0.100000000000000089 False
     10.0 0.099999999999999645 False
    100.0 0.099999999999994316 False
   1000.0 0.100000000000022737 False
  10000.0 0.100000000000363798 False
 100000.0 0.100000000005820766 False
1000000.0 0.099999999976716936 False
```

In [20]:

```
print(TENTH.hex())
for e in range(6+1):
    x = 10.0 ** e
    y = x + TENTH - x
    print(f"{x:>9} {x.hex():<21} {y.hex():<21} {y == TENTH}")
```

```
0x1.999999999999ap-4
      1.0 0x1.0000000000000p+0  0x1.99999999999a0p-4  False
     10.0 0x1.4000000000000p+3  0x1.9999999999980p-4  False
    100.0 0x1.9000000000000p+6  0x1.9999999999800p-4  False
   1000.0 0x1.f400000000000p+9  0x1.999999999a000p-4  False
  10000.0 0x1.3880000000000p+13 0x1.99999999a0000p-4  False
 100000.0 0x1.86a0000000000p+16 0x1.9999999a00000p-4  False
1000000.0 0x1.e848000000000p+19 0x1.9999999800000p-4  False
```

Note that value of `y` deviates more and more from \$0.1\$

- *Least significant bits* of `y` are zeroed by large `x`
- A.k.a. **cancelation**

Intermezzo: Closures

In [21]:

```
def poly(*coefficients: float) -> Callable[[float], float]:
    """Return polynomial with given coefficients.

    >>> poly()(1)
    0.0
    >>> poly(3)(1)   # 3 (constant polynomial)
    3.0
    >>> poly(2, 1)(1)   # 2*1 + 1 (linear)
    3.0
    >>> poly(1, 2, -1)(3)   # 3^2 + 2*3 - 1
    14.0
    """

    def f(x: float) -> float:
        """Evaluate polynomial with given coefficients.

        Uses Horner's scheme (to reduce number of multiplications)
        """
        result = 0.0
```

```python
        for c in coefficients:
            result = result * x + c

        return result

    return f
```

In [22]:
```python
doctest.run_docstring_examples(poly, globs=globals(), name='poly')
```

Note that definition of function `f` involves `coefficients`

- these are defined *outside* `f`
- these are needed when calling the returned `f`

This returned `f` *binds* `coefficients`

- Such an `f` is known as a **closure**

Quadratic Equation

Consider this problem:

- given $a, b, c \in \mathbb{R}$ with $a > 0$ and $b^2 > 4ac$
- finding smallest solution $x \in \mathbb{R}$ such that $a x^2 + b x + c = 0$

Mathematical solution (Quadratic Formula or abc-formula):

- $x = \mathcal{A}(a, b, c)$ where

$$\mathcal{A}(a, b, c) = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

or equivalently (algebra!)
$$\mathcal{A}(a, b, c) = \frac{2c}{-b + \sqrt{b^2 - 4ac}}$$

Python solutions $\widehat{\mathcal{A}}$ and $\widehat{\mathcal{A}}'$ (`a_hat` and `a_hat_`)

In [23]:
```python
def a_hat(a: float, b: float, c: float) -> float:
    """Compute approximation of smallest solution x of poly(a, b, c)(x) == 0.

    Assumptions:

    * a > 0
    * b ** 2 > 4 * a * c
    """
    return (-b - math.sqrt(b ** 2 - 4 * a * c)) / (2 * a)
```

In [24]:
```python
coefficients = 1.0, -1e6, 1.0
quadratic = poly(*coefficients)

x = a_hat(*coefficients)
```

```
x
```

Out[24]: 1.000007614493337e-06

In [25]:
```
quadratic(x)
```

Out[25]: -7.614492369967252e-06

In [26]:
```
def a_hat_(a: float, b: float, c: float) -> float:
    """Compute approximation of smallest solution x of poly(a, b, c)(x) ==
 0,
    using alternative abc-formula.

