ITCS 4101/5356
Intro to NLP & Intro to ML

Python for Programming, ML, and NLP

Razvan C. Bunescu

Department of Computer Science @ CCI

razvan.bunescu@uncc.edu

Python Programming Stack

- Python = object-oriented, interpreted, scripting language.
  - imperative programming, with functional programming features.

- NumPy = package for powerful N-dimensional arrays:
  - sophisticated (broadcasting) functions.
  - useful linear algebra, Fourier transform, and random number capabilities.

- SciPy = package for numerical integration and optimization.

- Matplotlib = comprehensive 2D and 3D plotting library.

Python Programming Stack

- PyTorch = a wrapper of NumPy that enables the use of GPUs and automatic differentiation:
  - **Tensors** similar to NumPy's ndarray, but can also be used on GPU.

- Jupyter Notebook = a web app for creating documents that contain live code, equations, visualizations and markdown text.

- Anaconda = an open-source distribution of Python and Python packages:
  - Package versions are managed through Conda.
  - Install all packages above using Anaconda / Conda install.

Anaconda Install

- **Anaconda**: Installation instructions for various platforms can be found at: https://docs.anaconda.com/anaconda/install
  - For Mac and Linux users, the system PATH must be updated after installation so that ‘conda’ can be used from the command line.
    - Mac OS X:
      - For bash users: export PATH=~/opt/anaconda3/bin:$PATH
      - For csh/tcsh users: setenv PATH ~/opt/anaconda3/bin:$PATH
    - For Linux:
      - For bash users: export PATH=~/opt/anaconda3/bin:$PATH
      - For csh/tcsh users: setenv PATH ~/opt/anaconda3/bin:$PATH
  - It is recommend the above statement be put in the ~/.bashrc or ~/.cshrc file, so that it is executed every time a new terminal window is open.
  - To check that conda was installed, running “*conda list*” in the terminal should list all packages that come with Anaconda.

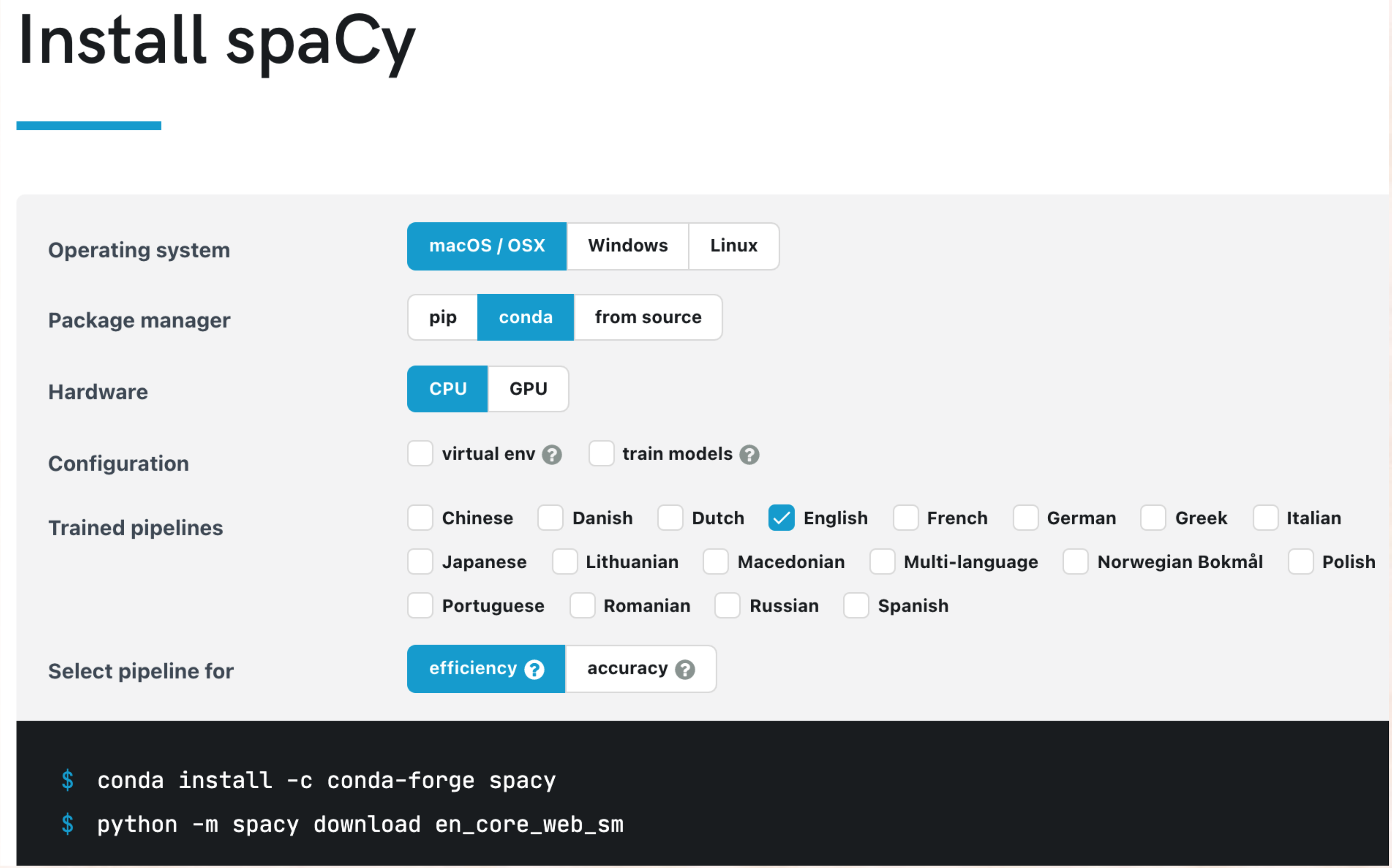
Installing Packages with Conda / Anaconda

