Introduction to Programming Lecture 4

- Instructor : Andrew Yarmola andrew.yarmola@uni.lu
- Course Schedule : Wednesday 14h00 - 15h30 Campus Kirchberg B21
- Course Website : sites.google.com/site/andrewyarmola/itp-uni-lux
- Office Hours : Thursday 16h00 - 17h00 Campus Kirchberg G103 and by appointment.

Remarks on homework and questions

Box problem solution

In [1]:

```python
def individual_strategy(boxes_list, num_tries, num_to_find) :
    """ Implements the cycle strategy for a player to
    look for their number (num_to_find) in a list (boxes_list) with
    a limited number of tries (num_tries). The function
    returns True if the number is found, False otherwise.
    The algorithm treats boxes[i] as a permtuation and
    looks through the cycle containing i = person. """
    count = 0
    idx = num_to_find
    while count < num_tries :
        curr = boxes_list[idx]
        if curr == num_to_find :
            return True
        idx = curr
        count += 1
    return False
```

Permutations

The permutations exercise seems to be the most challenging. Depending on how all the homeworks look, I might grade most of the exercise only for extra points. We will see.

For cycle notation we are using a tuple of tuples. Here are some examples

In [2]:

```python
a = ( (1,2,4) , (0,3) )
b = ((),)
c = ( (0,3) , (9,), (1,2,4) )
d = ( (1,2), (6,), (), (0,3) )
```

In [3]:

```python
def max_or_minusone(t) :
    """ Returns the max of a tuple of ints,
    or -1 if it is empty. """
    if len(t) > 0 :
        return max(t)
    else :
        return -1

def min_perm_size(p) :
    """ Returns the minimum permutation size given a tuple
    of tuples representing a disjoint cycle decomposition.
    Raises a SyntaxError on bad input. """
    if not valid_disjoint_cycle_rep(p) :
        raise SyntaxError("Bad disjoint cycle representation")
    else :
        return max(map(max_or_minusone,p)) + 1
```

Instead of using `map`, you could have just used a `for` loop to find the maximum integer in each tuple.

In [4]:

```python
def min_perm_size_v2(p) :
    """ Returns the minimum permutation size given a tuple
    of tuples representing a disjoint cycle decomposition.
    Raises a SyntaxError on bad input. """
    if not valid_disjoint_cycle_rep(p) :
        raise SyntaxError("Bad disjoint cycle representation")
    else :
        perm_max = -1
        for cycle in p :
            if len(cycle) > 0 :
                cycle_max = max(cycle)
                if cycle_max > perm_max :
                    perm_max = cycle_max
        return perm_max + 1
```

Let's look at the `valid_disjoint_cycle_rep` function.

In [5]:

```
def valid_disjoint_cycle_rep(p) :
    """ Returns True if the input is a tuple of tuples representing
    a disjoint cycle decomposition of a permutation. """
    # check type and length
    if type(p) is not tuple or len(p) == 0 : return False
    # we wrote our own max_or_minusone function to deal with empty tuples
    perm_size = max(map(max_or_minusone,p)) + 1
    if perm_size < 0 : return False
    # we will use a seen list to keep track of seen elements
    # note : a set or dict is better for this
    seen = [0]*perm_size
    for cycle in p :
        if type(cycle) is not tuple : return False
        for i in cycle :
            if type(i) is not int or i < 0 :
                return False
            elif seen[i] == 1 :
                return False
            else :
                seen[i] = 1
    return True
```

We will use sets to make a slightly cleaner version of the above solution in the lecture. Let's try our code on a few examples.

