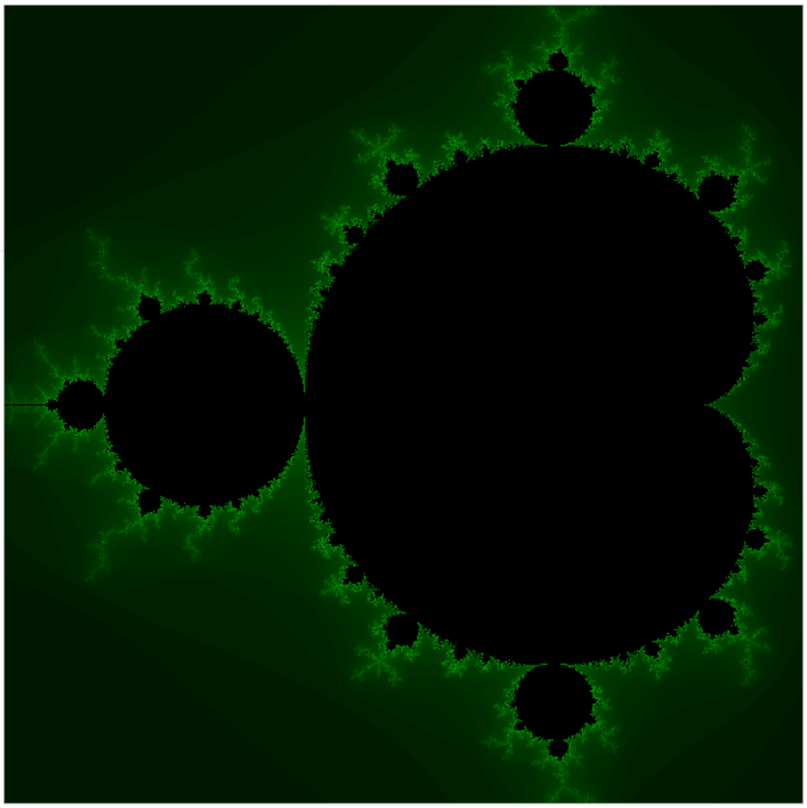
2IS50 – Software Development for Engineers – 2022-2023

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

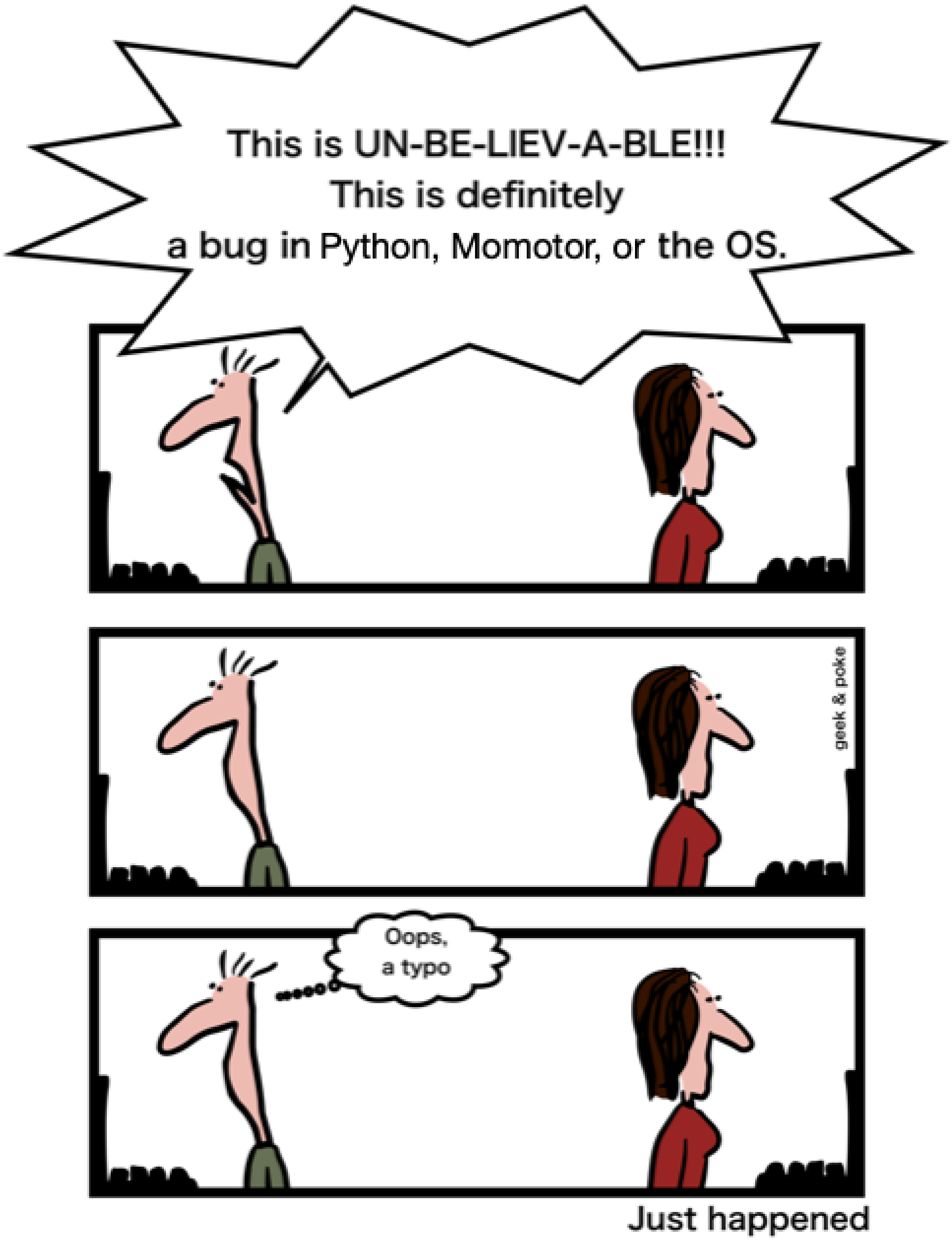
- Programming = The art and discipline of developing computer software
  - Developing = designing and implementing (writing code)