- Tools and libraries that can be configured from Anaconda:
  - Python 3, **NumPy**, SciPy, Matplotlib, **Jupyter Notebook**, Ipython, Pandas, **Scikit-learn**.
  - **PyTorch** can be installed with ‘conda’ from the command line:
    - The actual command line depends on the platform as follows:
      - Using the GUI on pytorch.org, choose the appropriate OS, conda, Python 3.6, CUDA or CPU version.

NLP/ML APIs in Python

- spaCy: https://spacy.io/usage
- Scikit-learn: https://scikit-learn.org/stable/
- Hugging Face: https://huggingface.co/docs
- OpenAI: https://platform.openai.com/docs/introduction

spaCy



Install spaCy

Operating system: [✓ macOS / OSX] Windows Linux

Package manager: pip [✓ conda] from source

Hardware: [✓ CPU] GPU

Configuration: □ virtual env ? □ train models ?

Trained pipelines: □ Chinese □ Danish □ Dutch ☑ English □ French □ German □ Greek □ Italian □ Japanese □ Lithuanian □ Macedonian □ Multi-language □ Norwegian Bokmål □ Polish □ Portuguese □ Romanian □ Russian □ Spanish

Select pipeline for: [✓ efficiency ?] accuracy ?

```
$ conda install -c conda-forge spacy
$ python -m spacy download en_core_web_sm
```

Python

- Designed by Guido van Rossum in the early 1990s.
- Current development done by the Python Software Foundation.

- Python facilitates multiple programming paradigms:
  - imperative programming.
  - object oriented programming.
  - functional programming.

⇒ **multi-paradigm programming language**.

Python: Important Features

- Python = object-oriented interpreted “scripting” language:
  - **Object oriented**:
    - modules, classes, exceptions.
    - dynamically typed, automatic garbage collection.
  - **Interpreted**, interactive:
    - rapid edit-test-debug cycle.
  - **Extensible**:
    - can add new functionality by writing modules in C/C++.
  - **Standard library**:
    - extensive, with hundreds of modules for various services such as regular expressions, TCP/IP sessions, etc.