In [10]:

```
valid_disjoint_cycle_rep( ( (2,1) , (3,0,4) ) )
```

Out[10]:

```
True
```

In [11]:

```
valid_disjoint_cycle_rep( ( (1,2) , (3,0,4) , (5,2,2) ) )
```

Out[11]:

```
False
```

In [12]:

```
a = ( (1,2,4) , (0,3) )
b = ((),)
c = ( (0,3) , (9,), (4,1,2) )
d = ( (1,2), (6,), (), (0,3) )

print( min_perm_size(a) )
print( min_perm_size_v2(b) )
print( min_perm_size(c) )
print( min_perm_size(d) )
```

```
5
0
10
7
```

In [13]:

```
min_perm_size( ((0,1),(2,1)) )
```

```
Traceback (most recent call last):

  File "/Users/yarmola/miniconda3/lib/python3.6/site-packages/IPython/core/interactiveshell.py", line 2963, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)

  File "<ipython-input-13-daf6bd9cd54a>", line 1, in <module>
    min_perm_size( ((0,1),(2,1)) )

  File "<ipython-input-3-ab62b314add3>", line 14, in min_perm_size
    raise SyntaxError("Bad disjoint cycle representation")

  File "<string>", line unknown
SyntaxError: Bad disjoint cycle representation
```

Now let us look at `func_from_disjoint_cycle`.

In [16]:

```
def func_from_disjoint_cycle(p) :
    """ Returns the function representation of a permutation
    given a tuple of tuples representing a disjoint cycle
    decomposition. Raises a SyntaxError on bad input. """
    # this will validate input, so I don't have to here
    perm_size = min_perm_size(p)
    # start with idenity and modify
    f = list(range(perm_size))
    for cycle in p :
        cycle_len = len(cycle)
        for i in range(cycle_len) :
            f[cycle[i]] = cycle[ (i+1) % cycle_len ]
    return tuple(f)
```

In [17]:

```
print("Cycle rep :", a)
print("Function rep :", func_from_disjoint_cycle(a))
```

```
Cycle rep : ((1, 2, 4), (0, 3))
Function rep : (3, 2, 4, 0, 1)
```

The most challenging was `cycle_from_func`. This requires us to find all the cycles in a function representation, similar to the box problem.

In [18]:

```
def is_perm(f) :
    """ Returns True is the tuple f represents a
    permutation function f[i] of {0,...,len(f)}. """
    if type(f) is not tuple : return False
    return sorted(f) == list(range(len(f)))

def cycle_from_func(f) :
    """ Returns the disjoint cycle representation of a permutation
    given a tuple as a function reprepresentation.
    Raises a SyntaxError on bad input. """
    if not is_perm(f) :
        raise SyntaxError("Bad function representation")
    # we will use a cycle_start and seen to keep track of the cycles
    cycle_rep = []
    n = len(f)
    seen = [0]*n
    while 0 in seen :
        # find the first non-visited int
        c_start = seen.index(0)
        cycle = [c_start]
        seen[c_start] = 1
        c_next = f[c_start]
        while c_next != c_start :
            cycle.append(c_next)
            seen[c_next] = 1
            c_next = f[c_next]
        if len(cycle) > 1 :
            cycle_rep.append(tuple(cycle))
    if len(cycle_rep) > 0 :
        return tuple(cycle_rep)
    else :
        return ((),)
```

In [19]:

```
f = (4, 5, 3, 2, 1, 0)
print("Function rep :", f)
print("Cycle rep :", cycle_from_func(f))
```

```
Function rep : (4, 5, 3, 2, 1, 0)
Cycle rep : ((0, 4, 1, 5), (2, 3))
```

More control flow

Assertions, exceptions and exception handling

In the homework, you were introduced to the `raise` exception keyword. Exceptions in python are tools to tell the interpreter that something "bad" has happened and that the code **should stop** executing. This could be because someone gave bad input, there was a problem writing or reading data, or a function returned unexpected output (note: you should check the output of poorly documented functions that you don't trust). The keywords here are

- `assert some_boolean_statement` will cause the code to stop and raise an AssertionError.
- `raise exception_type("some user message")` will cause the code to stop and raise an exception of type `exception_type` and a user message `"some user message"`. There are many many built-in exception types. Some useful ones are
  - `Exception()` this is the general type
  - `RuntimeError()` something bad happened as your code was running
  - `TypeError()` bad type given to function
  - `ValueError()` good type, but bad value
  - `SyntaxError()` bad syntax, couldn't parse