- 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?


This is UN-BE-LIEV-A-BLE!!!
This is definitely
a bug in Python, Momotor, or the OS.
geek & poke
Oops,
a typo
Just happened

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 $12 \times 12$.

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

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

```
ooo
oo
o
```

In [10]:
```python
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]:
```python
print_triangle(4)
```

```
oooo
ooo
oo
o
```

In [12]:
```python
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]:
```python
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]:
```python
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]:
```python
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:
        print(" {:3}".format(a * b), end='')
        b = b + 1

    print()
```

In [23]:
```
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):

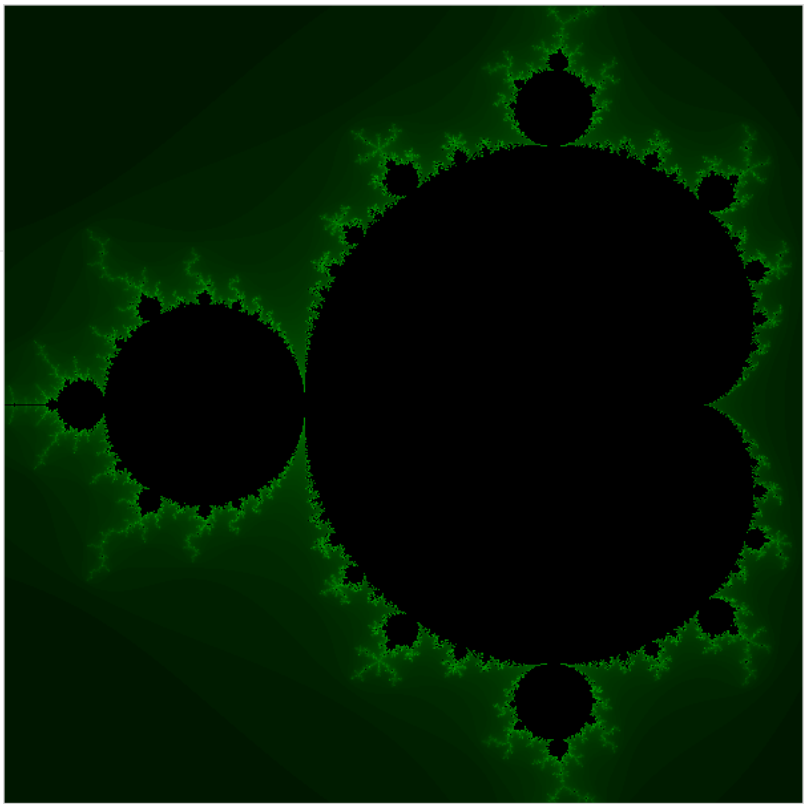
In [24]:
```
def print_line_4(n: int, s: str = "o") -> None:
    """Print a row of n times string s.
    """
    print(n * s)
```

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

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

(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]:
```
<map at 0x7faa7841b970>
```

In [21]:
```python
list(map(len, subjects))
```
Out[21]:
```
[11, 11, 11, 8]
```

Anonymous functions: `lambda` expressions

Syntax of `lambda` **expression**:

```
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]: list(filter(lambda s: len(s) == 11, subjects))
```

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

```
In [23]: 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]: 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

```
[ 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]:
```
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

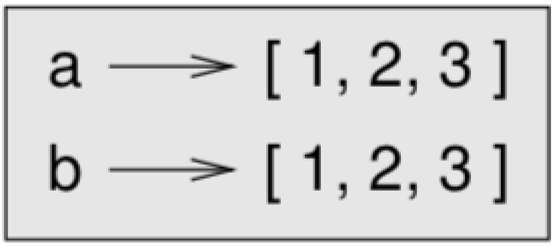
- **Flexibility**: lists are mutable, tuples are immutable
- **Performance**: lists have more overhead than tuples
  - Lists take up more memory space
  - List operations are slower

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.



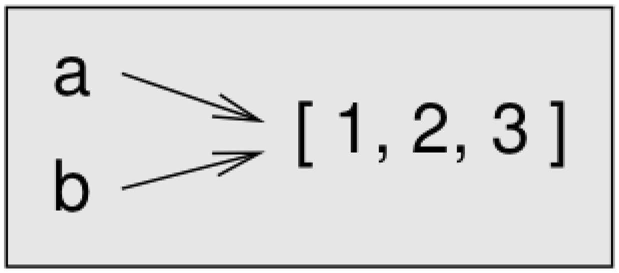
In [29]:
```
a = [1, 2, 3]
b = [1, 2, 3]

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

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

In [30]:
```
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]:
```python
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]:
```python
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]:
```python
# 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]:
```python
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*.