Scripting Languages

- **Scripts** vs. **Programs**:
  - – interpreted vs. compiled
  - – one script = a program
  - – many {*.c, *.h} files = a program

- Higher-level **“glue” language**:
  - – glue together larger program/library components, potentially written in different programming languages.
    - orchestrate larger-grained computations.
    - vs. programming fine-grained computations.
  - ➢ grep –i programming *.txt | grep –i python | wc -l

The Python Interpreter: Terminal

```
(base) rbunescu@CCI5CFQ05NALT code % python
Python 3.10.9 (main, Mar  1 2023, 12:33:47) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> □
```

```
(base) rbunescu@CCI5CFQ05NALT code % ipython
Python 3.10.9 (main, Mar  1 2023, 12:33:47) [Clang 14.0.6 ]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.10.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: □
```

The Python Interpreter: Help()

```
>>> help()

Welcome to Python 3.10's help utility!

If this is your first time using Python, you should definitely check out the tutorial on the
Internet at https://docs.python.org/3.10/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing Python programs and
using Python modules.  To quit this help utility and return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type "modules", "keywords",
"symbols", or "topics".  Each module also comes with a one-line summary of what it does; to
list the modules whose name or summary contain a given string such as "spam", type
"modules spam".

help>
```

The Python Interpreter: Keywords

```
help> keywords

    Here is a list of the Python keywords.  Enter any keyword to get more help.

    False          def            if             raise
    None           del            import         return
    True           elif           in              try
    and            else           is             while
    as             except         lambda         with
    assert         finally        nonlocal       yield
    break          for            not
    class          from           or
    continue       global         pass
```

The Python Interpreter: Keywords

```
help> lambda

Lambdas
*******

   lambda_expr        ::= "lambda" [parameter_list]: expression
   lambda_expr_nocond ::= "lambda" [parameter_list]: expression_nocond

Lambda expressions (sometimes called lambda forms) have the same syntactic position as
expressions.  They are a shorthand to create anonymous functions; the expression "lambda
arguments: expression” yields a function object.  The unnamed object behaves like a function
object defined with

   def <lambda>(arguments):
       return expression

See section *Function definitions* for the syntax of parameter lists. Note that functions
created with lambda expressions cannot contain statements or annotations.
```

The Python Interpreter: Modules

help> modules

commands:

execute shell commands vio os.popen() and return status, output.

compiler:

package for parsing and compiling Python source code.

gzip:

functions that read and write gzipped files.

HTMLParser:

a parser for HTML and XHTML (defines a class HTMLParser).

math:

access to the mathematical functions defined by the C standard.

exceptions:

Python’s standard exception class hierarchy.

The Python Interpreter: Modules

help> modules

os:

OS routines for Mac, NT, or Posix depending on what system we’re on.

re:

support for regular expressions (RE).

string:

a collection of string operations (most are no longer used).

sys:

access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter:

sys.argv: command line arguments.

sys.stdin, *sys.stdout*, *sys.stderr*: standard input, output, error file objects.

threading:

thread modules emulating a subset of Java’s threading model.

The Python Interpreter: Integer Precision

```
>>> def  fib(n):
. . .      a, b, i = 0, 1, 0
. . .      while i < n:
. . .         a,b,i = b, a+b, i+1
. . .      return a
. .
>>> fib(10)
55
>>> fib(100)
354224848179261915075
>>> fib(1000)
43466557686937456435688527675040625802564660517371780402481729089536555541
4905189040387984007925516929592259308032263477520968962323987332247116164
996440906533187938298969649928516003704476137795166849228875
```

Built-in Types: Basic

- **integers**, with unlimited precision – int().
  - decimal, octal and hex literals.
- **floating point numbers**, implemented using double in C – float().
- **complex numbers**, real and imaginary as double in C – complex().
  - 10+5j, 1j
  - z.real, z.imag
- **boolean values**, True and False – bool().
  - False is: None, False, 0, 0L, 0.0, 0j, ‘’, (), [], {},
  - user defined class defines methods nonzero() or len().
- **strings –** str(), **class**, **function**, …
  - “Hello world”, ‘Hello world’

Built-in Types: Composite

- **lists**:
  - [], [1, 1, 2, 3], [1, “hello”, 2+5j]
- **tuples**:
  - (), (1,), (1, 1, 2, 3), (1, “hello, 2+5j)
- **dictionaries**:
  - {“john”: 12, “elaine”: 24}
- **sets**:
  - {1, 2, 3}
- **files**

