Python for Data Analysis

Session 1. Introduction

Session agenda

- Introduction
- Review of Python basics
- Basic data and sequences types.
- Functions.
- Elements of functional programming in Python: lambdas and list comprehensions.
- Python modules. The Python package index. Installing new packages.
- IPython. Setting up environment for data analysis.

A word about data science

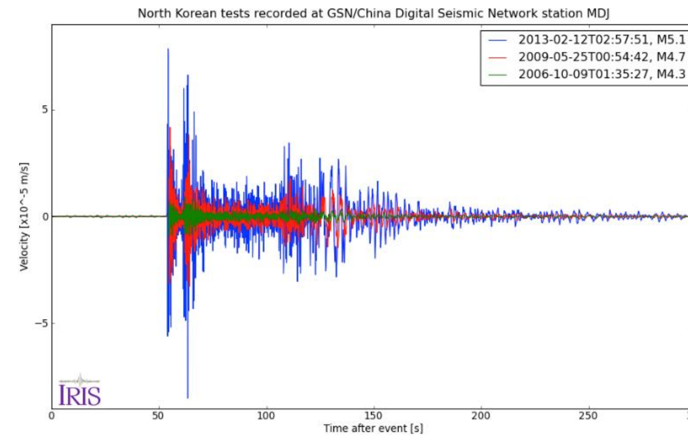
- *Data science*, also known as *data-driven* science, is an interdisciplinary field about scientific methods, processes and systems to extract knowledge or insights from data in various forms, either structured or unstructured
- *Data analysis* is a process of inspecting, cleansing, transforming, and modeling data with the goal of discovering useful information, suggesting conclusions, and supporting decision-making.

Our world is full of data

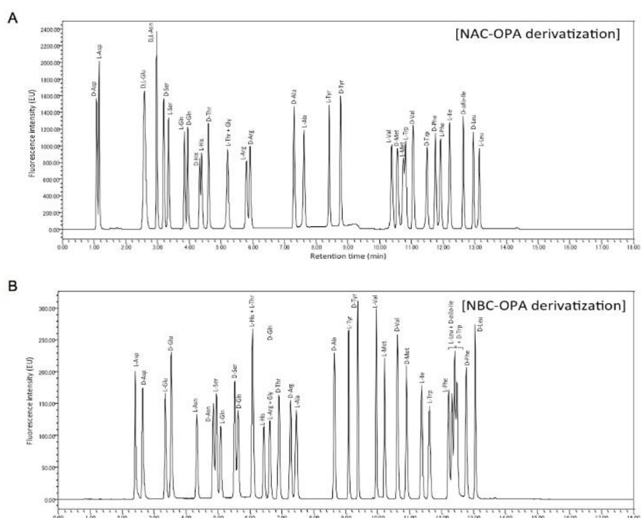
- Data come in a variety of forms and reside in a variety of objects and processes
- Mankind produced and collected huge amount of data
- We produce even more information every day
- Let us review a couple of examples of data forms collected in different fields