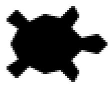
Also see: The Beginner's Guide to Python Turtle on *Real Python*

(A first encounter with recursive functions)

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

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

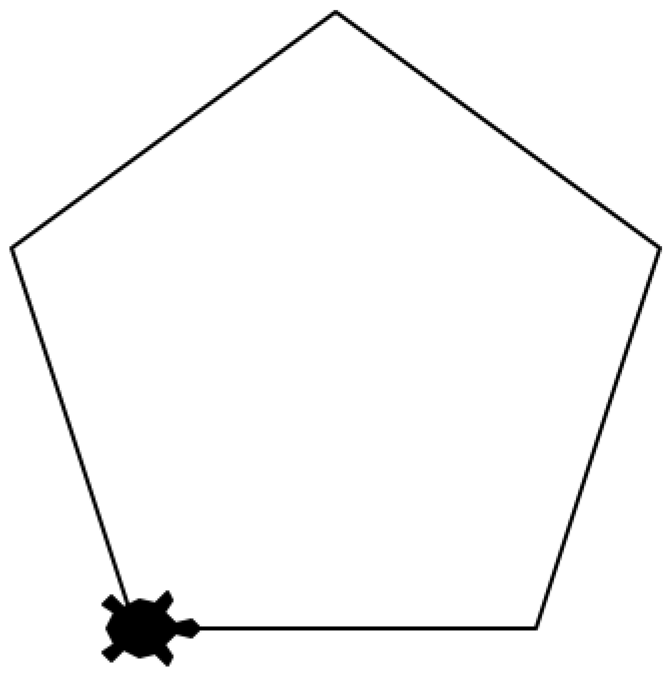
A new window should have opened showing the following:



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
```

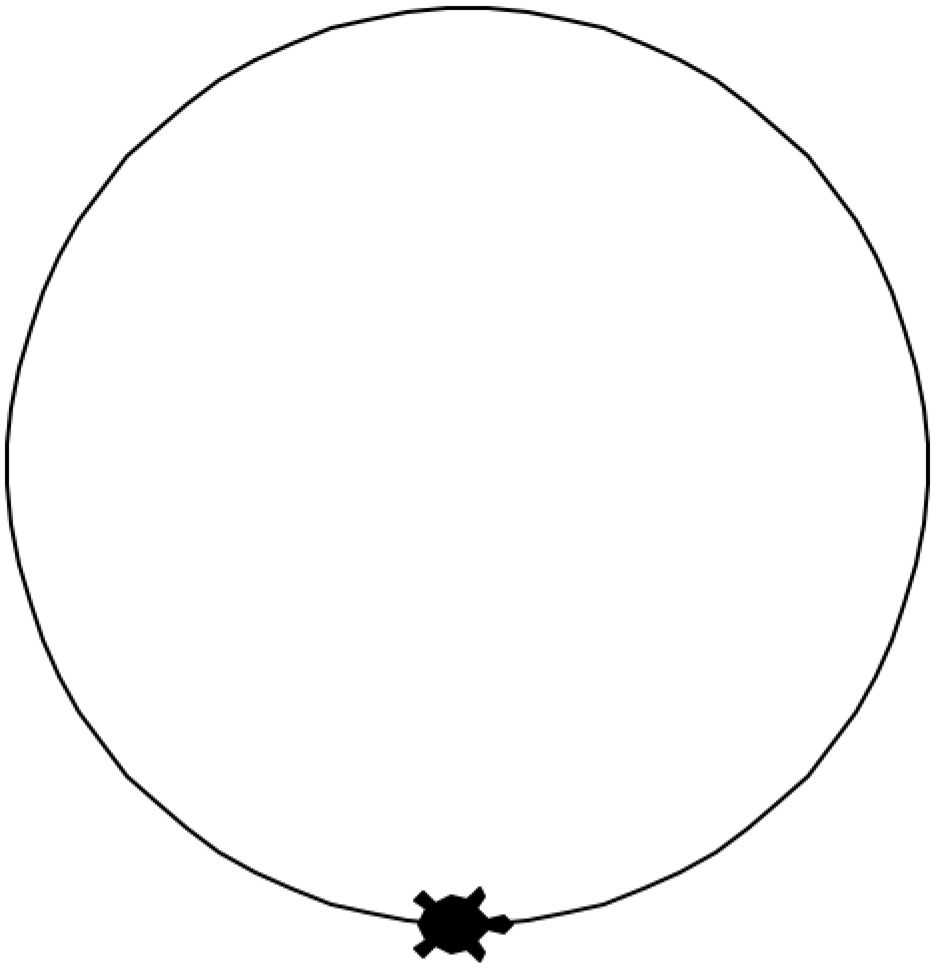
In [58]:
```
n_gon(5)
```

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):

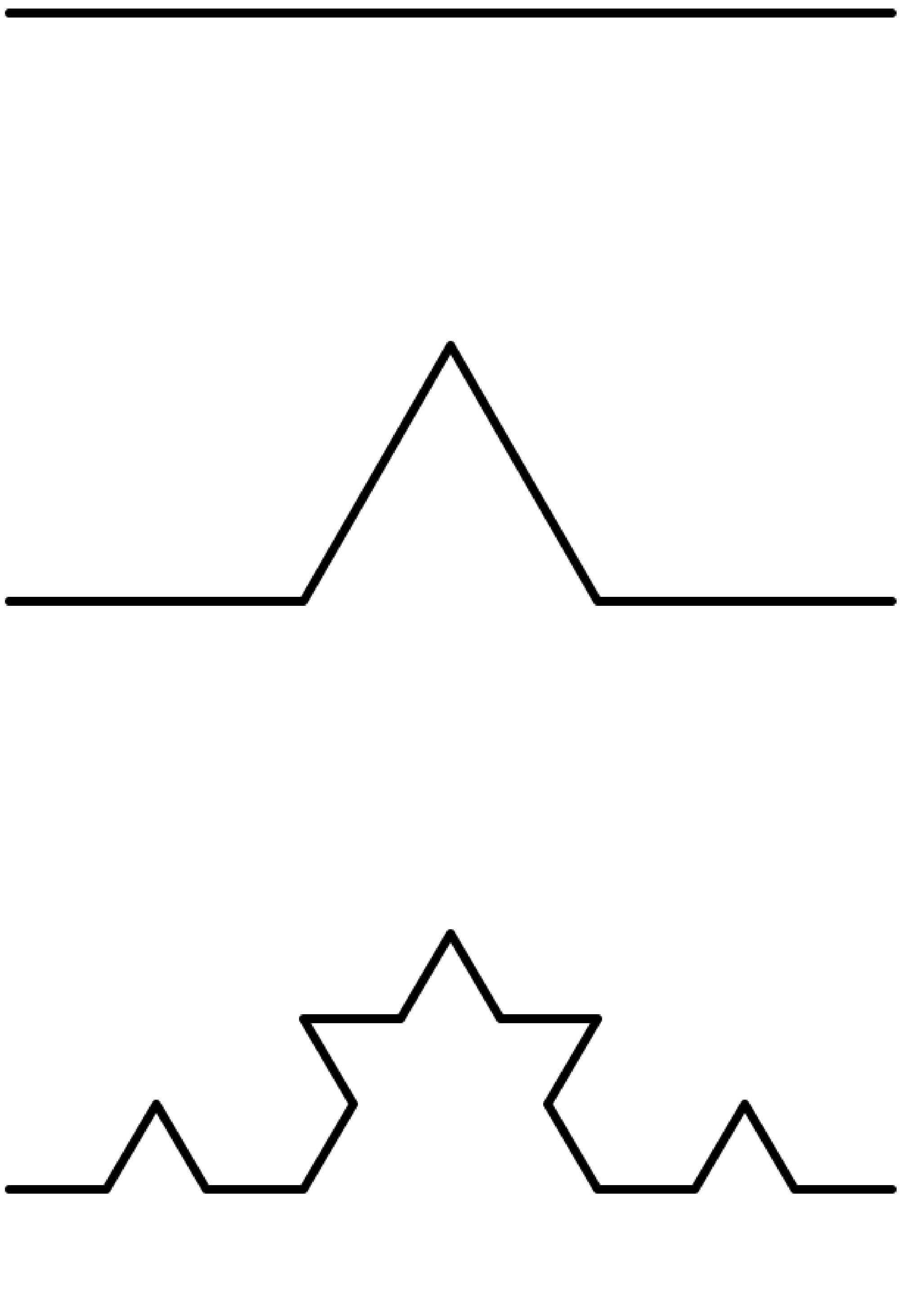


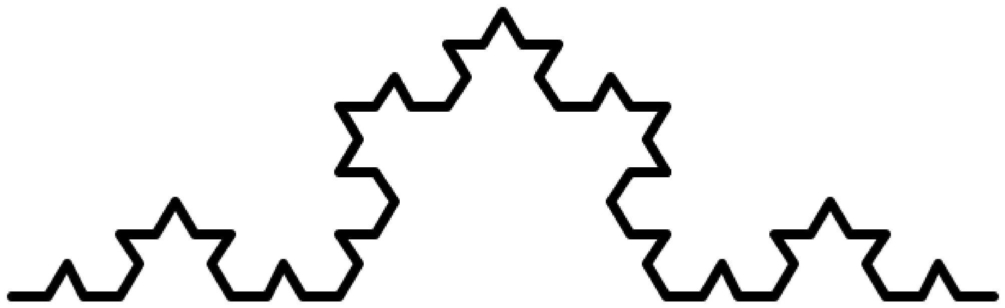
In [60]:
```python
def koch(t: turtle.Turtle, n: int, size: float = 100) -> None:
    """Draw a Koch fractal with n generations of given size.
    """
    t.pendown()
    if n == 0:
        t.forward(size)
    else:
        koch(t, n - 1, size / 3)
        t.left(60)
        koch(t, n - 1, size / 3)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

2is50-2223-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).

[ ]:
```
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

[ ]:
```
from sys import getsizeof

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

[ ]:
```
%%timeit -c

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

[ ]:
```
%%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)

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

[ ]:
```
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)

[ ]:
```
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**

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

[ ]:
```
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)

[ ]:
```
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.