Integers

```
>>> int
<class ‘int’>
>>> 1024
1024
>>> int(1024)
1024
>>> repr(1024)
‘1024’
>>> eval(‘1024’)
1024
>>> str(1111)
‘1111’
>>> int(‘1111’)
1111
```

```
>>> a = 100
>>> b = 200
>>> a + 1, b + 1  #this is a tuple
(101, 201)
>>> print(a, b + 1)
100 201
>>> int(3.6), abs(-10), 11%3, 11//3, 11/3, 2**3
(3, 10, 2, 3, 3.6666666666666665, 8)
>>> int(‘1110’,2), int(‘170’,8), int(‘40’,16)
(14, 120, 64)
>>>  [170, 0170, 0x40] #this is a list
[170, 120, 64]
>>> float(8), 2**3, pow(2,3)
(8.0, 8, 8)
```

Booleans

```
>>> bool
<class ‘bool’>
>>> [bool(0), bool(0.0), bool(0j),bool([]), bool(()), bool({}), bool(‘’), bool(None)]
[False, False, False, False, False, False, False, False]
>>> [bool(1), bool(1.0), bool(1j), bool([1]), bool((1,)), bool({1:’one’}), bool(‘1’)]
[True, True, True, True, True, True, True]
>>> str(True), repr(True)
(‘True’, ‘True’)
>>> True and True, True and False, False and False
(True, False, False)
>>> True or True, True or False, False or False
(True, True, False)
>>> not True, not False
(False, True)
```

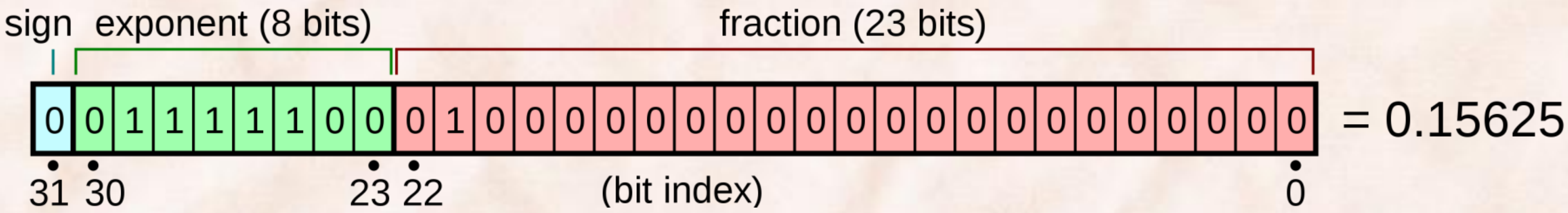
Floating Point

```
>>> float
<class ‘float’>
>>> 3.14, 3.1400000000000001
(3.14, 3.14)
>>> repr(3.1400000000000001)
3.14
>>> 3.14/2, 3.14//2
(1.5, 1.0)
>>> 1.99999999999999999999
2.0
>>> import math
>>> math.pi, math.e
(3.141592653589793l, 2.718281828459045l)
>>> help(‘math’)
```

```
=========  Python  ==========
>>> sum = 0.0
>>> for i in range(10):
. . .       sum += 0.1
. . .
>>> sum
0.99999999999999989
=========== C++ ===========
float sum = 0.0;
for (int i = 0; i < 10; i++)
      sum += 0.1;
cout.precision(17);
cout << sum << endl;
⇒ 1.0000001192092896
```

http://docs.python.org/3/tutorial/introduction.html#numbers

IEEE 754 Standard



- Use **decimal.Decimal** to represent 0.1 exactly (fixed point representation).

```
>>> from decimal import Decimal
>>> zpo = Decimal(‘0.1’)
          vs.
>>> zpo = Decimal(0.1)
```

Strings (Immutable)

- Immutable Sequences of Unicode Characters:

```
>>> str
<class ‘str’>
>>> str.upper(‘it’), ‘it’.upper()
(‘IT’, ‘IT’)
>>> ‘ ’, “”
(‘’,’’)
>>> “functional {0} lang”.format(‘programming’)
‘functional programming lang
>>> “object oriented “ + “programming”
‘object oriented programming’
>>> ‘orient’ in ‘object oriented’, ‘orient’ in ‘Object Oriented’
(True, False)
```

```
>>> s = “object oriented”
>>> len(s), s.find(‘ct’), s.split()
(15, 4, [‘object’, ‘oriented’])
>>> s[0], s[1], s[4:6], s[7:]
(‘o’, ‘b’, ‘ct’, ‘oriented’)
>>> s[7:100]
‘oriented’
>>> help(‘str’)
```