Time series data

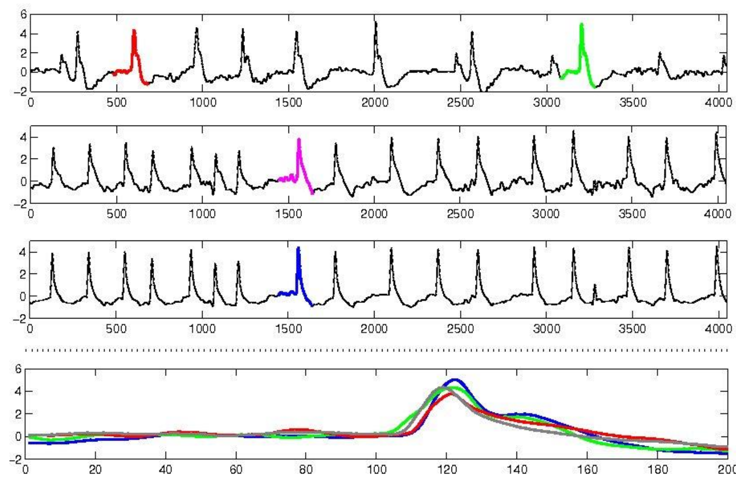
Financial data



Seismic data

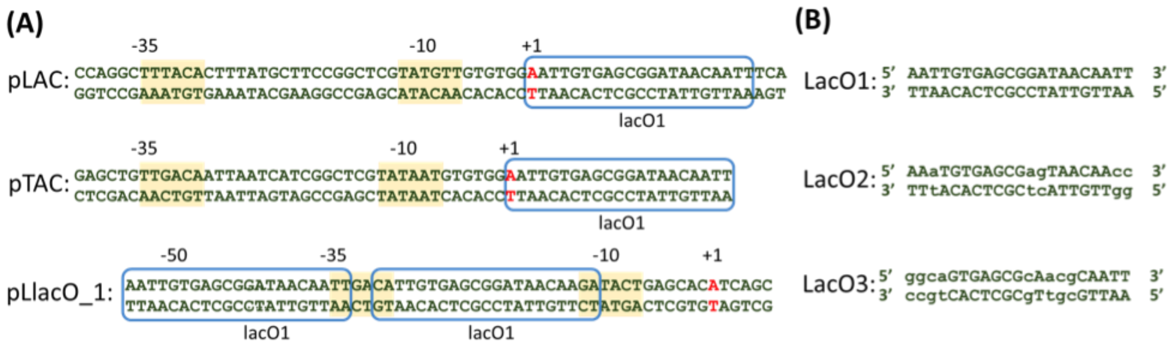


Chemical analytics

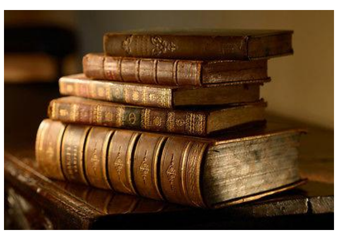


Process monitoring

Textual data



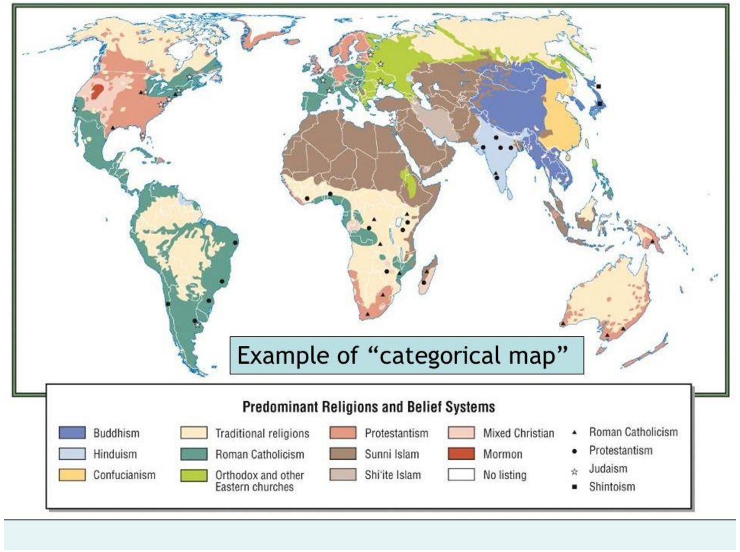
DNA sequences




Books, letters, webpages, emails, sms, tweets and many more

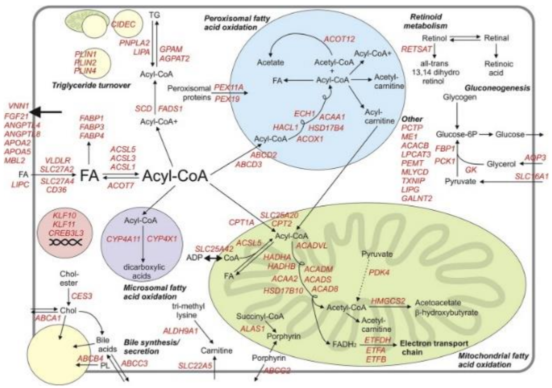
Categorical data

Surveys

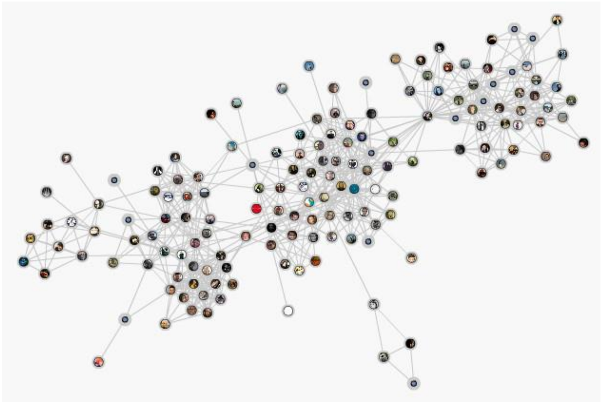


Collected geographical, social and historical data

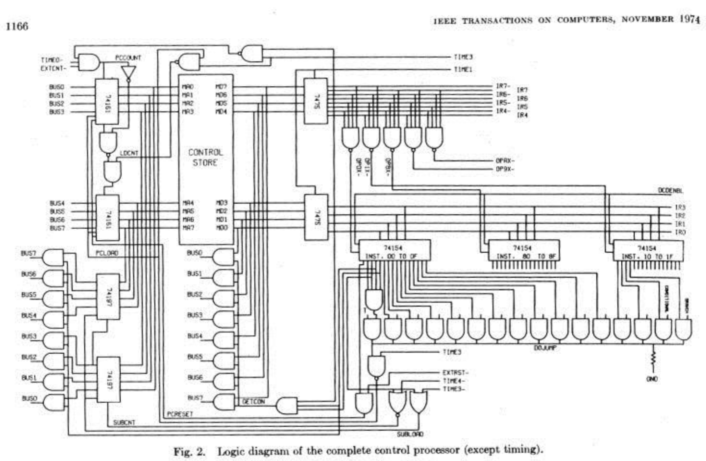
Graphs



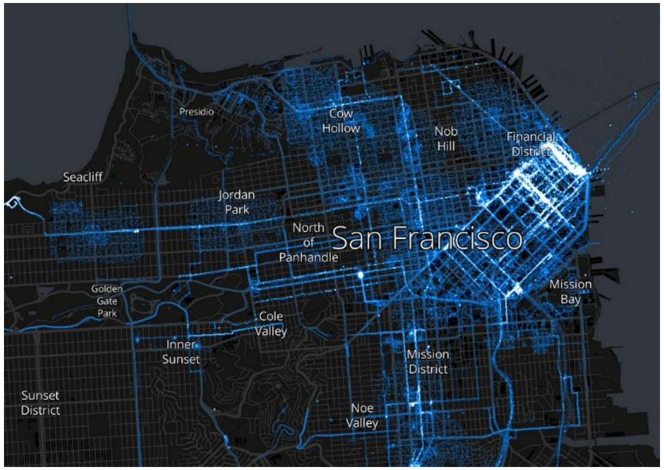
Metabolic networks



Social networks

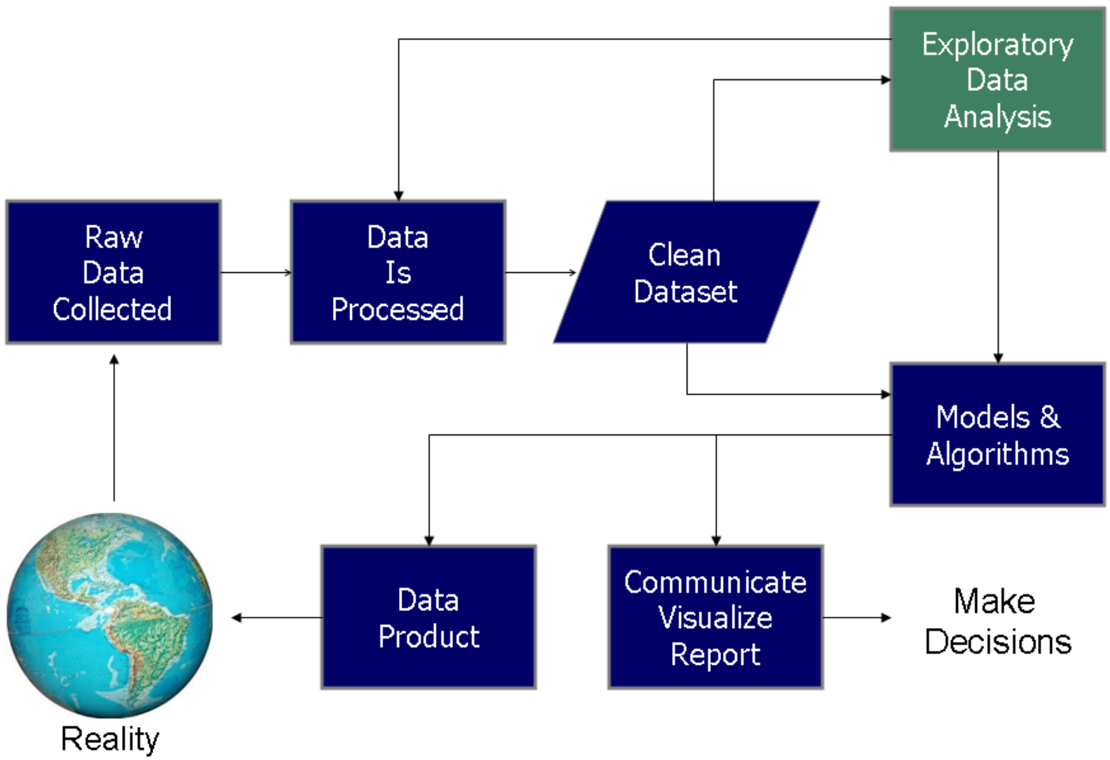


Integrated circuits



Road networks

Data Science Process



Data scientist

- Data scientist combines skills and expertise from the following fields:
  - Computer science:
    - Programming languages
    - Computer architecture
    - Databases
    - Data science toolboxes
  - Mathematical statistics
  - Machine learning
  - Expert knowledge in the field from which data is collected

Course overview

- Week 1
  - Review of key features and concepts of Python programming language
- Week 2
  - Python toolboxes for basic data processing tasks: extraction, preparation, manipulation and visualization
- Week 3
