Overview

Welcome back! We're almost done with this course and in this lesson we're gonna go through some really mind-boggling concepts.

We're gonna talk about:

- Factory functions,
- Closures,
- and Recursion.

Making own reduce()

To better understand, how does `reduce()` function look behind the scenes, your task will be to implement your own version of it. You can call it `my_reduce()` and it should behave as follows:

```
my_reduce(func, iterable, initializer=None)
```

Different inputs

1. No initializer provided:

```
>>> my_reduce(op.add,range(10))
45
```

2. With initializer:

```
>>> my_reduce(op.add,range(10),3)
48
```

3. Empty iterable:

```
>>> my_reduce(op.add,[],3)
3
```

4. Iterable not provided:

```
>>> my_reduce(op.add,12,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in my_reduce
TypeError: 'int' object is not iterable
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Chessboard Using Comprehension

If you remember, we have already done the chessboard exercies. This time, we will do an advanced version of that, using comprehensions!

Your task is to use create a string that if passed into `print()` function, will look as follows in the terminal:

```
# # #
 # #
# # #
 # #
# # #
```

The string should be generated using list comprehension and you can use only one line of code.

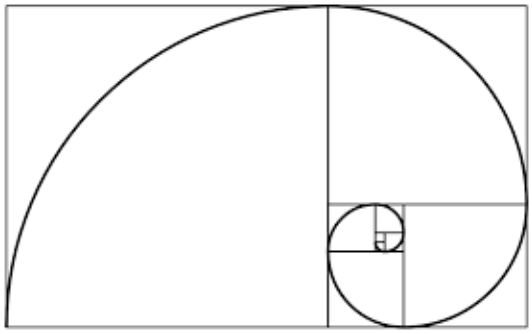
Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

What is Recursion?

Recursion occurs when a object is being defined by itself. This term is well know in fields of mathematics or computer science. Principle of recursion consists in solving a problem by breaking it into smaller and smaller subproblems until we get to a problem small enough that can be solved trivially.



An example of solving a problem recursively is the sum of all numbers smaller than for example number 5. How would we calculate the result? In other words, what is the most trivial operation, we

could do repeatedly in order to arrive to the result?

We could repeatedly discount value 1 from a given number `num` and add the result to the total sum in other words `total += num-1` (5 + (5-1) + ((5-1)-1) + (((5-1)-1)-1) etc.. That means: 5 + (5-1) + (4-1) + (3-1) etc.

The question is, when should the operation be stopped. The answer is, when it reaches so called **base case**. Every recursive solution should have defined a base case. Therefore our recursive solution above will have sense, once we add, that the base case should be value 0.

To sum up, recursive solution to a problem repeatedly performs a given trivial action on continually changing input until the base case value is reached. In that moment the result of all the operations performed is returned.

Recursion in Programming

In programming, recursion involves a function calling itself. That means that a function definition contains call to function that is being defined. This inner call should receive a changed input compared to the input of the enclosing function.

Let's look at the example of summing all the numbers smaller than 5. We could solve this task using **while loop**:

```
1 num = 5
2 total = 0
3 while num > 0:
4     total += num
5     num -= 1
```

The **base case** here is when the variable `num` reaches the value 0.

The above call could be written as a recursive function:

```
1 def sum_rec(num):
2     if num==0:
3         return 0
4     return num + sum_rec(num-1)
```

Calling the above function with the input value 5, we get the following result:

```
>>> sum_rec(5)
15
```

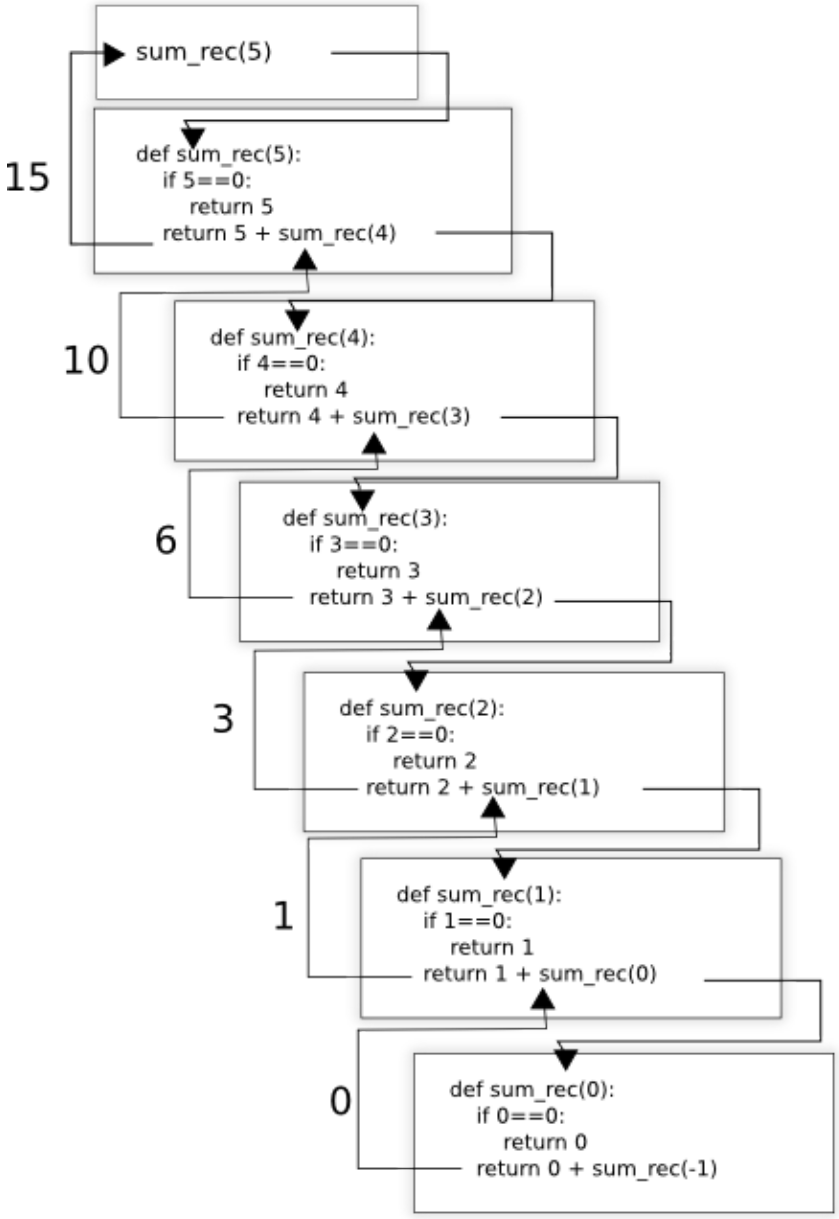
What Actually Happens

In the previous section, we defined and then called the function `sum_rec()` :

```
1 def sum_rec(num):
2     if num==0:
3         return num
4     return num + sum_rec(num-1)
```

On the image below, we can see how the code is actually executed. Python calls recursively function `sum_rec()` until it reaches base case, that is, until the condition `if num==0` evaluates True. This happen inside the sixth execution frame, where function received argument with the value 0.

Until the base case is reached, with each new function call, Python add new execution frame to frame stack. So all the functions that were invoked earlier in the program run, have to wait until those that were invoked later, return.



Missing Base Case

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / RECURSION / MISSING BASE CASE

If we had not adjusted the argument value for the inner function call (`sum_rec(num-1)`) as follows:

```
1 def sum_rec(num):
2     if num==0:
3         return 0
4     return num + sum_rec(num)
```

... we would not be approaching the base case and we would get stuck inside infinite recursive calls tp the function.

In reality, if we submit such code, after a given number of recursive calls a `RecursionError` is be raised:

```
...
...
  File "<stdin>", line 4, in sum_rec
  File "<stdin>", line 2, in sum_rec
RecursionError: maximum recursion depth exceeded in comparison
```

This Python interpreter behaviour is implemented in order only reasonable memory resources are occupied by our progam. It is a kind of a built-in protection.

To find out what recursion limit do we have on our computer, we can call the `sys.getrecursionlimit()` function:

```
>>> sys.getrecursionlimit()
1000
```

Recursion Summary

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / RECURSION / RECURSION SUMMARY

These are the rules that could guide us to formulate recursive solution

1. A recursive solution must have a base case.
2. A recursive solution must change state of inputs and thus move toward the base case.
3. A recursive solution must call itself, recursively.

The last point is actually the most representative of recursion. The first two points are valid whenever we want to repeat a action and get a result. **While loops** have to contain a test that will eventually evaulate to `False` and thus terminate the loop as well as the variable contained inside the condition has to be modified inside the loop.

What is a factory function?

Let's dissect the term **factory function** for better understanding:

- **factory**

Factory is a place where goods are produced. Therefore the same principle will be applied in Python's factory function - we will **produce a new object** - specifically a new **function object**

- **function**

There are two functions in the game:

1. Function that is being created
2. Function that creates the function - the factory function

So an example of a factory function could be a function that we will call `factory()` :

```
def factory():

    def add_num(n):
        return n + 5

    return add_num
```

Inside the `factory()` function, we create a function called `add_num()` and then we return it as the result.

Use of Factory Function

We have our factory function, but we do not know, what to do with it:

```
def factory():

    def add_num(n):
        return n + 5

    return add_num
```

Let's try to call this function to see the result it returns in Python console:

```
>>> factory()
<function factory.<locals>.add_num at 0x7f80dc7e7268>
```

This function returns a function that is somehow related to its creater through `<locals>`. What if we stored the returned `add_num` in a variable `func`? We should be able to call the function as `func(argument)`:

```
>>> func(4)
9
>>> func(5)
10
```

The function works as the definition dictates - it adds number 5 to any number we pass it.

The term factory function is the first part of the story we want to tell in this section. The next part is the term **closure** - explained in the forthcoming section. Before that we will look at the term **enclosing scope** which is created once we have a function defined inside another function.

Enclosing Scope

Until now, we have seen mainly cases, when a function accessed built-in, global or local variables. With factory functions, we need to understand the **enclosing scope** as well.

Enclosing scope is a relative term. It describes location of variables relative to a given place in a code. This relative place will now be the nested function inside the factory function. Variables stored in enclosing scope are created inside enclosing functions. That is, in the example below variables inside the `func1` namespace which is enclosing to `func2` namespace.

In the below example variables:

- `e` , `arg1` , `func2` belong to enclosing scope
- `l` , `arg2` belong to local scope
- `g` , `func1` belong to global scope

```
1 g = 5
2
3 def func1(arg1):
4     e = 3
5
6     def func2(arg2):
7         l = 6
8
9     return func2
```

We need to understand enclosing scope in order we can understand another term - **closure**.

What is closure?

The principle of factory functions may not seem so useful as of now, but **closure** is a term that changes everything.

A closure occurs when a function has access to a variable from an enclosing scope that has finished its execution.

Let's take our `factory()` function we have defined in previous sections and change it a bit:

```
1 def factory():
2     const = 5
3     def add_num(n):
4         return n + const
5 
6     return add_num
```

Output:

```
>>> f = factory()
>>> f(4)
9
>>> f(5)
10
```

We can see that the function `add_num()` stored inside the variable `f` has now access to variable `const` from enclosing scope. Moreover, this access is possible even if the function `factory()` is not being currently executed. This is called **closure** - the inner function closes somehow over the enclosing scope.

In the next section, we will look at how is this possible.

How is it possible?

We have learned that once a function execution is over, all the local variables are destroyed. Why this does not seem to work with closures?

The trick is, that the inner function keeps the track of enclosing scope variables inside the attribute `__closure__`. We can check it ourselves:

```
>>> f.__closure__
(<cell at 0x7f80de470ac8: int object at 0x7f80de2ddaa0>,)
```

This attribute stores a tuple, that lists all the enclosing scope variables - their original content - inside `cell` objects. Our function `factory()` creates one local variable called `const`. Once it is generated by the mother factory function, reference to `const` 's value is then stored inside the inner function's object `__closure__` attribute.

We can now inspect the value of the `cell` object using its `cell_contents` attribute. And it is true, that the original value refered by `const` was 5:

```
>>> f.__closure__[0].cell_contents
5
```

Arguments As Well

Factory function's arguments belong to enclosing scope as well. We can use this knowledge to our benefit and finally create some useful and interesting function.

We will again tweak our `factory()` function a bit, changing the constant enclosing scope value into dynamic enclosing scope value. We achieve this by passing a parameter into `factory()` functions definition:

```
1  def factory(num):
2      def add_num(n):
3          return n + num
4
5      return add_num
```

Now we can begin to generate adder functions with different characteristics. In the below example, we first create a function, that will always add value 2, to whatever value we give it. Then we generate

a new function object, that adds value 54 to whatever value we give it.

```
>>> add_two = factory(2)
>>> add_two(5)
7
>>> add_two(6)
8
>>> add54 = factory(54)
>>> add54(32)
86
```

We now do not have to create new definition for each new factory function based on the default number we want to be added to. We get by with one factory function definition, that creates multitude of different functions - functions, that will return different values with same inputs. This all, thanks to values stored inside the `__closure__` attribute.

Summary

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / FACTORY FUNCTIONS & CLOSURES / SUMMARY

The main characteristics of factory functions are:

- inside a factory function another function is defined
- after the function is defined, it is returned by the factory function
- the returned can be called independently of the original factory function
- values of variables created inside the factory function are remembered by the generated function, this is called closure

No use, no closure

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / FACTORY FUNCTIONS & CLOSURES / NO USE, NO CLOSURE

Until now, in our examples, we have always used variables from the enclosing scope in the inner function. However, we should bare in mind that **the values of enclosing variables, which are not**

used inside the inner function are not recorded in `__closure__`:

- `arg` not included in `__closure__`

```
def wrapper(arg):
    var = 1
    def inner():
        print(var)
    return inner
```

```
>>> f = wrapper(35)
>>> for cell in f.__closure__:
...     print(cell.cell_contents)
...
1
```

- `arg` included in `__closure__`

```
def wrapper(arg):
    var = 1
    def inner():
        print(var,arg)
    return inner
```