http://docs.python.org/3/tutorial/introduction.html#strings

(Fancy) String Formatting

```
>>> n, x, s = 10, math.pi, 'charlotte'
>>> 'We live in {city}.'.format(city = s)
>>> 'We live in {}.".format(s)'
>>> 'There are {num} bears living in {city}.'.format(city = s, num = n)
>>> 'There are {1} bears living in {0}.'.format(s, n)
>>> 'The approximate value of pi is {:.2f}.'.format(math.pi)
>>> 'The approximate value of pi is {math.pi:.2f}.'
>>> 'The approximate value of pi is {math.pi:.2f}.'
>>> 'The value of pi is approximately {x:.2f}.'
>>>           vs.
>>> f'The approximate value of pi is {x:.2f}.’
```

Read more here:

https://docs.python.org/3/tutorial/inputoutput.html

List (Mutable)

```
>>> []    #empty list
[]
>>> x = [3.14, “hello”, True]
[3.14, ‘hello’, True]
>>> x + [10, [], len(x)]
[3.14, ‘hello’, True, 10, [], 3]
>>> x.append(0.0)
>>> x.reverse()
>>> x
[0.0, True, ‘hello’, 3.14]
>>> x * 2
[0.0, True, ‘hello’, 3.14, 0.0, True, ‘hello’, 3.14]
```

```
>>> x.sort()
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < bool()
            >>> x = [0.0, 3.14, True]
            >>> sorted(x)
            [0.0, True, 3.14]
            >>> x
            [0.0, 3.14, True]
            >>> x.sort()
            >>> x
            [0.0, True, 3.14]
```

```
>>> help(‘list’)
```

Tuple (Immutable)

```
>>> tuple
<class ‘tuple’>
>>> ()    # empty tuple
()
>>> (10)   # not a one element tuple!
10
>>> (10,) # a one element tuple
(10,)
>>> (1, 2) + (3, 4)
(1, 2, 3, 4)
>>> (1, 2) * 2
(1, 2, 1, 2)
>>> x = (0, 1, ‘x’, ‘y’)
```

```
>>> x[0], x[1], x[:-1]
(0, 1, (0, 1, ‘x’))
>>> y = (13, 20, -13, -5, 0)
>>> temp = list(y)
>>> temp.sort()
>>> y = tuple(temp)
>>> y
 (-13, -5, 0, 13, 20)
>>> help(‘tuple’)
```

Immutable types are hashable!

Set (Mutable) & Frozenset (Immutable)

```
>>> set, frozenset
(<class ‘set’>, <class ‘frozenset’>)
>>>  set()   # empty set
set()
>>> type(set())
<class ‘set’>
>>> {}   # not an empty set!
{}
>>> type({})
<class ‘dict’>
>>> s = {1, 3, 3, 2}
>>> s
{1, 2, 3}
>>> frozenset({1, 3, 3, 2})
frozenset({1, 2, 3})
```

```
>>> 3 in s, 4 in s
(True, False)
>>> s = set(‘yellow’)
>>> t = set(‘hello’)
>>> s
{‘e’, ‘o’, ‘l’, ‘w’, ‘y’}
>>> t
{‘h’, ‘e’, l’, ‘o’}
>>> s – t,    s | t
({‘w’, ‘y’},    {‘h’, ‘e’, ‘o’, ‘l’, ‘w’, ‘y’})
>>> s & t, s^ t
({‘e’, l’, ‘o’}, {‘h’, ‘w’, ‘y’}
>>> {1, 3} <= {1, 2, 3, 2}
True
>>> help(set) >>>help(frozenset)
```

Mutable Operations on Sets

```
>>> s = set([‘abba’, ‘dada’, ‘lola’, ‘bama’])
>>> s
{‘abba’, ‘bama’, ‘dada’, ‘lola’}
>>> s |= {‘bama’, ‘lily’}
>>> s
{‘abba’, ‘bama’, ‘dada’, ‘lily’, ‘lola’}
>>> s −= {‘lola’, ‘abba’, ‘mama’}
>>> s
{‘bama’, ‘dada’, ‘lily’}
>>> s &= {‘lily’, ‘dodo’, ‘bama’}
>>> s
{‘bama’, ‘lily’}
```

```
>>> s = {[1]}
TypeError: unhashable type:  ‘list’
```

How can we prove the actual set object changes and not a new one is created?