  - Basic data analysis and machine learning tasks. Python toolboxes for data preprocessing, regression analysis, classification and clustering.

About Python

- Development started in the 1980's by Guido van Rossum.
- Two versions: Python 2.x and Python 3.x without backward compatibility
- Interpreted, very-high-level programming language.
- Supports a multitude of programming paradigms.
  - OOP, functional, procedural, logic, structured, etc.
- General purpose.
  - Very comprehensive standard library includes numeric modules, crypto services, OS interfaces, networking modules, GUI support, development tools, etc.
  - A huge number of additional packages supported by the community for a variety of fields and applications

Philosophy

- From *The Zen of Python* (https://www.python.org/dev/peps/pep-0020/)
- *Beautiful is better than ugly.*
  Explicit is better than implicit.
  Simple is better than complex.
  Complex is better than complicated.
  Flat is better than nested.
  Sparse is better than dense.
  Readability counts.
  Special cases aren't special enough to break the rules.
  Although practicality beats purity.
  Errors should never pass silently.
  Unless explicitly silenced.
  In the face of ambiguity, refuse the temptation to guess.
  There should be one-- and preferably only one --obvious way to do it.
  Although that way may not be obvious at first unless you're Dutch.
  Now is better than never.
  Although never is often better than ***right*** *now.*
  If the implementation is hard to explain, it's a bad idea.
  If the implementation is easy to explain, it may be a good idea.
  Namespaces are one honking great idea -- let's do more of those!

Notable Features

- Easy to learn.
- Supports quick development.
- Cross-platform.
- Open Source.
- Extensible.
- Embeddable.
- Large standard library and active community.
- Useful for a wide variety of applications.

Licensing and use in commercial applications

- Python is distributed under Python Software Foundation License
- Most Python third-party libraries and packages are distributed under BSD (Berkeley Software Distribution) license
- These licenses provide free usage and modification of provided software as well as free linkage with any commercial third-party software

Coding style

- So now that we know how to write a Python program, let’s break for a bit to think about our coding style. Python has a style guide that is useful to follow, you can read about it in PEP8.
- I encourage you all to check out pylint, a Python source code analyzer that helps you maintain good coding standards.