In [20]:

```
assert type(3) is int
print("Assertion passed!")
```

```
Assertion passed!
```

In [21]:

```
assert type(1.) is int
print("Assertion passed!")
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-21-c7cd3f9347b0> in <module>()
----> 1 assert type(1.) is int
      2 print("Assertion passed!")

AssertionError: 
```

In [22]:

```
raise Exception("Instead of returning, \
I can raise an exception on bad input")
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-22-4f511be2af8e> in <module>()
----> 1 raise Exception("Instead of returning, I can raise an exception on bad input")

Exception: Instead of returning, I can raise an exception on bad input
```

Catching exceptions : try, except, finally block

Now, let's say there is a very useful function you want to use, but it will `raise` an exception on you for bad input. For example, if you look for an element in a list using `list.index()`, it will `raise` an exception if that element is not found.

In [23]:

```
[0,1,2,3].index(5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-23-9646355bb2af> in <module>()
----> 1 [0,1,2,3].index(5)

ValueError: 5 is not in list
```

We can catch an exception by using the `try` block. It's best demonstrated with an example.

In [24]:

```
try :
    [0,1,2,3].index(5)
    print("I will only run in the lines above me don't fail")
except ValueError : # Run if ValueError is raised
    print("Got a ValueError")
finally : # Will always be run at the end
    print("I always run at the very end")
```

```
Got a ValueError
I always run at the very end
```

You don't have to use the `finally` block at all, if you don't need it. Also, if you want to catch all exceptions, you can just use `except` without the exception type.

In [25]:

```
def max_or_minusone(t) :
    """ Tries to returns max(t) ,
    or -1 if max(t) fails. """
    try :
        return max(t)
    except :
        return -1
```

In [26]:

```
max_or_minusone(())
```

Out[26]:

```
-1
```

Lastly, if you want to do nothing, just use the `pass` keyword in the exception block. The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. You can also use it as a place holder while working on code.

In [27]:

```
something = 'a string'
try :
    someting += 1
except : # catch all exceptions
    pass # TODO : figure out what I want to do
print(something)
```

```
a string
```

Return and read-in multiple values

On you homework, we had the `float_repr` function which returned a pair of integers. When a function returns a `tuple`, you can actually read in the output to different variable names. Here is an example.

In [28]:

```
def float_repr(x) :
    """ Given a float x, returns a pair of ints (c,q)
    such that x = c*2**q exactly. """
    # if a function returns a tuple, you can read off
    # the multiple inputs using this syntax
    (m, n) = x.as_integer_ratio()
    q_minus =  bad_log_base_2(n)
    if q_minus < 0 :
        raise RuntimeError("Unexpected denominator in float.as_integer_ratio()
.")
    return (m,-q_minus)
```

The tuples order is extremely important here. So carefully read what the function outputs before you use it. Sometimes, it also reads well to evaluate multiple things in one line.

In [29]:

```
def int_sqrt(n) :
    """ If n > 0, returns the larges x with x**2 <= n. """
    assert n > 0
    # smallest integer less than n
    if type(n) is not int :
        m = int(n//1)
    else :
        m = n
    # We use Newton's method for the function
    # f(x) = x^2 - n. We start with x_0 = int(n)+1, and apply
    # x_{k+1} = x_k - f(x_k)/f'(x_k) = (x_k + n/x_k)/2
    prev, curr = 0, m
    while True:
        prev, curr = curr, (curr + m // curr) // 2
        # Notice that (curr + n//curr) // 2 is at most
        # 1 less than (curr + n/curr)/2.
        # Thus, by convexity, the first time we are
        # f(curr) is negative, we have out answer
        if curr**2 <= m :
            return curr
```

If you want to ignore all but the first few return values of a function, you can use a dummy variable name. You can chose that name to be anything you want, just keep it consistent.