```
>>> f = wrapper(35)
>>> for cell in f.__closure__:
...     print(cell.cell_contents)
...
35
1
```

- enclosing variable not in `__closure__`

```
def wrapper(arg):
    var = 1
    def inner():
        print(arg)
    return inner
```

```
>>> f = wrapper(35)
>>> for cell in f.__closure__:
```

```
...     print(cell.cell_contents)
...
35
```

Closure When Generating Multiple Functions

Let's imagine that one day, we would need to create a list of multiple function objects. For example we may want to generate functions that add number 1, the other adds number 2 etc. to a given argument. We would probably generate these functions using for loop.

Here we generate lambda functions:

```
1  def actions_generator(start, stop):
2      actions = []
3      for i in range(start, stop+1):
4          actions.append(lambda x: x+i)
5      return actions
```

Here we generate classic functions:

```
1  def actions_generator(start, stop):
2      actions = []
3      for i in range(start, stop+1):
4          def action(x):
5              return x + i
6          actions.append(action)
7      return actions
```

We store the returned list in a variable `actions` :

```
>>> actions = actions_generator(1,4)
```

Now we could expect, that the first function in the returned `actions` list may add number 1 to whatever value we pass to it:

```
>>> actions[0](2)
```

```
6
```

But that is not true. The first function added number 4 to number 2. What about the second function in the `actions` list:

```
>>> actions[1](2)
6
```

It is the same story. We can see, that every function keeps in the `__closure__` attribute the last value, that has been generated by the **for loop**.

Therefore, we cannot generate functions inside the **for loop** expecting that each one will use value corresponding to the cycle, withing which it has been created.

Intro

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / DATETIME MODULE / INTRO

Already few times happened, that in our programs we needed to record information about time. For example, when we wanted to record date or time:

- when an error occured,
- when a given information has been downloaded from the internet
- when a back-up file has been created
- when a new record has been created
- number of days between two dates etc.

Until now, we how not covered these topics systematically. Therefore in this section we will answer questions like:

- How can record dates, times or both?
- How many days are there between two dates?
- What is todays date? What is the current time?
- How can I format my date and time information?
- What is the weekday of a given date?
- What about timezones, how do I manage them?

Datetime Module

To unlock all the functionality, we have listed in the previous section, we need to import the datetime module into our scripts:

```
import datetime
```

This module provides us with multitude of specialized objects, with their own special powers:

Syntax	Description	Example	Output in Terminal

Syntax	Description	Example	Output in Terminal
`datetime.time`	Records time info (hour, minute, second, microseconds)	`datetime.time(17,34,54,654)`	`datetime.time(17, 34, 54, 654)`
`datetime.date`	Records info about dates only (year, month, day)	`datetime.date(1970,1,1)`	`datetime.date(1970, 1, 1)`
`datetime.datetime`	Records info about date and time	`datetime.datetime(1970,1,1,17,34,54,654)`	`datetime.datetime(1970, 1, 1, 17, 34, 54, 654)`

`datetime.timedelta`	Records info about the difference between two dates or times	`d1 = datetime.date(1970, 1, 1)`
		`d2 = datetime.date(1970,1,15)`
		`d2-d1`
			`datetime.timedelta(14`

Good Import Saves Typing

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / DATETIME MODULE / GOOD IMPORT SAVES TYPING

Name of the `datetime` module as well as object names it defines are pretty long and therefore, if we want to save typing in our scripts, that use the module's funcitonality, we want to probably import the individual object names (constructors) as follows:

```
1 from datetime import datetime as dt
2 from datetime import date as d
3 from datetime import time as t
4 from datetime import timedelta as td
```

Date and Time Objects' Attributes

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / DATETIME MODULE / DATE AND TIME OBJECTS' ATTRIBUTES

So let's say we have the following datetime objects:

```
>>> t = datetime.time(17,34,54,654)
>>> d1 = datetime.date(1970, 1, 1)
>>> d2 = datetime.date(1970,1,15)
>>> dd = d2 - d1
```

Each of the previously presented objects can be decomposed as follows:

Datetime object	Getting attribute	Output in Terminal
`t`	`t.hour`	17
	`t.minute`	34
	`t.second`	54
	`t.microsecond`	654
`d1`	`d1.year`	1970
	`d1.month`	1
	`d1.day`	1
`dd`	`dd.days`	14
	`dd.seconds`	0
	`dd.microseconds`	0

Difference Between dt objects

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / DATETIME MODULE / DIFFERENCE BETWEEN DT OBJECTS

To calculate the difference between two dates, we use the minus `-` operator. The result returned by such operation is a **timedelta** object:

Difference between two dates

The difference between two dates will **always be number of days**, not seconds or microseconds:

```
>>> d1 = datetime.date(1970, 1, 1)
>>> d2 = datetime.date(1970, 1, 15)
>>> td = d2-d1
>>> td
```

```
datetime.timedelta(14)
>>> td.seconds
0
>>> td.days
14
```

Difference between two datetimes

The difference among two `datetime` objects can be stored as `timedelta` represented in number of days, seconds and microseconds:

```
>>> dt1
datetime.datetime(1970, 1, 1, 17, 34, 54, 654)
>>> dt2
datetime.datetime(1970, 1, 14, 12, 12, 34)
>>> td = dt2-dt1
>>> td
datetime.timedelta(12, 67059, 999346)
```

Difference between two times

Time objects do not support arithmetic operations:

```
>>> t1 = datetime.time(17, 34, 54)
>>> t2 = datetime.time(16, 15, 24)
>>> t2-t1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'datetime.time' and
'datetime.time'
>>> t1+t2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'datetime.time' and
'datetime.time'
```

We cannot subtract `date` object from a `datetime` object and vice versa. If the subtracted date or datetime value is greater than the result is represented in negative number of days:

```
>>> dt1-dt2
```

```
datetime.timedelta(-13, 19340, 654)
```

Adding Days to Dates

Sometimes in our programs, we may need to increase a `date` or `datetime` value by a given number days or possibly seconds and microseconds. This is supported by both objects under the condition that the original value is increased by a `timedelta` object, not `date` or `datetime`.

Example:

What is the date in 14 days from `datetime.date(2034,8,28)`?

```
>>> d = datetime.date(2034,8,28)
>>> new_d = d + datetime.timedelta(days=14)
>>> new_d
datetime.date(2034, 9, 11)
```

If we provided `seconds` parameter as well, its influence will depend on, whether the number of seconds is greater than the number of seconds in one day (24*3600):

- no influence

```
>>> new_d = d + datetime.timedelta(days=14, seconds=2600)
>>> new_d
datetime.date(2034, 9, 11)
```

- increased by 1 day

```
>>> new_d = d + datetime.timedelta(days=14, seconds=24*3600 + 356)
>>> new_d
datetime.date(2034, 9, 12)
```

Increasing Datetime

The `datetime` objects support the increase operation in the same manner as `date` objects. Only `timedelta` value can be added to a `datetime` value. The difference is, that we can keep track of change in seconds and microseconds if needed - these are not discarded as is the case with `date` object.

Example:

What will be the date and time in 12 days and 51367 seconds from `datetime(2034,12,31,10,12,34)` ?

```
>>> dt = datetime.datetime(2034,12,31,10,12,34)
>>> new_dt = dt + datetime.timedelta(days=12,seconds=51367)
>>> new_dt
datetime.datetime(2035, 1, 13, 0, 28, 41)
```

Subtracting from Date & Datetime

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / DATETIME MODULE / SUBTRACTING FROM DATE & DATETIME

To find out, what was the date or date and time a given number of days, seconds or microseconds before a given `date` or `datetime` , we have to subtract the `timedelta` object.

Here we subtract 12 days and 34 seconds from the orginal `datetime` value:

```
>>> dt = datetime.datetime(2034, 12, 31, 10, 12, 34)
>>> new_dt = dt - datetime.timedelta(12,34)
>>> new_dt
datetime.datetime(2034, 12, 19, 10, 12)
```

Here we subtract 32 days from the original `date` value:

```
>>> d = datetime.date(2034, 8, 28)
>>> new_d = d - datetime.timedelta(seconds=24*3600*32)
>>> new_d
datetime.date(2034, 7, 27)
```

Today's Date and Current Time

Both, `date` and `datetime` objects provides us with few handy methods. One of those is a method that:

1. provides today's date
2. provides current date and time

We will import both `date` and `datetime` objects as follows:

```
>>> from datetime import date as d
>>> from datetime import datetime as dt
```

Today's date

As this article has been written on the 1st of January, 1970, the `date`'s `today()` method returns the following `date` object:

```
>>> d.today()
datetime.date(1970, 1, 1)
```

Current date and time

The `datetime` object provides more possibilities to retrieve the current date and time:

- `datetime.datetime.today()` - shows current date and time without possibility to specify time zone related information
- `datetime.datetime.now(tz=None)` - shows current date and time. more precise than time.time(). Possible make use of tzinfo - specify the time zone
- `datetime.datetime.utcnow()` - returns current utc time (minus two hours compared to central european time)

We will not yet work with time zones, therefore we do not pass any argument to `now()`:

```
>>> dt.today()
datetime.datetime(1970, 1, 1, 13, 9, 8, 957772)
>>> dt.now()
```