Interpreter

- The standard implementation of Python is interpreted (CPython).
    - There are various alternative implementations (IronPython, Jython, PyPy and etc.)
- The interpreter translates Python code into bytecode, and this bytecode is executed by the Python VM (similar to Java).
- Two modes: normal and interactive.
    - Normal mode: entire .py files are provided to the interpreter.
    - Interactive mode: read-eval-print loop (REPL) executes statements piecewise.

Python programming paradigms

- Python supports a number of fundamental programming paradigms:
  - Python is designed on the basis of object-oriented programming (OOP) paradigm. Detailed revision of OOP in Session 2
  - Python supports functional programming. Detailed revision in Session 4.
  - Python also supports procedural and imperative programming

Python typing

- Python is a strongly, dynamically typed language.
- Strong Typing
  - Obviously, Python isn’t performing static type checking, but it does prevent mixing operations between mismatched types.
  - Explicit conversions are required in order to mix types.
- Dynamic Typing
  - All type checking is done at runtime.
  - No need to declare a variable or give it a type before use.

Python built-in types

- Boolean type: bool
- Numeric types:
  - int, long, float and complex.
- There are seven sequence subtypes:
  - str
  - list, tuple, range
  - bytes, bytearrays, memoryview.
- Set types
  - set, frozenset
- Mapping types
  - dict
- More on Python data types in Session 3

Boolean type - example

- True and False are keywords for bool type values
- Bool type supports basic logic operations not, and, or
- Comparison operations return bool type values
- Comparison operations can be chained

```
$ python
>>> 3 < 2
False
>>> i = 2
>>> il1 = 1 < i < 3
True
>>> il2 = not( il1 )
False
>>> il2 or il1
True
>>> type(i) is int
True
```

Numeric Types - examples

- Integer type is unlimited
- Common arithmetic operations are supported
- Numeric types are classes
- Type conversion is done by type's constructor

```
$ python
>>> 3 + 2
5
>>> 18 % 5
3
>>> abs(-7)
7
>>> float(9)
9.0
>>> int(5.3)
5
>>> complex(1,2)
(1+2j)
>>> 2 ** 8
256
```

Strings - examples

```
s1 = 'Just some string'
s2 = ' and some other string`

s3 = s1 + s2
print(s3)
Just some string and some other string

print(s3[5:9])
some

l = s3.split()
print(l)
['Just', 'some', 'string', 'and', 'some', 'other', 'string']

s4 = 'Variable {0} has value {1}'
fs = s4.format('a',4.3)
print(fs)
Variable a has value 4.3
```

Lists - examples

```
l1 = [1,2,'test',complex(1,2)]
print(l1[3])
(1+2j)

l2 = [3,2,5,7,6,0]
l2.sort()
print(l2)
[0, 2, 3, 5, 6, 7]

print(l2[1:4:2])
[2, 5]

l2.pop()
print(l2)
[0, 2, 3, 5, 6]

l3 = [[1,2],[3,4],[5,6]]
print(l3[2][1])
6
```

Sets - examples

```
bases = {'A','T','G','C'}
print(bases)
{'C', 'T', 'A', 'G'}

'A' in bases
True

r_bases = {'A','G','C','U'}
print(bases.symmetric_difference(r_bases))
{'U', 'T'}

print(bases.intersection(r_bases))
{'C', 'A', 'G'}
```

Dictionaries - examples

```
b_ratios = dict()
b_ratios['A'] = 0.26
b_ratios['T'] = 0.239
b_ratios['G'] = 0.249
b_ratios['C'] = 0.252

b_ratios.keys()
dict_keys(['C', 'T', 'A', 'G'])

b_ratios.values()
dict_values([0.252, 0.239, 0.26, 0.249])

'A' in b_ratios
True

b_ratios['A']
0.26
```

Conditional statements

- The if statement has the following general form.

```
if expression:
    statements
```

- If the boolean expression evaluates to True, the statements are executed. Otherwise, they are skipped entirely.

```
a = 1
b = 0
if a:
    print( "a is true!" )
if not b:
    print( "b is false!" )
if a and b:
    print( "a and b are true!" )
if a or b:
    print( "a or b is true!" )
```

```
a is true!
b is false!
a or b is true!
```