In [30]:

```
x = 6.6
numerator, _ = x.as_integer_ratio()
# above, the _ acts a dummy varaible name
print(numerator)
# you can also be more verbose
numerator, unused = x.as_integer_ratio()
print(numerator)
print("The unused options are still real \
varaible names. We can print:", _, 'and', unused)
```

```
3715469692580659
3715469692580659
The unused options are still real varaible names. We can print: 56
2949953421312 and 562949953421312
```

Miscellaneous remarks on lists

I wanted to mention a few other tools that can be used on lists (and in other places too).

Enumerate

Sometimes it is useful (or efficient) to loop though a list by looking at the index and value in pairs. There is a useful function `enumerate` for this. It returns a iterable of pairs `(index,value)` in a list. For example,

In [31]:

```
a = ['a',7,8,'n','q',8]
for idx, val in enumerate(a) :
    print("At index {} the list has value {}".format(idx,val))
```

```
At index 0 the list has value a
At index 1 the list has value 7
At index 2 the list has value 8
At index 3 the list has value n
At index 4 the list has value q
At index 5 the list has value 8
```

del keyword

If you want to delete a whole slice in a list, you can do the following.

In [32]:

```
a = [6,4,5,3,5,2]
print(a)
del a[::2] # delete every second element
print(a)
del a[0:2] # delete the first two elements
print(a)
```

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

Note: `del` works only by index, not by value.

sorting

The built-in call `sorted` can work on any iterable such as a `map` object, a tuple, list, or string (along with others).

In [33]:

```
sorted('a string')
```

Out[33]:

```
[' ', 'a', 'g', 'i', 'n', 'r', 's', 't']
```

One of the best tools of sorted is that you can provide a **key function** to help you sort. A `key` function takes **one argument** assigns it a comparable type, such as an integer. The idea is that for every element in a list, you compute a value that you then use to compare. For example,

In [34]:

```
def count_t(s) :
    return s.count('t')

data = "a long string that we will split into words".split()
sorted_data = sorted(data, key = count_t, reverse = True)
print(sorted_data)
```

```
['that', 'string', 'split', 'into', 'a', 'long', 'we', 'will', 'wo
rds']
```

We can also use key functions when looking for minimums and maximums. Here is a problem form the homework for returning the canonical cycle representation.

In [35]:

```
def get_first(x) :
    """ Returns the fist element of x. Assumes x is not empty. """
    return x[0]

def get_second(x) :
    """ Returns the second element of x. Assumes x as at
    least 2 elements. """
    return x[1]

def rot_to_min(x) :
    """ Given a list or tuple x, returns its cyclical rotation
    starting with the minumum value. Return type is the same as x. """
    # enumerate(x) returns an iterable of paits (idx, val)
    min_idx, x_min = min(enumerate(x), key = get_second)
    # return the rotated version
    return x[min_idx:] + x[:min_idx]

def canonical_disjoint_rep(p) :
    """ Returns the canonical cycle decomposition given a tuple
    of tuples representing a disjoint cycle decomposition. All
    empty tuples and singletons are removed. Raises a
    SyntaxError on bad input. """
    if not valid_disjoint_cycle_rep(p) :
        raise SyntaxError("Bad disjoint cycle representation")
    # the identity is already canonical
    if p == ((),) : return p
    # delete any empty or singleton tuples in p
    p_clean = [ c for c in p if len(c) > 1 ]
    # now we can cyclically rotate each tuple in p
    inner_sorted = map(rot_to_min, p_clean)
    # sorting the lists in the inner_sorted iterator
    outer_sorted = sorted(inner_sorted, key = get_first)
    return tuple(outer_sorted)
```

In [36]:

```
a = ( (1,2) , (6,0,4) , (5,3) )
a_canon = canonical_disjoint_rep(a)
print(a_canon)
```

```
((0, 4, 6), (1, 2), (3, 5))
```

Since the `get_first` seems like a commonly used function, there is a module that contains it. There is a module called `operator` that includes the functions `itemgetter` and `attrgetter`. Calling `operator.itemgetter(i)` returns a function with gives the i^{th} element in a list. The function `operator.attrgetter('attr_name')` returns a function which gives the attribute with `attr_name` of an object. Here are two examples.

