Chapter 5: Basic algorithms

Initialize a list of random numbers to work with in the following exercises. Generate 10 numbers between 1 and 100:

In [1]:

```
import random

random.seed(42) # to reproduce the same results
data = []
for i in range(10):
    data.append(random.randint(1, 100))
print(data)
```

```
[82, 15, 4, 95, 36, 32, 29, 18, 95, 14]
```

Same with using Python's list generator expressions:

In [2]:

```
random.seed(42) # to reproduce the same results
data = [random.randint(1, 100) for i in range(10)]
print(data)
```

```
[82, 15, 4, 95, 36, 32, 29, 18, 95, 14]
```

Help: given a list of numbers in variable `numbers` , produce the `halves` list, which contains each number divided by 2:

In [3]:

```
numbers = [10, 20, 30, 40, 50]
print(numbers)

# iteration
halves = []
for x in numbers:
    halves.append(x / 2)
print(halves)

# list generation
halves = [x / 2 for x in numbers]
print(halves)
```

```
[10, 20, 30, 40, 50]
[5.0, 10.0, 15.0, 20.0, 25.0]
[5.0, 10.0, 15.0, 20.0, 25.0]
```

Summation

Let $f : [m..n] \to H$ be a function. Let the addition operator $+$ be defined over the elements of H, which is an associative operation with a left identity element 0. Our task is to summarize the values of f function over the interval. Formally:

$$s = \sum_{i=m}^{n} f(i)$$

Theoretical way

In [4]:

```
result = 0
for i in range(0, len(data)):
    result += data[i]
print("Sum: {0}".format(result))
```

```
Sum: 420
```

Pythonic way

In [5]:

```
result = 0
for value in data:
    result += value
print("Sum: {0}".format(result))
```

```
Sum: 420
```

Built-in function

In [6]:

```
result = sum(data)
print("Sum: {0}".format(result))
```

```
Sum: 420
```

Counting

Given the $[m..n]$ interval, count the number of items inside it. Formally:

$$s = \sum_{i=m}^{n} 1$$

Theoretical way

In [7]:

```
result = 0
for i in range(0, len(data)):
    result += 1
print("Count: {0}".format(result))
```

```
Count: 10
```

Pythonic way

In [8]:

```
result = sum([1 for _ in data])
print("Count: {0}".format(result))
```

```
Count: 10
```

Remark: a single underscore (_) is a valid variable name. We usually name a variable like this to emphasize that this variable will not be used later.

Built-in function

In [9]:

```
result = len(data)
print("Count: {0}".format(result))
```

```
Count: 10
```

Maximum search

Let $f : [m..n] \rightarrow H$ be a function, $m \leq n$. Over the elements of H let a total ordering relation be defined (reflexivity, antisymmetry, transitivity and connexity), with a symbol $\leq$, for the strict version $<$. Our task is to determine the greatest value in the interval. Also determine an element of the interval, where function f evaluates to this greatest value. Formally:

$$max = f(ind) \wedge \forall i \in [m..n] : f(i) \leq f(ind)$$

Theoretical way

In [10]:

```
result = data[0]
index = 0
for i in range(1, len(data)): # we don't need to compare the 0th element
    if data[i] > result:
        result = data[i]
        index = i
print("Max: {0}, Index: {1}".format(result, index))
```

```
Max: 95, Index: 3
```

Pythonic way

In [11]:

```
result = data[0]
index = 0
for idx, value in enumerate(data):
    if value > result:
        result = value
        index = idx
print("Max: {0}, Index: {1}".format(result, index))
```

```
Max: 95, Index: 3
```

Little optimization to skip the 0^{th} element:

In [12]:

```
result = data[0]
index = 0
for idx, value in enumerate(data[1:], start = 1):
    if value > result:
        result = value
        index = idx
print("Max: {0}, Index: {1}".format(result, index))
```

```
Max: 95, Index: 3
```

Built-in function

In [13]:

```
result = max(data)
print("Max: {0}".format(result))
```

```
Max: 95
```

If the index of the element is also needed:

In [14]:

```
result = max(data)
index = data.index(result)
print("Max: {0}, Index: {1}".format(result, index))
```

```
Max: 95, Index: 3
```

Note: this will iterate over the list twice, hence the computational cost is also doubled.