Conditional statements

- You can also pair an else with an if statement.

```
if expression:
    statements
else:
    statements
```

- The elif keyword can be used to specify an else if statement.
- Furthermore, if statements may be nested within each other.

```
a = 1
b = 0
c = 2
if a > b:
    if a > c:
        print "a is greatest"
    else:
        print "c is greatest"
elif b > c:
    print "b is greatest"
else:
    print "c is greatest"
```

```
c is greatest
```

Loops - while

- While loops have the following general structure.

```
while expression:
        statements
```

- Here, **statements** refers to one or more lines of Python code. The conditional expression may be any expression, where any non-zero value is true. The loop iterates while the expression is true.
- Note: All the statements indented by the same amount after a programming construct are considered to be part of a single block of code.

```
i = 1
while i < 4:
    print i
    i = i + 1
flag = True
while flag and i < 8:
    print( flag, i )
    i = i + 1
```

```
1
2
3
True 4
True 5
True 6
True 7
```

Loops - for

- The for loop has the following general form.

```
for var in sequence:
    statements
```

- If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable **var**. Next, the statements are executed. Each item in the sequence is assigned to **var**, and the statements are executed until the entire sequence is exhausted.
- For loops may be nested with other control flow tools such as while loops and if statements.

```
for letter in "aeiou":
    print( "vowel: ", letter )
for i in [1,2,3]:
    print( i )
```

```
vowel: a
vowel: e
vowel: i
vowel: o
vowel: u
1
2
3
```

Built-in range() function

- Python has range() function for generating a sequence of integers, typically used in for loops.
- It has three parameters:
  - start – the first element of the sequence
  - stop – next to the last element of the sequence
  - step – distance between to consecutive elements
- When not passed:
  - start defaults to 0
  - step defaults to 1
- range() function is very efficient, because it doesn't generate the entire sequence at once. More on that later.

```
for i in range(4):
    print(i)
for i in range(0,8,2):
    print(i)
for i in range(20,14,-2):
    print(i)
```

```
0
1
2
3
0
2
4
6
20
18
16
```

Break, continue and pass

- There are four statements provided for manipulating loop structures. These are break, continue, pass, and else.
- **break**: terminates the current loop.
- **continue**: immediately begin the next iteration of the loop.
- **pass**: do nothing. Use when a statement is required syntactically.
- **else**: represents a set of statements that should execute when a loop terminates.

```
for number in range(10,20):
    if number%2 == 0:
        continue
    for i in range(3,number):
        if number%i == 0:
            break
    else:
        print(number)
        print(' is prime')
```

```
11 is prime
13 is prime
17 is prime
19 is prime
```

Functions

- Basic definitions
- Default argument values
- Positional and keyword arguments
- Variadic arguments
- Decorators will be covered in Session 2
- Function annotations

Function's definition

- A function is created with **<u>def</u>** keyword, which is followed by the function name with round brackets enclosing the arguments and a colon. The indented statements form a body of the function.
- The return keyword is used to specify a list of values to be returned.
- All parameters in the Python language are passed by reference.
- However, only mutable objects can be changed in the called function.

```
def function_name(args):
    statements
```

```
# Defining the function
def f(parameter):
    print(parameter)
    return

def pair(a, b):
    return a, b

# Calling the function
f(3)
3
f('test')
test
a, b = pair(3, 4)
print(a, b)
3 4
```

Default argument values

- Default values can be provided for any number of arguments in a function
- Allows functions to be called with a variable number of arguments
- Arguments with default values must appear at the end of the arguments list
- Python's default arguments are evaluated *once* when the function is defined, not every time the function is called. Changes to a mutable default argument will be reflected in future calls to the function.

```
def f2(default = 2):
    print (default)
    return
f2()
2
f2(5)
5

test = list(range(5))
def f3(par = test):
    print(par)
    return
f3()
[0, 1, 2, 3, 4]
test.append(5)
f3()
[0, 1, 2, 3, 4, 5]
```