In [37]:

```
import operator
a = [ ('A',0, 2), ('G', -2, 4) ]
a_sorted = sorted(a, key = operator.itemgetter(1))
print(a_sorted)
```

```
[('G', -2, 4), ('A', 0, 2)]
```

In [38]:

```
import operator
a = [ 1.1 + 5.2j, 3.0 + 6.7j, -2.2 - 3.1j]
a_sorted = sorted(a, key = operator.attrgetter('imag'))
print(a_sorted)
```

```
[(-2.2-3.1j), (1.1+5.2j), (3+6.7j)]
```

Sets

As you have seen, lists and tuples can contain equivalent values multiple times inside the list (i.e. `a = [1,1,1,1]` is a totally valid list even though `1 == 1 is True`). Sets in python are designed to contain equivalent values only once.

In python, sets come in two flavors, **mutable** (type `set`) and **immutable** (type `frozenset`)

To create a `set` in python, we can use the `set()` or `frozenset()` constructors.

In [39]:

```
a = set([1,1,1,'hello'])
b = frozenset([1,1,2,3,2])
print(a)
print(b)
```

```
{1, 'hello'}
frozenset({1, 2, 3})
```

In addition, there is the `{ }` notation, which creates **mutable** sets.

In [40]:

```
a = {1, 'hello', 492.4, 'hello', 1}
print(a)
```

```
{1, 492.4, 'hello'}
```

Sets also support **iteration** and **set comprehension**

In [41]:

```
# set comprehension
a = { x % 10 for x in range(15) }
print(a)
# iteration
v = 0
for x in a :
    v += x
print(v)
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
45
```

Warning : Sets are **unordered**, which means that there is **no guarantee** that iteration will be in any particular order.

Warning : To create an **empty** set you must always use `set()`.

Sets have plenty of useful methods. Here are some methods that work for both `set` and `frozenset`.

- `len(s)` returns the number of elements in set `s` (cardinality of `s`).
- `x in s` tests `x` for membership in `s`. Similarly `x not in s`.
- `s.isdisjoint(other_set)` returns `True` if the sets have empty intersection, `False` otherwise.
- `s.issubset(other_set)` or `s <= other_set` tests whether `s` is a subset of `other_set`.
    - `s < other_set` tests whether `s` is a **proper** subset of `other_set`.
- `s.issuperset(other_set)` or `s >= other_set` tests whether `other_set` is a subset of `s`.
    - `set > other_set` tests whether `other_set` is a **proper** subset of `s`.
- `s.union(other_set)` or `s | other_set` returns a **new** set which is the union of `s` and `other_set`.
- `s.intersection(other_set)` or `s & other_set` returns a **new** set which is the intersection of `s` and `other_set`.
- `s.difference(other_set)` or `s - other_set` returns `s` minus `other_set` in a **new** set.
- `s.symmetric_difference(other_set)` or `s ^ other_set` returns the symmetric difference in a **new** set.
- `s.copy()` returns a **shallow** copy of `s` (you will learn about shallow copies in your homework).

In [42]:

```
a = {1,2,3}
b = {3,4,5}
c = {4,5}
print(a | b)
print(a & b & c) # can intersect multiple sets like this
print(a - b)
print(a ^ b)
print(c < b)
print(a.intersection(b,c)) # can also intersect multiple sets like this
```

```
{1, 2, 3, 4, 5}
set()
{1, 2}
{1, 2, 4, 5}
True
set()
```

There are mutating variants of these operations that only work on `set`.

- `s.update(other_set)` or `s |= other_set` adds the contents of `other_set` to `s`.
- `s.intersection_update(other_set)` or `s &= other_set`.
- `s.difference_update(other_set)` or `s -= other_set`.
- `s.symmetric_difference_update(other-_et)` or `s ^= other_set`.
- `s.add(elem)`
- `s.remove(elem)` removes `elem` from `s` but **raises** KeyError if `elem` is not in `s`.
- `s.discard(elem)` removes `elem` from `s` if present.
- `s.pop()` removes and return an **arbitrary** element from the set. Raises KeyError if the set is empty.
- `s.clear()` removes all elements from `s`.