Linear search

Let $\beta : [m..n] \rightarrow \mathbb{L}$ condition be defined. Determine the first element of the interval which fulfills the condition (if any). Formally:

$$l = (\exists i \in [m..n] : \beta(i))$$
$$l \rightarrow (ind \in [m..n] \wedge \beta(ind) \wedge \forall i \in [m..ind-1] : \neg\beta(i))$$

Beta condition

Introduce an `is_odd` function which determines whether a number is odd or not:

In [15]:

```
def is_odd(number):
    return number % 2 != 0
```

Theoretical way

In [16]:

```
result = 0
index = 0
found = False
i = 0
while not found and i < len(data):
    if is_odd(data[i]):
        result = data[i]
        index = i
        found = True
    i += 1
if found:
    print("Linear search: {0}, Index: {1}".format(result, index))
else:
    print("Linear search did not found an appropriate item")
```

```
Linear search: 15, Index: 1
```

Pythonic way

In [17]:

```
result = [x for x in data if is_odd(x)]
if len(result) > 0:
    print("Linear search: {0}".format(result))
else:
    print("Linear search did not found an appropriate item")
```

```
Linear search: [15, 95, 29, 95]
```

Built-in function

In [18]:

```
result = filter(is_odd, data)
print("Linear search: {0}".format(list(result)))
```

```
Linear search: [15, 95, 29, 95]
```

Here `result` is a special filter object which can be either converted to a list to get all results (as above) or dynamically evaluated and step to the next result with the `next()` function:

In [19]:

```
result = filter(is_odd, data)
print("Linear search: {0}".format(next(result, None))) # None is the default val
ue to use if no number was odd.
```

```
Linear search: 15
```

Conditional summation

The algorithm of summation can be further generalized when a $\beta : [m\,..\,n] \to \mathbb{L}$ condition is defined to restrict the set of elements.

Let $f : [m\,..\,n] \to H$ be a function and $\beta : [m\,..\,n] \to \mathbb{L}$ a condition. Let the addition operator $+$ be defined over the elements of H, which is an associative operation with a left identity element 0. Our task is to summarize the values of f function over the interval where the β condition is fulfilled. Formally:

$$s = \sum_{\substack{i=m \\ \beta(i)}}^{n} f(i)$$

Theoretical way

In [20]:

```
result = 0
for i in range(0, len(data)):
    if is_odd(data[i]):
        result += data[i]
print("Sum: {0}".format(result))
```

```
Sum: 234
```

Pythonic way

In [21]:

```
result = 0
for value in data:
    if is_odd(value):
        result += value
print("Sum: {0}".format(result))
```

```
Sum: 234
```

Built-in function

In [22]:

```
result = sum(filter(is_odd, data))
print("Sum: {0}".format(result))
```

```
Sum: 234
```

Conditional counting

Let $\beta : [m..n] \to \mathbb{L}$ be a condition. Count how many items of the interval fulfills the condition! Formally:

$$s = \sum_{\substack{i=m \\ \beta(i)}}^{n} 1$$

Theoretical way

In [23]:

```
result = 0
for i in range(0, len(data)):
    if is_odd(data[i]):
        result += 1
print("Count: {0}".format(result))
```

```
Count: 4
```

Pythonic way

In [24]:

```
result = sum([1 for x in data if is_odd(x)])
print("Count: {0}".format(result))
```

```
Count: 4
```

Built-in function

In [25]:

```
result = len(list(filter(is_odd, data)))
print("Count: {0}".format(result))
```

```
Count: 4
```

Conditional maximum search

The algorithm of maximum search can be further generalized with combining the $\beta : [m..n] \to \mathbb{L}$ condition used in *linear search* as a restriction. Note that now the existence of a maximum value is not guaranteed.