Keyword arguments

- When the formal parameter is specified, this is known as a *keyword argument.*
- By using keyword arguments, we can explicitly tell Python to which formal parameter the argument should be bound. Keyword arguments are always of the form *kwarg = value*.
- If keyword arguments are used they must follow any positional arguments, although the relative order of keyword arguments is unimportant.

```
def f4(p_1,\
       p_2,\
       k_1 = 'k_1',\
       k_2 = 'k_2'):
    print(p_1, p_2, k_1, k_2)
    return

f4('p_1', p_2='test')
p_1 test k_1 k_2
f4(p_2='test', p_1 = 'test2')
test2 test k_1 k_2
```

Variadic argument values

- Parameters of the form *param contain a variable number of arguments within a tuple
- Parameters of the form **param contain a variable number of keyword arguments.
- Within the function, we can treat args as a list of the positional arguments provided and kwargs as a dictionary of keyword arguments provided

```
def function5(*args, **kwargs):
    print("Variadic args:")
    for item in args:
        print(item)
    print("Variadic k_args:")
    for item in kwargs:
        print(item,':',\
              kwargs[item])
    return

function5('test',\
          k_arg = 'key_1')
Variadic args:
test
Variadic k_args:
k_arg : key_1
```

Packing and unpacking arguments

- Starred arguments syntax is used to pack and unpack arbitrary number of arguments
- When *args is used in function definition arguments passed to the function will be packed in a tuple
- When *args is used in function call arguments will be unpacked
- Similar for **kwargs packing and unpacking is performed for keyword arguments

```
def f4(p_1,\
       p_2,\
       k_1 = 'k_1',\
       k_2 = 'k_2'):
    print(p_1, p_2, k_1, k_2)

def p(*args):
    args = list(args)
    args[0] = 'Changed'
    f4(*args)

def k(*args,**kwargs):
    kwargs['k_2'] = 'new'
    f4(*args,**kwargs)

p('a','b','c','d')
Changed b c d
k('a','b',k_1='k_1')
a b k_1 new
```

Function annotations

- Python supports special syntax for annotation of function's arguments and returned value (see PEP3107)
- Can be used by third-party libraries for extracting additional information about function
- In Python 3.6 syntax for variable annotations was added (see PEP526)

```
def a_f(a:int,\
        k:str = 'k'\
       )->None:
    print(a, k)
    return

a_f.__annotations__
{'a': int, 'k': str, 'return':
None}
```

Built-in functions

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Lambda functions

- One can also define lambda functions within Python.
  - Use the keyword *lambda* instead of *def.*
  - Can be used wherever function objects are used.
  - Restricted to one expression.
  - Typically used with functional programming tools

```
>>> def f(x):
...     return x**2
...
>>> print f(8)
64
>>> g = lambda x: x**2
>>> print g(8)
64
```

List comprehensions

- List comprehensions provide a nice way to construct lists where the items are the result of some operation.
- The simplest form of a list comprehension is

```
[expr for x in sequence]
```

- Any number of additional for and/or if statements can follow the initial for statement. A simple example of creating a list of squares:

```
>>> squares = [x**2 for x in range(0,11)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

List Comprehensions

- Here’s a more complicated example which creates a list of tuples.

```
>>> squares = [(x, x**2, x**3) for x in range(0,9) if x % 2 ==
0]
>>> squares
[(0, 0, 0), (2, 4, 8), (4, 16, 64), (6, 36, 216), (8, 64, 512)]
```

- The initial expression in the list comprehension can be anything, even another list comprehension.

```
>>> [[x*y for x in range(1,5)] for y in range(1,5)]
[[1, 2, 3, 4], [2, 4, 6, 8], [3, 6, 9, 12], [4, 8,
12, 16]]
```

Modules

- A module is a file containing Python definitions and statements.
- The file name is the module name with the suffix .py appended.
- Within a module, the module's name (as a string) is available as the value of the global variable *__name__*.
- If a module is executed directly however, the value of the global variable *__name__* will be "__main__".
- Modules can contain executable statements aside from definitions. These are executed only the *first* time the module name is encountered in an import statement as well as if the file is executed as a script.

The import statement

- The **import** statement is used to search for Python modules and initialize them
- Submodules can be loaded separately
- Aliasing of modules is supported

```
import numpy as np
import scipy as sp
import pandas as pd
from pandas import Series
from pandas import DataFrame
```

Module search path

- When a module is imported, Python does not know where it is located so it will look for the module in the following places, in order:
- Built-in modules.
- The directories listed in the sys.path variable. The sys.path variable is initialized from these locations:
  - The current directory.
  - PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).
  - The installation-dependent default.
- The sys.path variable can be modified by a Python program to point elsewhere at any time.
- At this point, we'll turn our attention back to Python functions. We will cover advanced module topics as they become relevant.

Module search path

- The sys.path variable is available as a member of the sys module. Here is the example output when I echo my own sys.path variable.