In [43]:

```
def valid_disjoint_cycle_rep(p) :
    """ Returns True if the input is a tuple of tuples representing
    a disjoint cycle decomposition of a permutation. """
    if type(p) is not tuple or len(p) == 0 : return False
    # keep track of seen integers
    seen = set()
    for c in p :
        if type(c) is not tuple : return False
        for i in c :
            if type(i) is not int or i < 0 :
                return False
            elif i in seen :
                return False
            else :
                seen.add(i)
    return True
```

Not all objects can be added to a set

Unlike a list, a set in python can only contain **immutable** objects. In fact, sets can only contain **hashable** objects. An object is hashable if it has a method `obj.__hash__()`, `obj.hash()`, or `hash(obj)` returns without error. A **hash function** is a map

$$\text{hash} : \{\text{objects of a given type}\} \to \mathbb{Z}.$$

that is very **fast** to compute and satifies

- if `obj_1 ==  obj_2` then `hash(obj_1) == hash(obj_2)`
- if `hash(obj_1) == hash(obj_2)` then `obj_1` and `obj_2` are very likely to be equivalent (==).

Most immutable types in python support hashing. Sets require a hash function to quickly sort, store, and remove objects. You can read more on this at http://www.laurentluce.com/posts/python-dictionary-implementation/

Here is an example from the homework. In the solutions, I also provide a variant without using sets.

In [44]:

```
def int_sqrt(n) :
    """ If n > 0, returns the larges x with x**2 <= n. """
    assert n > 0
    # smallest integer less than n
    if type(n) is not int :
        m = int(n//1)
    else :
        m = n
    # We use Newton's method for the function
    # f(x) = x^2 - n. We start with x_0 = int(n)+1, and apply
    # x_{k+1} = x_k - f(x_k)/f'(x_k) = (x_k + n/x_k)/2
    prev, curr = 0, m
    while True:
        prev, curr = curr, (curr + m // curr) // 2
        # Notice that (curr + n//curr) // 2 is at most
        # 1 less than (curr + n/curr)/2.
        # Thus, by convexity, the first time we are
        # f(curr) is negative, we have out answer
        if curr**2 <= m :
            return curr

def primes_less_than_v2(n) :
    """ Given an integer n, returns the list of primes less than n. """
    if type(n) is not int or n < 3 : return []
    # We start wil all odd numbers less than n
    candidates = set(range(3,n,2))
    # For odd number x = 3, 5, ..., int_sqrt(n), we will
    # remove x^2, x(x+2), x(x+4),... < n from candidates.
    # Notice that we are removing only the odd ones.
    # Why this works : assume y = a*b with 1 < a <= b
    # and y < n, then a < int_sqrt(n), so y will be eliminated.
    for i in range(3, int_sqrt(n) + 1, 2) :
        candidates.difference_update(range(i**2, n, 2*i))
    candidates.add(2)
    return sorted(candidates)
```

In [45]:

```
primes_less_than_v2(30)
```

Out[45]:

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Dictionaries

Dictionaries are another very useful built-in type. They are a like a table of assignment. They map **keys** to **values**. For example,

In [46]:

```
month_to_days = {'April': 30,
 'August': 31,
 'December': 31,
 'February': 28,
 'January': 31,
 'July': 31,
 'June': 30,
 'March': 31,
 'May': 31,
 'November': 30,
 'October': 31,
 'September': 30}

print("October has {} days.".format(month_to_days['October']))
```

```
October has 31 days.
```

Above, the months are they **keys** of the dictionary and the days are the **values**. As you can see, we can construct dictionaries using the notation `{ key_1 : value_1, key_2 : value_2, ... }`. Once also also use the `dict()` constructor that takes a list (or any iterable) of **pairs**. For example,

In [47]:

```
coefs = dict([('x^2',1784),('x',3244), ('1', 9)])
print(coefs)
```

```
{'x^2': 1784, 'x': 3244, '1': 9}
```

You can always read values using the `[]` notation or by calling `.get(key)`. For example

In [48]:

```
print(coefs['1'])
print(coefs.get('x^2'))
```

```
9
1784
```

Dictionaries are **mutable** objects. So you can use the assignment operator to define a new key-value pair.