    Assumptions:

    * a > 0
    * b ** 2 > 4 * a * c
    """
    return (2 * c) / (-b + math.sqrt(b ** 2 - 4 * a * c))
```

In [27]:
```
x_ = a_hat_(*coefficients)
x_
```

Out[27]: 1.000000000001e-06

In [28]:
```
quadratic(x_)
```

Out[28]: 0.0

Explanation:

- $b < 0$
- $-b$ is large compared to a and c
- So, $-b$ and $\sqrt{b^2 - 4ac}$ are roughly equal, with *same* sign
- Their *difference* loses significant digits due to *cancelation*
- In alternative formula, these are *added*: no cancelation

However, for *positive* b:

- `a_hat` is okay
- `a_hat_` suffers from cancelation

In [29]:
```
coefficients_ = 1.0, 1e6, 1.0
quadratic_ = poly(*coefficients_)

x = a_hat(*coefficients_)
x_ = a_hat_(*coefficients_)

print(f"{x:16.6f}", quadratic_(x))
print(f"{x_:16.6f}", quadratic_(x_))  # useless!
```

```
  -999999.999999 -7.614492369967252e-06
  -999992.385565 -7614376.410360885
```

Solving quadratic equations is hard!

- Don't just use the (a,b,c)-formula
- Take a course on *Scientific Computing*

The following diagram does *not* commute:

$$\begin{tikzcd}
\mathbb{R}^n \arrow[r, "\mathit{fl}^n"] \arrow[d, "\mathcal{A}"] & \mathbb{F}^n \arrow[d, "\widehat{\mathcal{A}}"] \\
\mathbb{R} \arrow[r, "\mathit{fl}"] & \mathbb{F}
\end{tikzcd}$$

- fl maps real number to its best floating-point approximation
- $\mathcal{A}$ is real-valued function of n real numbers.
- $\widehat{\mathcal{A}}$ is floating-point version of $\mathcal{A}$ (it involves rounding)
- No guarantee that $\mathit{fl}(\mathcal{A}(x, y, \ldots)) = \widehat{\mathcal{A}}(\mathit{fl}(x), \mathit{fl}(y), \ldots)$

Also see: Floating Point Arithmetic: Issues and Limitations.

Example for 2-digit decimal floating-point arithmetic:

- $\mathit{fl}(1.54 + 0.34) = \mathit{fl}(1.88) = 1.9$
- $\mathit{fl}(1.54) \mathbin{\widehat{+}} \mathit{fl}(0.34) = 1.5 \mathbin{\widehat{+}} 0.34 = 1.8$

Floating-point advice

- Don't use `float` type to solve *integer* problems.
  - Example: Is `a` divisible by `b`?
  - **BAD**: `a / b == round(a / b)`
  - **GOOD**: `a % b == 0`

  - Example: Find *integer* centered between `a` and `b`
  - **BAD**: `round((a + b) / 2)`
  - **GOOD**: `(a + b) // 2`

- Example: Are fractions $\frac{a}{b}$ and $\frac{c}{d}$ equal?
- **BAD**: `a / b == c / d`
- **GOOD**: `a * d == b * c`

- Don't use `float` type for finance (fixed-point decimal problems)
  Instead, use `Decimal`
- Don't compare `float` values for equality.
  Exception (in some situations): `a == 0.0`
  Instead, check *absolute* or *relative* difference.
  - **BAD**: `a == b`
  - **GOOD**: `abs(b - a) < 1e-6` (absolute difference)
  - **GOOD** `abs(b - a) / abs(a) < 1e-3` (relative diff.)
- Beware of **rounding** errors and **cancelation**.
- Prefer `math.fsum(...)` over `sum(...)` (see next example).

In [30]:
```
floats = [1.0, 1e20, 1.0, -1e20] * 1000
```

`floats` is list with $4\,000$ numbers

- What is the *exact* sum?
- What do you think `sum(floats)` returns?

In [31]:
```
sum(floats)
```

Out[31]:
```
0.0
```

Better do *compensated summation*:

In [32]:
```
math.fsum(floats)
```

Out[32]:
```
2000.0
```

Difficulty of Computational Problems

Some computational problems are harder than others

- Very easy:
  - Locate item in sorted list (*logarithmic*)
- Easy:
  - Find maximum in list (*linear*)
- Fairly easy:
  - Sort a list (better than *quadratic*: *linearithmic*)
  - List of length N can be sorted in roughly $N \log_2 N$ comparisons
- Hard:
  - Finding a shortest tour visiting all cities in The Netherlands

exponential

- Impossible:
    - Decide whether a loop terminates
    - Input: Python code of the loop, and initial values of variables
    - Output: Yes/No, whether loop terminates

In [33]:
```python
def collatz(n: int) -> int:
    """Determine number of Collatz steps to reach 1.