Hint: are s −= t and s = s − t equivalent for sets? How about strings?

Dictionaries (Mutable)

```
>>> dict
<class ‘dict’>
>>> {}    # empty dictionary
{}
>>> d = {‘john’:23, ‘alex’:25, ‘bill’:99}
>>> d[‘alex’]
25
>>> d[‘alex’] = 26
>>> del d[‘john’]
>>> d[‘tammy’] = 35
>>> d
{‘alex’:26, ‘bill’:99, ‘tammy’:35}
>>>  for key in d:
>>>   … print(key, end = ‘ ‘)
‘alex’ ‘bill’ ‘tammy’
```

```
>>>  d.items()    # this is a view
dict_items[(‘alex’,26), ‘(bill’,99), (‘tammy’,35)]
>>> d.keys() )    # this is a view
dict_keys[‘alex’, ‘bill’, ‘tammy’]
>>> d.values()    # this is a view
dict_values[26, 99, 35]
>>> for x, y in d.items():
>>> … print (x, y)
‘alex’ 26
‘bill’ 99
‘tammy’ 35
>>> d.keys() | [‘alex’, ‘john’]    # set ops.
{‘alex’, ‘bill’, ‘john’, ‘tammy’}
>>> d[‘mary’] = 10
>>> d.keys()
dict_keys[‘alex’, ‘bill’, ‘tammy’, ‘mary’]
```

Dictionaries (Mutable)

```
>>> dict
<class ‘dict’>
>>> {}
{}
>>> d =
>>> d[‘a
25
>>> d[‘a
>>> del d
>>> d[‘ta
>>> d
{‘alex’:2
>>>  for
>>>  … print(key, end = ‘ ‘)
‘alex’ ‘bill’ ‘tammy’
```

```
>>>  d.items()    # this is a view
dict_items[(‘alex’,26), ‘(bill’,99), (‘tammy’,35)]
```

Immutable types are hashable, mutable types are not:

```
>>> d = dict(abba=1, dada=2)
>>> d[frozenset({1, 2, 3})] = 3
>>> d
{‘abba’:1, ‘dada’:2, frozenset({1, 2, 3}):3}

>>> d[{1, 2, 3, 4}] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set’
```

```
>>> d[‘mary’] = 10
>>> d.keys()
dict_keys[‘alex’, ‘bill’, ‘tammy’, ‘mary’]
```

Files

```
>>> file
<type ‘file’>
>>> output = open(‘/tmp/output.txt’, ‘w’)    # open tmp file for writing
>>> input = open(‘/etc/passwd’, ‘r’)         # open Unix passwd file for reading
>>> s = input.read()                         # read entire file into string s
>>> line = input.readline()                  # read next line
>>> lines = input.readlines()                # read entire file into list of line strings
>>> output.write(s)                          # write string S into output file
>>> output.write(lines)                      # write all lines in ‘lines’
>>> output.close()
>>> input.close()
```

```
>>> from urllib import urlopen
>>> html = urlopen(‘http://www.ohio.edu’)
>>> lines = [line[:-1] for line in html.readlines()]
```

```
>>> help(‘file’)
```

Statements & Functions

- Assignment Statements
- Compound Statements
- Control Flow:
  - Conditionals
  - Loops
- Functions:
  - Defining Functions
  - Lambda Expressions
  - Documentation Strings
  - Generator Functions

Assignment Forms

- Basic form:
  - x = 1
  - y = ‘John’
- Tuple positional assignment:
  - x, y = 1, ‘John’
  - x == 1, b == ‘John’ => (True, True)
- List positional assignment:
  - [x, y] = [1, ‘John]
- Multiple assignment:
  - x = y = 10

Compound Statements

- Python does not use block markers (e.g. ‘begin .. end’ or ‘{ … }’) to denote a compound statements.
  - Need to indent statements at the same level in order to place them in the same block.
  - Can be annoying, until you get used to it; intelligent editors can help.
  - Example:

```
if n == 0:
   return 1
else:
   return  n * fact(n - 1)
```

Conditional Statements

```
if <bool_expr_1>:
   <block_1>
elif <bool_expr_2>:
   <block_2>
…
else:
   <block_n>
```

- There can be zero or more elif branches.
- The else branch is optional.
- “Elif “ is short for “else if” and helps in reducing indentation.