In [49]:

```
coefs['x^3'] = 209
print(coefs)

a = {} # empty dictionary, NOT a set
a['something'] = 'else'
print(a)
```

```
{'x^2': 1784, 'x': 3244, '1': 9, 'x^3': 209}
{'something': 'else'}
```

We can also delete key-value pairs using the `del` keyword or you can use `pop()` to return and remove.

In [50]:

```
del coefs['1']
print(coefs)

value = coefs.pop('x^3')
print(coefs)
print(value)
```

```
{'x^2': 1784, 'x': 3244, 'x^3': 209}
{'x^2': 1784, 'x': 3244}
209
```

Here are a few other useful commands.

- `some_dict.keys()` return the keys in `some_dict` as an iterable
- `some_dict.values()` return the values in `some_dict` as an iterable
- `some_dict.items()` return the (key,value) paris in `some_dict` as an iterable
- `some_dict.popitem()` remove and return some item from `some_dict`
- `some_dict.update(other_dict)` **merge** the contents of `other_dict` into `some_dict`, overwriting for equivalent keys.
- `some_dict.clear()` remove all contents from a dictionary
- `some_dict.copy()` return a **shallow** copy of the dictionary

Here is a simple function that counts letters in a sting and returns a dictionary with the count.

In [51]:

```
def char_count(string) :
    char_dict = {}
    for c in string :
        char_dict[c] = string.count(c)
    return char_dict

print(char_count("some random string"))
```

```
{'s': 2, 'o': 2, 'm': 2, 'e': 1, ' ': 2, 'r': 2, 'a': 1, 'n': 2, '
d': 1, 't': 1, 'i': 1, 'g': 1}
```

Not all objects can be keys

Just like with sets, the keys of a dictionary have to be **immutable** (and **hashable**) types. Also, just like with sets, **keys don't all have to be the same type**.

In [52]:

```
f = {1 : 2, '1': 2, ('a','non','mutable', 'tuple') : 2}
print(f)
```

```
{1: 2, '1': 2, ('a', 'non', 'mutable', 'tuple'): 2}
```

In [53]:

```
f = {['some', 'mutable', 'list'] : 2}
```

```
-------------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-53-0c641f883a21> in <module>()
----> 1 f = {['some', 'mutable', 'list'] : 2}

TypeError: unhashable type: 'list'
```

Membership, iteration and dictionary comprehension

Just like all other container types we have seen, we can test membership in dictionaries, iterate over dictionaries and use comprehensions to define them. Since dictionaries have both keys and values, we have to be a little careful.

Dictionaries only support the `key in dict` and `key not in dict` for **keys**.

In [54]:

```
coefs = dict([('x^2',1784),('x',3244), ('1', 9)])

if '1' in coefs :
    print("'1' is a key of coefs")
if 9 not in coefs :
    print("9 is not a key of coefs")
```

```
'1' is a key of coefs
9 is not a key of coefs
```

Remark : If you need to search for a value, you can always do that by looking in `list(some_dict.values())`

We can also iterate over the keys, values, or key-value pairs in a dictionary

In [55]:

```
# iterate over keys
for k in coefs :
    print("Found key", k)

print('-'*20)

# iterate over values
for v in coefs.values() :
    print("Found value", v)

print('-'*20)

# iterate over keys and values
for k, v in coefs.items() :
    print("Key {} has value {}.".format(k, v))
```

```
Found key x^2
Found key x
Found key 1
--------------------
Found value 1784
Found value 3244
Found value 9
--------------------
Key x^2 has value 1784.
Key x has value 3244.
Key 1 has value 9.
```

Note: to iterate over keys, you can also use `for k in some_dict.keys()`.

Finally, we mention dictionary comprehension. It's very similar to sets, however, we must use the `:` to separate keys and values.