    Assumption: n > 0
    """
    result = 0

    while n != 1:
        if n % 2 == 0:  # n is even
            n //= 2
        else:
            n = 3 * n + 1
        result += 1

    return result
```

In [34]:
```python
[collatz(n) for n in range(1, 20+1)]
```

Out[34]:
```
[0, 1, 7, 2, 5, 8, 16, 3, 19, 6, 14, 9, 9, 17, 17, 4, 12, 20, 20, 7]
```

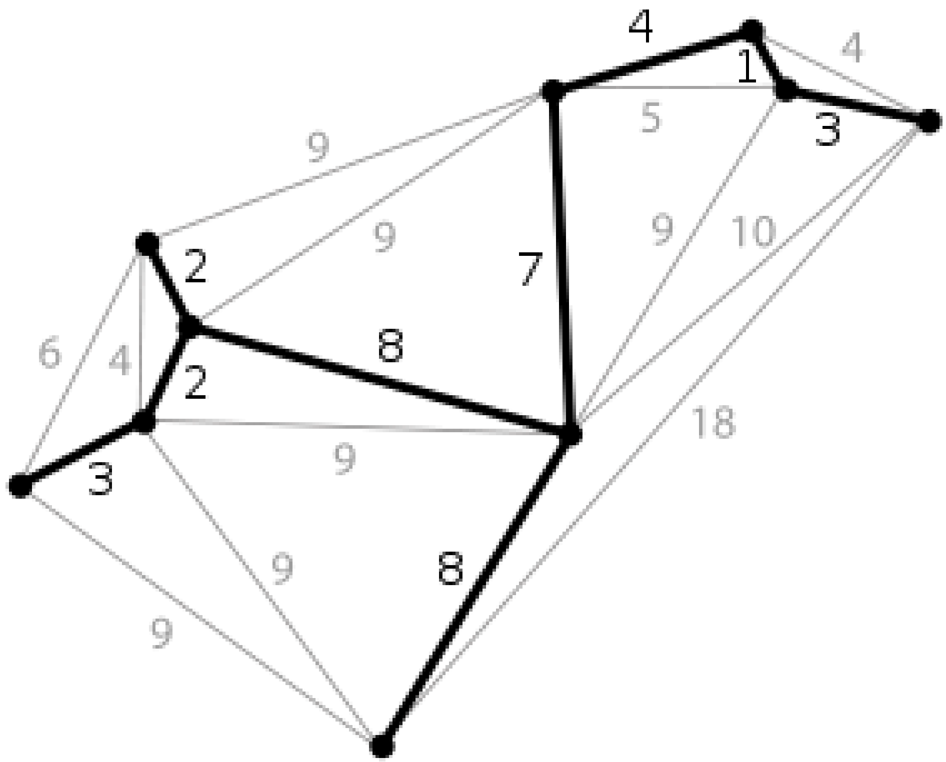
In [35]:
```python
max(((n, collatz(n)) for n in range(1, 1000+1)), key=lambda t: t[1])
```

Out[35]:
```
(871, 178)
```

It is unknown whether `while`-loop in `collatz(n)` terminates for all `n`

Graph Problems and Graph Algorithms

- Eulerian Path Problem
    - Does there exist a path that visits *each edge once*?
- Hamiltonian Path Problem
    - Does there exist a path that visits *each node*



once?

- [Shortest Path Problem]
  - Find shortest path from source node to target node
- [Minimum Spanning Tree]
  - Find subset of edges with mimimum weight that *connects all nodes*
  - This is always a 'tree'-like network
- [Traveling Salesman Problem]
  - Find path with minimum weight that visits *each node once*

- Easy:
  - Eulerian Path Problem
  - Shortest Path Problem
  - Minimum Spanning Tree
- Hard:
  - Hamiltonian Path Problem
  - Traveling Salesman Problem

Efficient algorithms for hard problems

- *Approximation* algorithms: sacrifice *accuracy*
- *Randomized* algorithms: sacrifice *reliability*
- *Heuristic* algorithms: sacrifice *provability*

Consult experts

Bonus: Uses of Python

- Raspberry Pi
- Mobile
  - iOS: Pythonista
  - Android: QPython
- Rhino3D: 3D design, modeling, etc.
  - scriptable in Python
- Blender: open-source animation engine
  - scriptable in Python
  - Blender demo
  - CHARGE - Blender Open Movie

(End of Notebook)