Let $f : [m..n] \to H$ be a function and $\beta : [m..n] \to \mathbb{L}$ a condition. Over the elements of H let a total ordering relation be defined (reflexivity, antisymmetry, transitivity and connexity), with a symbol $\leq$, for the strict version $<$. Our task is to determine the greatest value in the interval which fulfills the β condition. Also determine an element of the interval, where function f evaluates to this greatest value. Formally:

$$l = (\exists i \in [m..n] : \beta(i))$$
$$l \to (\beta(ind) \land max = f(ind) \land \forall i \in [m..n] : \beta(i) \to f(i) \leq f(ind))$$

Theoretical way

In [26]:

```
found = False
result = 0
index = 0
for i in range(0, len(data)):
    if is_odd(data[i]) and (not found or data[i] > result):
        found = True
        result = data[i]
        index = i
print("Max: {0}, Index: {1}".format(result, index))
```

```
Max: 95, Index: 3
```

The `found` variable can be omitted if initialize the `result` variable with the special `None` value and compare to that:

In [27]:

```
result = None
index = -1
for i in range(0, len(data)):
    if is_odd(data[i]) and (result == None or data[i] > result):
        result = data[i]
        index = i
print("Max: {0}, Index: {1}".format(result, index))
```

```
Max: 95, Index: 3
```

Pythonic way

In [28]:

```
result = None
index = -1
for idx, value in enumerate(data):
    if is_odd(value) and (result == None or value > result):
        result = value
        index = idx
print("Max: {0}, Index: {1}".format(result, index))
```

```
Max: 95, Index: 3
```

Built-in function

In [29]:

```
result = max(filter(is_odd, data))
print("Max: {0}".format(result))
```

```
Max: 95
```

Exercise

Task: the name, area and population data for the neighbouring countries are given in the `countries`, `areas` and `popultions` lists below. Calculate the population density for each neighbouring country and display it. Determine which country has the highest population density.

In [30]:

```
countries = ['Austria', 'Slovakia', 'Ukraine', 'Romania', 'Serbia', 'Croatia',
'Slovenia']
areas = [83871, 49037, 603500, 238397, 88361, 56594, 20273]
populations = [8877036, 5450017, 42010063, 19405156, 6963764, 4130304, 2084301]
```

In [31]:

```
densities = []
for i in range(len(countries)):
    densities.append(populations[i] / areas[i])
    print('{0}: {1:.2f} persons/km2'.format(countries[i], densities[i]))
```

```
Austria: 105.84 persons/km2
Slovakia: 111.14 persons/km2
Ukraine: 69.61 persons/km2
Romania: 81.40 persons/km2
Serbia: 78.81 persons/km2
Croatia: 72.98 persons/km2
Slovenia: 102.81 persons/km2
```

In [32]:

```
result = max(densities)
index = densities.index(result)
print("Max: {0}, Index: {1}, Country: {2}".format(result, index, countries[index
]))
```

```
Max: 111.14091400371149, Index: 1, Country: Slovakia
```

Logarithmic search (optional)

Also called binary search.

Let $f : [m..n] \to H$ be a *monotonically increasing* function. Over the elements of H let a total ordering relation be defined (reflexivity, antisymmetry, transitivity and connexity), with a symbol $\leq$, for the strict version $<$. Determine whether function f evaluates to a given value $h \in H$ at any location. If yes, specify such a location. Formally:

$$l = (\exists i \in [m..n] : f(i) = h) \land l \to f(ind) = h$$

In [33]:

```
def log_search(elements, value):
    first = 0
    last = len(elements) - 1
    while first <= last:
        i = (first + last) // 2
        if elements[i] == value:
            return i
        elif elements[i] < value:
            first = i + 1
        else:
            last = i - 1
    return -1

data_sorted = sorted(data)
print("Sorted data: {0}".format(data_sorted))

index = log_search(data_sorted, data[0])
print("Logarithmic search: value={0}, index={1}".format(data[0], index))
```

```
Sorted data: [4, 14, 15, 18, 29, 32, 36, 82, 95, 95]
Logarithmic search: value=82, index=7
```