Else If vs. Elif

```
if a == 0:
  print('nothing')
else:
  if a == 1:
    print('one thing’)
  else:
    if a == 2:
      print('two things’)
    else:
      if a == 3:
        print('three things’)
      else:
        print('many things')
```

```
if a == 0:
  print('nothing')
elif a == 1:
  print('one thing’)
elif a == 2:
  print('two things’)
elif a == 3:
  print('three things’)
else:
  print('many things')
```

Conditional Statements

- Python 3.10 added a `match … case` statement.
- Same behavior can be achieved using:
  - `if … elif … elif` sequences.
  - dictionaries:

```
name = ‘John’
dict = {‘Mary’:’female’, ‘John’: ‘male’,
  ‘Alex’:’either’, ‘Paris’:’either’}
if name in dict:
  print(dict[name])
else:
 print(‘unknown’)
```

While Loops

```
x = ‘university’
while x:
   print(x, end = ‘ ‘)
   x = x[1:]

a, b = 1, 1
while b <= 23:
   print(a, end = ‘ ‘)
   a, b = b, a + b
```

For Loops

```
sum = 0
for x in [1, 2, 3, 4]
   sum += x
print(sum)

D = {1:’a’, 2:’b’, 3:’c’, 4:’d’}
for x, y in D.items():
   print(x, y)
```

Names and Scopes

- Static scoping rules.
  - if ***x*** is a variable name that is only read, its variable is found in the closest enclosing scope that contains a defining write.
  - a variable ***x*** that is written is assumed to be local, unless it is explicitly imported.
  - use global and nonlocal keywords to override these rules.
- Example:

Functions

```
def mul(x,y):
   return x * y

mul(2, 5) => ?
mul(math.pi, 2.0) => ?
mul([1, 2, 3], 2) => ?
mul((‘a’, ‘b’), 3) => ?
mul(‘ou’, 5) => ?
```

Parameter Correspondence

```
def f(a, b, c): print(a, b, c)
f(1, 2, 3) => 1 2 3
f(b = 1, c = 2, a = 3) => 3 1 2

def f(*args): print(args)
f(“one argument”) => (‘one argument’)
f(1, 2, 3) => (1, 2, 3)

def f(**args): print args
f(a=2, b=4, c=8) => {‘a’:2, ‘b’:4, ‘c’:8}
```

Lambda Expressions

- Scheme:

```
>(define (make-adder (num)
   (lambda (x)
      (+ x num)))
```

- Python:

```
>>> def make_adder(num):
...     return lambda x: x + num
...
>>> f = make_adder(10)
>>> f(9)
19
```

Lambda Expressions

```
>>> formula = lambda x, y: x *x + x*y + y*y
>>> formula
<function <lambda> at 0x2b3f213ac230>
>>> apply(formula, (2,3))
19
>>> list(map(lambda x: 2*x, [1, 2, 3]))
[2, 4, 6]
>>> list(filter(lambda x: x>0, [1, -1, 2, -2, 3, 4, -3, -4]))
[1, 2, 3, 4]
>>> from functools import reduce
>>> reduce(lambda x,y:x*y, [1, 2, 3, 4, 5])
120
>>> def fact(n): return reduce (lambda x, y: x*y, range(1, n+1))
>>> fact(5)
120
```

Iterators

- An **iterator** is an object representing a stream of data:
  - to get the next element in the stream:
    - call __next__() method.
    - pass the iterator to the built-in function next().
  - to create an iterator:
    - call iter(*collection*).
    - some functions create iterators/iterables instead of collections:
      - map(), filter(), zip(), ...
      - range(), dict.keys(), dict.items(), ...
      - **why create iterators/iterables instead of collections?**
- Examples:

Iterators

```
for x in range(5):
   print(x)
```

an **iterable** (provides the __iter__() method)