```
import sys
sys.path
['',
 'C:\\Program Files\\Anaconda3\\python35.zip',
 'C:\\Program Files\\Anaconda3\\DLLs',
 'C:\\Program Files\\Anaconda3\\lib',
...
```

PyPi – Python package index

- Universal repository for Python packages: https://**pypi**.python.org/
- Almost 100K different packages for all need and purposes
- Easy install with pip:

```
$ pip install package
```

- Today we will be using iPython and Jupyter Notebooks:

```
$ pip install ipython
$ pip install jupyter
```

Interactive Python

- IPython - an enhanced Python shell designed to accelerate the writing, testing, and debugging of Python code
- Additional features
  - IDE-like features (e.g. tab-completion, introspection and etc.)
  - Debugging and profiling tools
  - Integrated interaction with operating system
  - Integration with visualization tools (e.g. Matplotlib)
- iPython (Jupyter) HTML notebooks – integrate iPython with a web browser to create interactive reports and presentations

iPython - introspection

- Using a question mark (?) before or after a variable will display some general information about the object. If object is a function or instance method, then docstring, if defined, will be shown
- Double question mark (??) can be used to display the source code
- Question mark (?) can also be used to introspect the iPython namespace using wildcards similar to Windows and Unix command line

iPython - introspection

```
import numpy as np
np.*range?

Docstring:
arange([start,] stop[, step,], dtype=None)

Return evenly spaced values within a given interval.
...

np.arange??
Type: builtin_function_or_method
```

iPython - introspection

```
def sum(a,b):
    return a + b
sum??

Signature: sum(a, b)
Source:
def sum(a,b):
    return a + b
File:      c:\users\mikle.shupletsov\dropbox\test\<ipython-
input-16-15968be02bca>
Type:      function
```

Magic commands

- iPython has many special commands, known as "magic" commands, which are designed to faciliate common tasks and enable you to easily control the behavior of the iPython system.
- A magic command is any command prefixed by the percent symbol %.
- Single percent (%) commands are line magic commands and double percent (%%) are cell magic commands

iPython – magic commands

```
#Information about about iPython magic command subsystem
%magic

#Enables use of magic commands without % symbol
#Not encouraged, when used with jupyter notebooks
%automagic on

#Lists all magic commands
%lsmagic
Available line magics:
%alias  %alias_magic  %autocall  %automagic ...
...

Available cell magics:
%%!  %%HTML  %%SVG  %%bash  %%capture ...
...
```

iPython – common magic commands

Command	Description
`%run script args`	Run a Python script
`%time, %timeit`	Measures execution time of Python expression
`%prun, %run -p`	Profiles Python script or expression
`%lprun`	Line-by-line profiling
`%debug`	Invokes “post-mortem” debugger

- These are just the most common magic commands. You are encouraged to learn the rest on your own.

iPython – interaction with OS

Command	Description
`!cmd`	Execute cmd in the system shell
`output = !cmd args`	Run cmd and store the stdout in output
`%alias alias_name cmd`	Define an alias for a system (shell) command
`%bookmark`	Utilize IPython's directory bookmarking system
`%cd directory`	Change system working directory to passed directory
`%pwd`	Return the current system working directory
`%pushd directory`	Place current directory on stack and change to target directory
`%popd`	Change to directory popped off the top of the stack
`%dirs`	Return a list containing the current directory stack
`%dhist`	Print the history of visited directories
`%env`	Return the system environment variables as a dict

Jupyter Notebook

- Jupyter Notebook integrates iPython interactive Python environment with a web browser
- You can invoke it by running:

```
$ jupyter notebook
```

- The rest of the time will be devoted to getting hands-on experience with iPython and Jupyter Notebook