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

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

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

[ ]:
```
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*

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

[ ]:
```
# 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?

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

[ ]:
```
# 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 concludes 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()}")
```

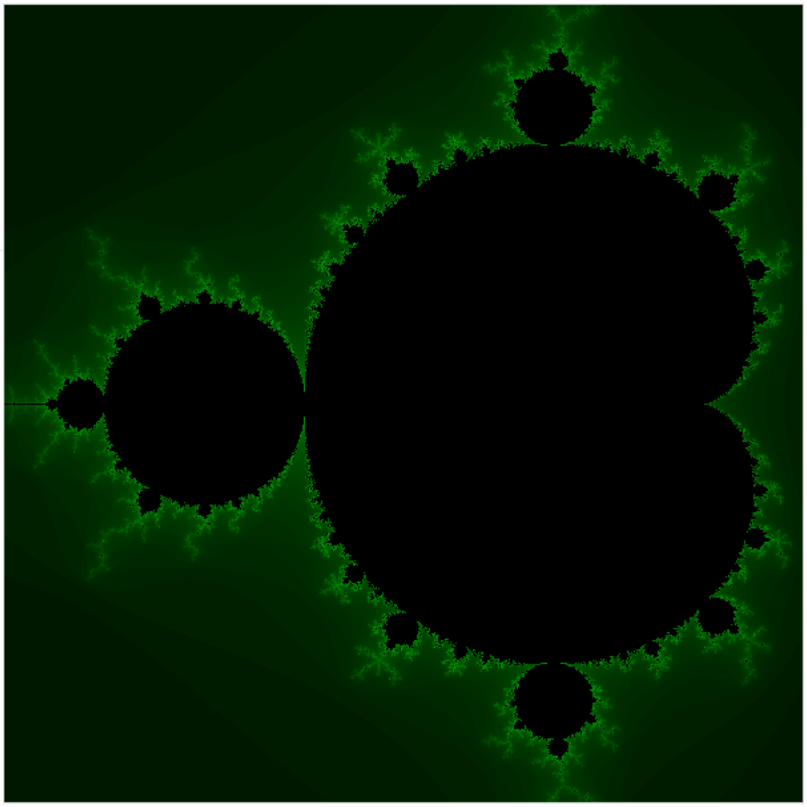
1.8 (End of Notebook)

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]:
```python
# Preliminaries

from collections import defaultdict, Counter
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]:
```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
    """
```

In [3]:
```python
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]:
```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 s.index(max(s))
```

In [5]:
```python
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]:
```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
    """
    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:
```

```
    """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]:
```
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]:
```
arg_max([13, 42, 17, 42])
```

Out[15]:
```
1
```

Simplify this approach:

In [16]:
```
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]:
```
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]:
```python
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]:
```python
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]:
```python
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]:
```python
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]:
```python
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
```

```
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]: # 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]: # 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]: 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]: 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]:
```python
bag: Counter = Counter('Mississippi')
```

In [41]:
```python
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]:
```
bag.most_common
```

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

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

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

In [44]:
```
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]:
```
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
```

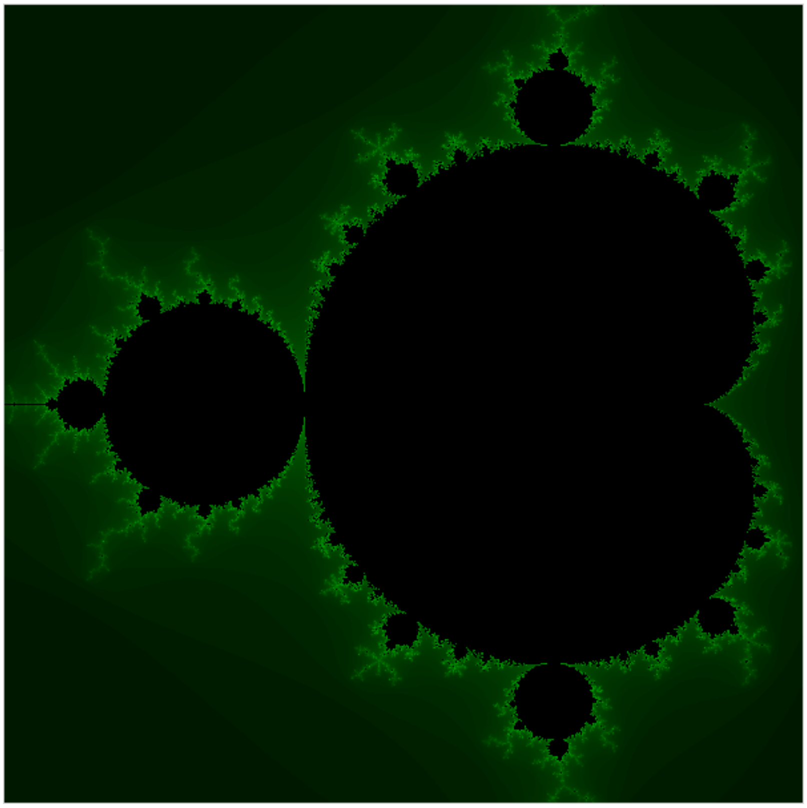
```
Finding tests in QuadPoly
Trying:
    q = QuadPoly(1, -8, 12)
Expecting nothing
ok
Trying:
    q
Expecting:
    QuadPoly(1, -8, 12)
ok
Trying:
    print(q)
Expecting:
    1 x^2 + -8 x + 12
ok
Trying:
    q.eval(0)
Expecting:
    12
ok
Trying:
    q.eval(2)
Expecting:
    0
ok
Trying:
    sorted(q.solve())
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 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]:
```
try:
    print(1 / 0)
    print("Further work")
except:
    print("Something went wrong")
```