Internally, this is implemented as:

```
it = iter(range(5))
while True:
  try:
      x = next(it)
      print(x)
  except StopIteration:
      break
```

A Convenient Shortcut to Building Iterators: Generator Functions/Expressions

```
def squares(n):
   for i in range(n):
      yield i*i

>>> for i in squares(5):
...    print(i, end = ‘ ’)
0 1 4 9 16

>>> s = squares(5)
>>> next(s) => 0
>>> next(s) => 1
>>> next(s) => 4
```

Equivalent **generator expression**:

>>> (i * i for i in range(n))

Generator Functions

```
def fib():  # generate Fibonacci series
  a, b = 0, 1
  while 1:
    yield b
    a, b = b, a+b

>>> it = fib()
>>> next(it) => 1
>>> next(it) => 1
>>> next(it) => 2
```

List/Set/Dict Comprehensions

- Mapping operations over sequences is a very common task in Python coding:
  - ⇒ introduce a new language feature called *list comprehensions.*

```
>>> [x**2 for x in range(5)]
[0, 1, 4, 9, 25]

>>> [x for x in range(5) if x % 2 == 0]
[0, 2, 4]

>>> [x+y for x in [1, 2, 3] for y in [-1,0,1]]
[0,1,2,1,2,3,2,3,4]

>>> [(x,y) for x in range(5) if x%2 == 0 for y in range(5) if y%2=1]
[(0,1), (0,3), (2,1), (2,3), (4,1), (4,3)]
```

List/Set/Dict Comprehensions

‘[‘*expression* for $target_1$ in $iterable_1$ [if *condition*]
for $target_2$ in $iterable_2$ [if *condition*]
…
for $target_n$ in $iterable_n$ [if *condition*] ‘]’

```
>>> [line[:-1] for line in open(‘myfile)]
>>> {x for x in ‘ohio’ if x not in ‘hawaii’}
>>> {x:2*x for x in ‘ohio}
>>> {x:y for x in [1, 2] for y in [3, 4]}
```

Errors & Exceptions

- Syntax errors:

```
while True print ‘True’
File “<stdin>”, line 1
   while True print ‘True’
                  ^
SyntaxError: invalid syntax
```

- Exceptions: errors detected during execution:

```
1 / 0
Traceback ( most recent call last):
  File “<stdin>”, line 1, in ?
ZeroDivisionError: integer division or modulo
  by zero
```

Handling Exceptions

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines’)
        f.close()
```

Modules

- Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter.
- 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__**

fibo.py

```
# Fibonacci numbers module
def fib(n):   # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

>>> import fibo
>>> fibo.fib(100)
1 1 2 3 5 8 13 21 34 55 89
```

How to write and run Python code

- Interactively:
  1. Terminal: run **python**, type code at the interpretor prompt >>>.
  2. Browser: run **jupyter-notebook**, type code and markdown.
     - Can File / Download as Python (.py) source code.

- Editor or IDE:
  1. **emacs**, **vi**, …
  2. **VSCode**, **Spyder**, **PyCharm**, …
     - Smart editors and IDE can also offer interactive features in parallel with code development.

How to write and run Python code

- Explicit interpretor invocation:

  > python main.py $\mathbf{arg_1}$ … $\mathbf{arg_n}$

- Implicit interpretor invocation:

  > ./main.py $\mathbf{arg_1}$ … $\mathbf{arg_n}$

  - Need to specify the path to the python interpretor on the first line of main.py:

    ```
    #!~/opt/anaconda3/bin/python
    ```

  - Need to give execution rights to main.py:

    ```
    > chmod 0755 main.py
    ```

Readings

- More about Python:
  - https://docs.python.org/3/tutorial
    - https://docs.python.org/3/tutorial/classes.html#iterators
    - https://docs.python.org/3/tutorial/classes.html#generators
  - https://docs.python.org/3/library/functions
  - https://docs.python.org/3/library/stdtypes
  - https://docs.python.org/3/whatsnew/3.0.html
  - https://docs.python.org/3/howto/regex.html

- Matplotlib and NumPy:
  - https://matplotlib.org/3.1.1/tutorials/index.html
  - https://docs.scipy.org/doc/numpy/user/quickstart.html