```
datetime.datetime(1970, 1, 1, 13, 9, 20, 775283)
>>> dt.utcnow()
datetime.datetime(1970, 1, 1, 11, 9, 24, 261556)
```

Date and Time Formatting

For now, we are able to perform calculations with dates and times, but we are not able to present the results in a form, that would be understandable to broad audience. Date and time has usually a given format of depiction in different parts of the world:

Examples of date and time formats:

- 'Fri Dec 6 00:00:00 2017'
- '2016-10-06T13:30:05.144542'

We can see, that all the above dates are actually strings. Therefore, our goal will now be to convert `date` or `datetime` objects into their formatted string representations.

Conversion between string and `date` / `datetime` objects consists in using **formatting flags**. Below we list few of those:

Flag	Description
`%Y`	Indicates place for year value
`%m`	Indicates place for month value
`%d`	Indicates place for day value
`%H`	Indicates place for hour value
`%M`	Indicates place for minute value
`%S`	Indicates place for second value

Example of a formatting string:

```
'%d.%m.%Y %H:%M:%S'
```

The above listed placeholders will be replaced by the actual values, everything else will be kept in the formatting string.

Converting date/datetime object into a formatted string

This conversion is performed using `strftime()` - string format time - method that takes the formatting string `'%d.%m.%Y %H:%M:%S'` as an argument:

```
>>> my_d = datetime.date(1970, 1, 1)
>>> my_d.strftime('%d.%m.%Y %H:%M:%S')
'01.01.1970 00:00:00'
```

```
>>> my_dt = datetime.datetime(1970, 1, 1, 17, 34, 54, 654)
>>> my_dt.strftime('%d.%m.%Y %H:%M:%S')
'01.01.1970 17:34:54'
```

We can pass two arguments to `strftime()` (the `date` or `datetime` object and the formatting string) if we call the method directly on the imported `date` or `datetime` class. This time, we will use different formatting string (replace dot `.` for slash `/`) just to demonstrate, how the result changes:

```
>>> d.strftime(my_d,'%d/%m/%Y %H:%M:%S')
'01/01/1970 00:00:00'
>>> dt.strftime(my_dt,'%d/%m/%Y %H:%M:%S')
'01/01/1970 17:34:54'
```

Converting formatted string into a date/datetime object

To convert a formatted string back to `datetime` object, we need to use `strptime()` - string parse time - method. `date` object unfortunately does not support this method:

```
>>> formatted = '01/01/1970 00:00:00'
>>> d.strptime(formatted,'%d/%m/%Y %H:%M:%S')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'datetime.date' has no attribute 'strptime'
```

However, this will work:

```
>>> formatted = '01/01/1970 17:34:54'
```

```
>>> dt.strptime(formatted,'%d/%m/%Y %H:%M:%S')
datetime.datetime(1970, 1, 1, 17, 34, 54)
```

Even with date only information:

```
>>> formatted = '01/01/1970'
>>> dt.strptime(formatted,'%d/%m/%Y')
datetime.datetime(1970, 1, 1, 0, 0)
```

For more formatting flags, see the Python documentation

Why Formatting

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / DATETIME MODULE / WHY FORMATTING

It is very important to learn date and time formatting in order we can write this information in a meaningful way into a file for example.

If we tried to write pure `date(2016,8,21)` or `datetime(2016,8,21,11,32)` object into a file, we would end up with error:

```
>>> d = date(2016,8,21)
>>> dt = datetime(2016,8,21,11,32)
>>> with open('time.txt','w') as f:
...     f.write(d)
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: write() argument must be str, not datetime.date
```

Weekday

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / DATETIME MODULE / WEEKDAY

Weekday refers to the order of a day in a week. Both `date` and `datetime` objects provide a method `weekday()` that returns an integer between 0 (Monday) to 6 (Sunday).

```
>>> from datetime import datetime
>>> dt = datetime.datetime(2016, 8, 21, 11, 32)
>>> dt.weekday()
6
```

To determine today's weekday number, we just chain the `today()` and `weekday()` method calls:

```
>>> datetime.today().weekday()
4
```

There is another method, `isoweekday()`, that returns values between 1 and 7 and works for both `date` and `datetime` objects:

```
>>> date(2017,10,9).isoweekday()
1
>>> date(2017,10,8).isoweekday()
7
```

Current Time

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / DATETIME MODULE / CURRENT TIME

To find out the current time, we have to use a new module called time:

```
import time
```

This module contains function called `time()` that returns the time in seconds since the epoch as a floating point number. The epoch is the point where the time starts, for the big part of the computer world is January 1, 1970, 00:00:00 (UTC).

We can use the `time.time()` function to determine the difference btw. two points in time:

```
1 import time
2 i=0
3 size = 36000
4 start = time.time()
5 while i<size:
6     i+=1
7 end = time.time()
```

```
8
9 print('Iterated {} times in {} seconds'.format(size, end-start))
```

The output could be:

```
Iterated 36000 times in 0.45011043548584 seconds
```

Recursion

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / QUIZ / RECURSION

1/4

What is the difference between a recursive function and while loop?

A. while does not have base case - termination condition

B. while does not perform recursive function call

C. while does not change the value of variables that are considered in the termination condition

D. only loops can iterate over sequence of items

Closures

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / QUIZ / CLOSURES

1/10

Which functions are called factory functions?

A. Recursive functions

B. Functions that create and return new functions

C. Functions that are returned by another function

D. Functions imported from another module

Datetime Module

1/8

What will be the result of the following operation?:

```
>>> from datetime import datetime, date
>>> datetime(2017,8,12,18,34) - date(2017,7,28)
```

A. TypeError - mixing datetime with date not permitted

B. Date object

C. Datetime object

D. Timedelta object

Fibonacci Numbers

Create a recursive function that will return **n-th** number from the Fibonacci sequence. Fibonacci sequence is characterized by the fact that every number after the first two is the sum of the two preceding ones:

0, 1, 1, 2, 3, 5, 8, 13, ...

Example of use:

```
>>> fibo(5)
5
>>> fibo(6)
8
>>> fibo(7)
13
>>> fibo(8)
21
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Traversing Dict of Dicts

Your task is to create a recursive function `traverse_dict` that will traverse all the branches of the below dictionary and will sum the numbers at its ends:

```
d = {
    'AA':{
            'AAA':{'AAAA':1,
                   'AAAB':2,
                   'AAAC':3
            },
            'AAB':{'AABA':4,
                   'AABB':5,
                   'AABC':6
            }
        },
    'AB':{
            'ABA':{'ABAA': {'ABAAA':17,
                            'ABAAB':12,
                            'ABAAC':15,
                            'ABAAD':16,
                            'ABAAE':1
                            },
                   'ABAB':{'ABABA':{'ABABAA':32,
                                    'ABABAB':54,
                                    'ABABAC':13,
                                    'ABABAD':98,
                                    },
                           'ABABB':65,
                           'ABABC':6,
                           'ABABD':12,
                           },
                   'ABAC':{'ABACA':7,
                           'ABACB':73
                          },
            },
            'ABB':{'ABBA':10,
```

```
                    'ABBB':11,
                    'ABBC':12
                }

            },

        'AC':{
                'ACA':{'ACAA':7,
                    'ACAB':8,
                    'ACAC':9
                },
                'ACB':{'ACBA':10,
                    'ACBB':11,
                    'ACBC':12
                }
            }
        }
```

The result should be as follows:

```
>>> traverse_dict(d)
532
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Greatest Common Divisor

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / EXERCISES / GREATEST COMMON DIVISOR

In the past we have created a function that calculated the greatest common divisor of two numbers.

Now it is time to create its recursive version.

Example of use:

```
>>> gcd(33,22)
11
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Recursive Reversed

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / EXERCISES / RECURSIVE REVERSED

Create a recursive function that will list all the digits in a number separately in reverse order.

Example of use:

```
>>> l = [0, 1, 2, 3, 4, 5, 6, 7]
>>> reverse(l)
[7, 6, 5, 4, 3, 2, 1, 0]
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Permutations

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / EXERCISES / PERMUTATIONS

Permutations represent **ways of arranging of all the members of a set into some order**. To get all the permutations of a sequence means to find all the possible orderings of elements in a set.

There can be permutations with and without repetition:

- Number of permutations without repetition of n elements is **n!**
- Number of permutations with repetition of n elements is $\mathbf{n^n}$

And if we are just selecting a subset of length k from a set, then:

- Number of permutations without repetition of n elements is **n!/(n-k)!**
- Number of permutations with repetition of n elements is $\mathbf{n^k}$

For more information see:

- mathisfun.com
- betterexplained.com

Task 1

Create a function that will return all permutations of given length for a given sequence without item repetition.

Example of function in use:

```
>>> p=permutations(['A','B','C','D'],4)
>>> print(*p,sep='\n')
['A', 'B', 'C', 'D']
['A', 'B', 'D', 'C']
['A', 'C', 'B', 'D']
['A', 'C', 'D', 'B']
['A', 'D', 'B', 'C']
['A', 'D', 'C', 'B']
['B', 'A', 'C', 'D']
['B', 'A', 'D', 'C']
['B', 'C', 'A', 'D']
['B', 'C', 'D', 'A']
['B', 'D', 'A', 'C']
```

```
['B', 'D', 'C', 'A']
['C', 'A', 'B', 'D']
['C', 'A', 'D', 'B']
['C', 'B', 'A', 'D']
['C', 'B', 'D', 'A']
['C', 'D', 'A', 'B']
['C', 'D', 'B', 'A']
['D', 'A', 'B', 'C']
['D', 'A', 'C', 'B']
['D', 'B', 'A', 'C']
['D', 'B', 'C', 'A']
['D', 'C', 'A', 'B']
['D', 'C', 'B', 'A']
```

Task 2

Create a function that will allow user to choose, whether the items in the generated permutations can repeat:

Example of function in use:

```
>>> p = permutations(['a','b','c','d'],4,repeat=True)
>>> len(p)
256
>>> print(*p,sep='\n')
['a', 'a', 'a', 'a']
['a', 'a', 'a', 'b']
['a', 'a', 'a', 'c']
['a', 'a', 'a', 'd']
['a', 'a', 'b', 'a']
['a', 'a', 'b', 'b']
```

Example of use:

- Sorting
- Anagrams
- Brute Force Password Cracking

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Combinations

Combinations are about selecting **k** elements from a set of length **n**, where the order does not matter. That means that (1,2,3) is the same as any of the tuples (3,2,1), (2,3,1), (3,1,2),(2,1,3),(1,3,2).

The program, that will list all combinations for us, should answer the questions, what are the ways of selecting items from a collection. For example, if we had to decide about creating special task force from our employees, what are possible teams, we could create.

There can be combinations with and without repetition, when selecting k elements from the set:

- Number of combinations without repetition of k elements is **n!/((n-k)!k!)**
- Number of combinations with repetition of k elements is **(n-1+k)!/((n-1)!k!)**

Example of use: Calculating the number of all possible combinations is used within the area of probability calculation.

Task:

Create a function that will return all combinations of **k** items selected from a given sequence.

Example function in use:

- Without repetition:

```
>>> c = combinations(['A','B','C','D'],2)
>>> print(*c,sep='\n')
['A', 'B']
['A', 'C']
['A', 'D']
```

```
['B', 'C']
['B', 'D']
['C', 'D']
```

- With repetition

```
>>> c = combinations(['A','B','C','D'],2,repeat=True)
>>> print(*c,sep='\n')
['A', 'A']
['A', 'B']
['A', 'C']
['A', 'D']
['B', 'B']
['B', 'C']
['B', 'D']
['C', 'C']
['C', 'D']
['D', 'D']
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Factory Calculator

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / EXERCISES / FACTORY CALCULATOR

Your task is to create a factory function `calculator()` that will return a function which will behave as a classical calculator.

One of the features of the calculator is that it remembers the currently calculated value.

Example of calculator in use:

```
>>> c = calculator()
>>> c(3)
3
>>> c('+3')
6
>>> c(4)
4
>>> c('5+4')
9
>>> c('-6')
3
```

To fulfill this task, you will need to learn about the built-in function `eval()` that expects a string representing python code. It then runs the code inside the string and returns the result.

Example:

```
>>> eval('5+3')
8
```

For more information about `eval` , see Python documentation

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Printing Folder Tree

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / EXERCISES / PRINTING FOLDER TREE

Your task is to create a recursive function `tree` that will serve to display the contents of the folder and folders that it contains.

Example of function in use on my computer:

```
>>> tree('/home/martin/PythonBeginner')
|___Lesson2
    |___unique.py
    |___list_items.py
    |___work
        |___repeat_lesson1.py
        |___Solved
            |___buggy_review.py
            |___student_names.py
        |___lesson2.md
        |___OnSite.zip
        |___day_num.py
    |___echo.py
    |___student_names.py
    |___sum_powers.py
    |___longest_word.py
    |___lesson2.md~
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Recursive Folder Deletion

PYTHON ACADEMY / 14. [HOME] CLOSURES, RECURSION / EXERCISES / RECURSIVE FOLDER DELETION

To remove the contents of a folder, we had to use `shutil.rmtree()` function. Now, when we know, how the recursion works, we can create our own function that will remove non-empty directories from a computer. All we need is `os` module.

We have the following folder structure:

```
|___Level1.1
```

```
    |___Level2.1
        |___test1.2.1
    |___Level2.2
        |___test1.2.1.2
|___Level1.2.
    |___Level2.1.
        |___test1.2.2.1
```

We cannot us `os.remove()` on non-empty directories

```
>>> os.remove('Level0')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IsADirectoryError: [Errno 21] Is a directory: 'Level0'
```

Example of function in use:

```
>>> import os
>>> rm_r('Level0')
Removed file: Level0/Level1.1/Level2.1/test1.2.1
Removed dir: Level0/Level1.1/Level2.1
Removed file: Level0/Level1.1/Level2.2/test1.2.1.2
Removed dir: Level0/Level1.1/Level2.2
Removed dir: Level0/Level1.1
Removed file: Level0/Level1.2./Level2.1./test1.2.2.1
Removed dir: Level0/Level1.2./Level2.1.
Removed dir: Level0/Level1.2.
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

DALŠÍ LEKCE