In [56]:

```
self_powers = { i : i**i for i in range(10) }
print(self_powers)
```

```
{0: 1, 1: 1, 2: 4, 3: 27, 4: 256, 5: 3125, 6: 46656, 7: 823543, 8:
16777216, 9: 387420489}
```

In [57]:

```
coefs_squared = { k : 2*v for k,v in coefs.items() }
print(coefs_squared)
```

```
{'x^2': 3568, 'x': 6488, '1': 18}
```

If you want, you can also think of a dictionary as a map and easily define its inverse with list comprehension. This only works if the values are also hashable.

In [58]:

```
self_roots = { v : k for k,v in self_powers.items() }
print(self_roots)
```

```
{1: 1, 4: 2, 27: 3, 256: 4, 3125: 5, 46656: 6, 823543: 7, 16777216
: 8, 387420489: 9}
```

Warning: If multiple keys have the same value, the above trick will not give consistent results

In [59]:

```
a = { 2: 0, 4: 0, 1 : 0}
a_inv = { v : k for k,v in a.items() }
print(a_inv)
```

```
{0: 1}
```

Recursion

Recursion is a very powerful programming technique that allows for very complex paths of code to be readable and easy to understand. The key tool here is that you can call a function within itself. It works as form of induction to perform computations. Here is an example,

In [60]:

```
def catalan(n) :
    """Computes the n^th Catalan number. """
    assert type(n) is int
    if n < 1 :
        return 1
    else :
        # I can use // here because I know the result will be divisible by n+2
        return (4*n+2)*catalan(n-1)//(n+2)
```

In [61]:

```
catalan(10)
```

Out[61]:

```
58786
```

When using recursion, you have to be very careful that your code paths always terminate! Otherwise, you may end up in in and "infinite" decent (at some point your computer will give up).

The way recursion works is fairly straight forward, however, it has its limits. If you recall, when a function is called, a new namespace (called a **strack frame**) is created. When calling a recursive function, we create a nested sequence of stack frames, which occupies computer memory and takes time. In fact, python has a **maximum recursion depth**. By default, this is set to something rather conservative.

In [65]:

```
catalan(10000)
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-65-3dc4546404fa> in <module>()
----> 1 catalan(10000)

<ipython-input-60-a2efbe6dd8dc> in catalan(n)
      6     else :
      7         # I can use // here because I know the result will be divisible by n+2
----> 8         return (4*n+2)*catalan(n-1)//(n+2)

... last 1 frames repeated, from the frame below ...

<ipython-input-60-a2efbe6dd8dc> in catalan(n)
      6     else :
      7         # I can use // here because I know the result will be divisible by n+2
----> 8         return (4*n+2)*catalan(n-1)//(n+2)

RecursionError: maximum recursion depth exceeded while calling a Python object
```

Remark : The code in `catalan(n)` is called **tail recursive** because it can be easily converted into a for-loop!

In [63]:

```
def find_value(our_list, value) :
    n = len(our_list)
    if n > 0 :
        if value == our_list[n//2] :
            return True
        elif n == 1 : # Our list had only one element
            return False
        else :
            return find_value(our_list[:n//2],value) or \
                   find_value(our_list[n//2+1:],value)
    else :
        return False
```

In [64]:

```
find_value([0,1,2,3,4,5],4)
```

Out[64]:

```
True
```

Let us examine the **decision tree** in this code. We divide the list in two and check if the middle element of the resulting list is what we are looking for, if not we first check **all of the left**, and then all of the right.

Tower of Hanoi

Tower of Hanoi is an interesting mathematical puzzle. It consists of three pegs, and a number of disks of different sizes which can slide onto any peg. The puzzle starts with the disks in a stack in ascending order of size on one peg with the smallest at the top.

The objective of the puzzle is to move the entire stack to another peg, obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.

Exercise : Write a function `tower_of_hanoi(n)` to compute a number of steps to solve the problem when starting with `n` disks on one peg. Use an auxiliary recursive function for your computation. (Note: I know there is a closed form, but just implement using recursion)