```
Something went wrong
```

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

```
+inf
```

In [6]:
```
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]:
```
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:

```
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]:
```
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]:
```python
itr = iter('abc')
iter(itr) is itr
```

Out[21]:
```
True
```

In [22]:
```python
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]:
```python
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]:
```
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
```

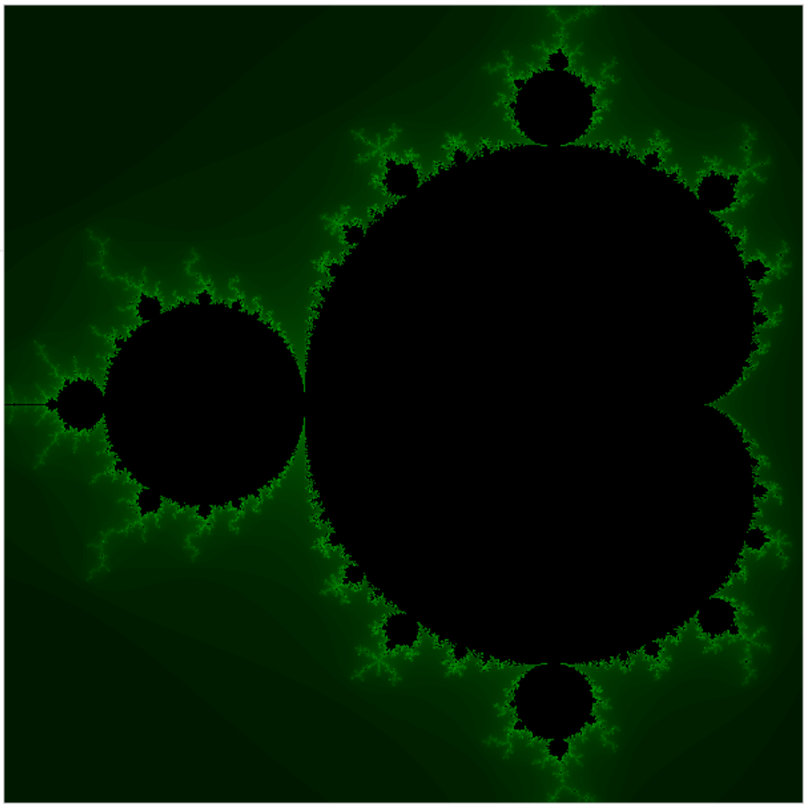
(End of Notebook)

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

Processing math: 100%

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)
```

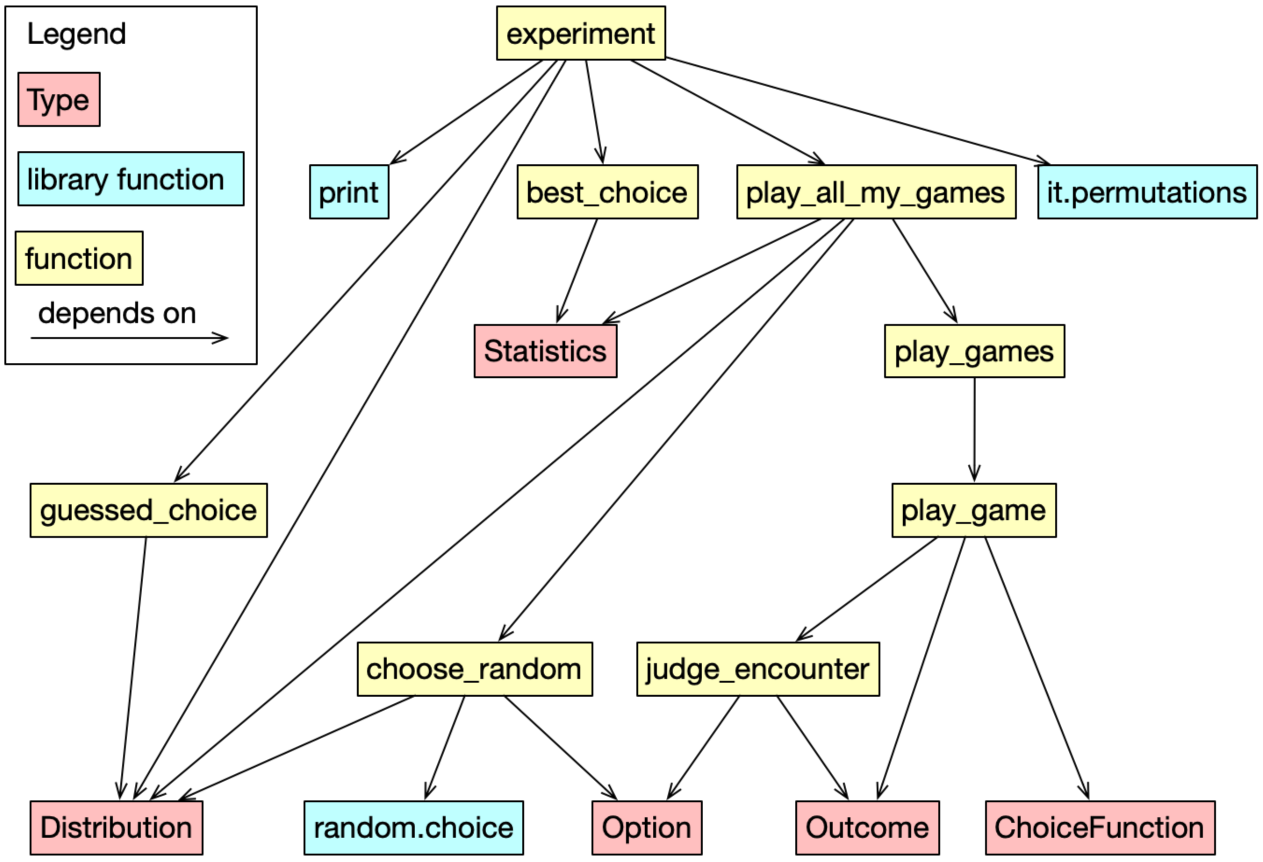
```python
 -> int:
    return sum(play_game(choice_1, choice_2) % 2 for _ in range(n))

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='M
arkovChainChoice')
```

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]:
```python
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
```

```
    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]:
```
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]:
```
# 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]:
```python
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]:
```python
doctest.run_docstring_examples(BeatPreviousPlayer, globs=globals(), name='
BeatPreviousPlayer')
```

In [30]:
```python
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]:
```
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",
                              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]:
```
doctest.run_docstring_examples(OutcomeStats, globs=globals(), name='Outcom
Stats')
```

Referee

In [36]:
```
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}")
        return stats
```

In [37]:
```
referee = Referee(markov, beat_previous)

referee.play_games(1000, True)
```

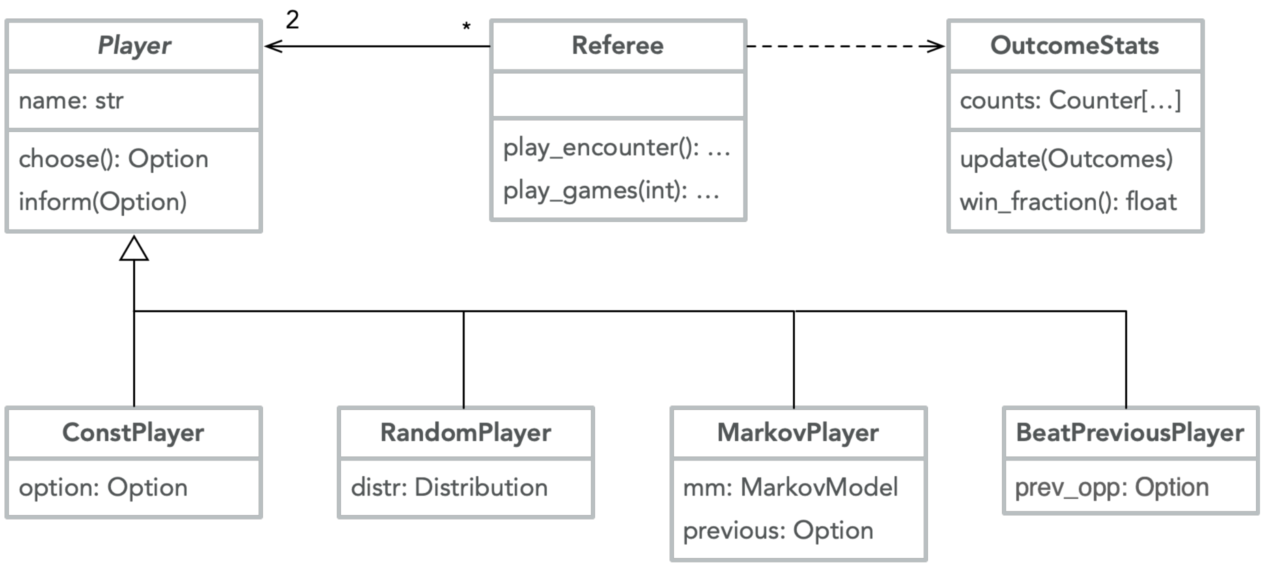
```
MarkovPlayer('Mark Ov') vs BeatPreviousPlayer('Beat Prev')
477 wins - 523 ties - 0 losses
win fraction: 1.00
```

Out[37]:
```
OutcomeStats(Counter({0: 523, 1: 477}))
```

In [38]:
```
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]:
```
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]:
```
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='bina
ry_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]:

```python
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]:
```
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]:
```
doctest.run_docstring_examples(binary_numbers_2, globs=globals(), name='bi
nary_numbers_2')
```

In [62]:
```
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]:
```
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
```

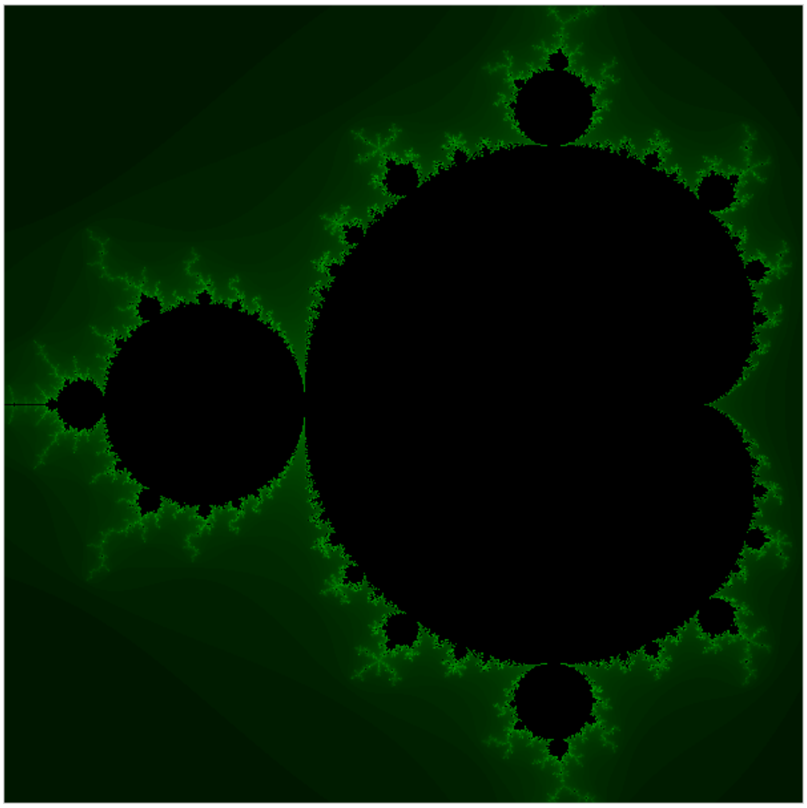
```
0  0  1  0
0  0  1  1
0  1  0  0
0  1  0  1
0  1  1  0
0  1  1  1
1  0  0  0
1  0  0  1
1  0  1  0
1  0  1  1
1  1  0  0
1  1  0  1
1  1  1  0
1  1  1  1
```

(End of Notebook)

2IS50 – Software Development for Engineers – 2022-2023

Lecture 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
        for c in coefficients:
            result = result * x + c

        return result

    return f
```

In [22]:
```
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]:
```
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]:
```
coefficients = 1.0, -1e6, 1.0
quadratic = poly(*coefficients)

x = a_hat(*coefficients)
```

```
x
```

```
Out[24]: 1.000007614493 37e-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]:
```
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]:
```
[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]
```

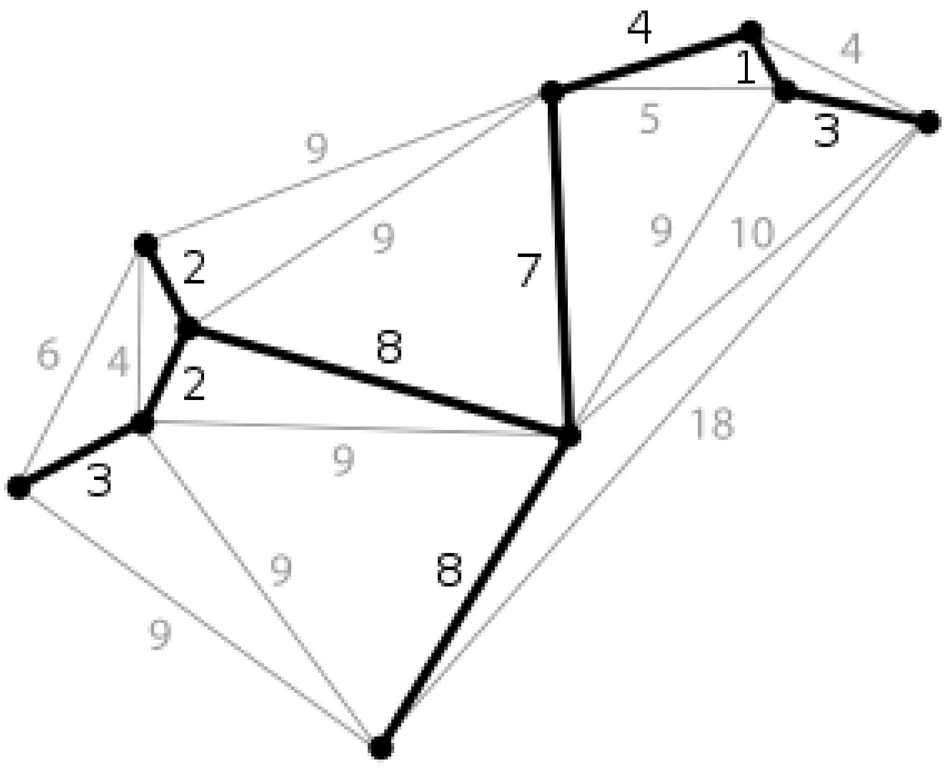
In [35]:
```
max(((n, collatz(n)) for n in range(1, 1000+1)), key=lambda t: t[1])
```

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)