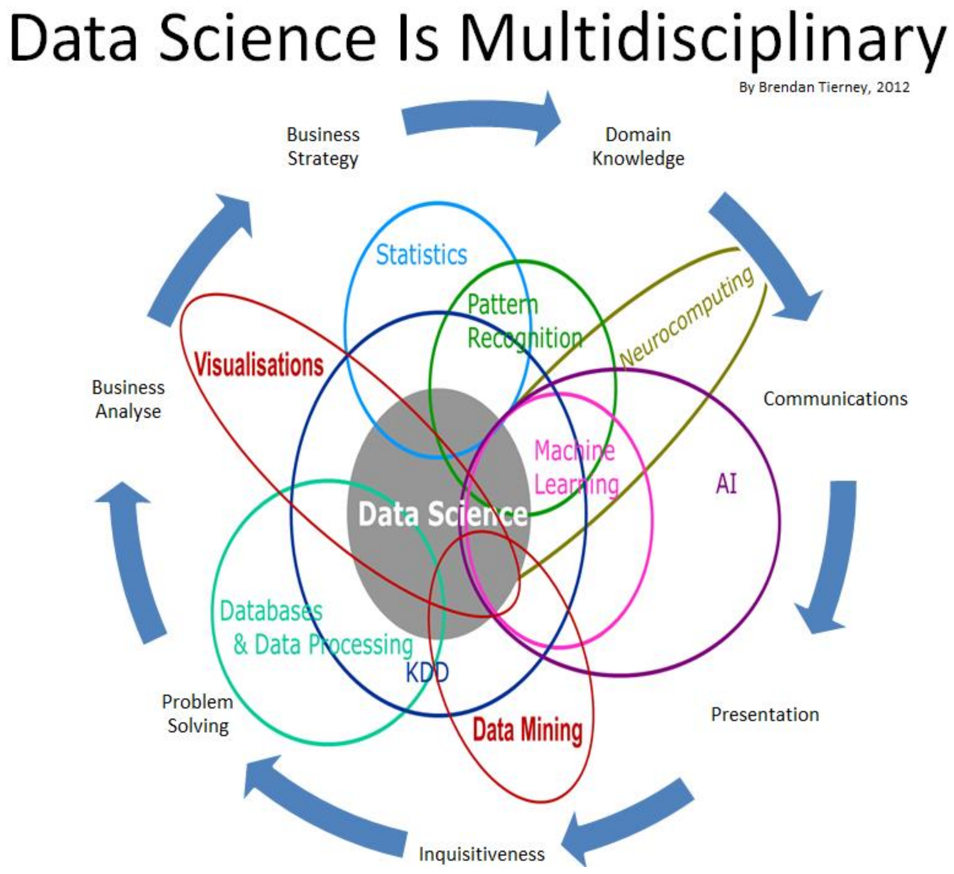
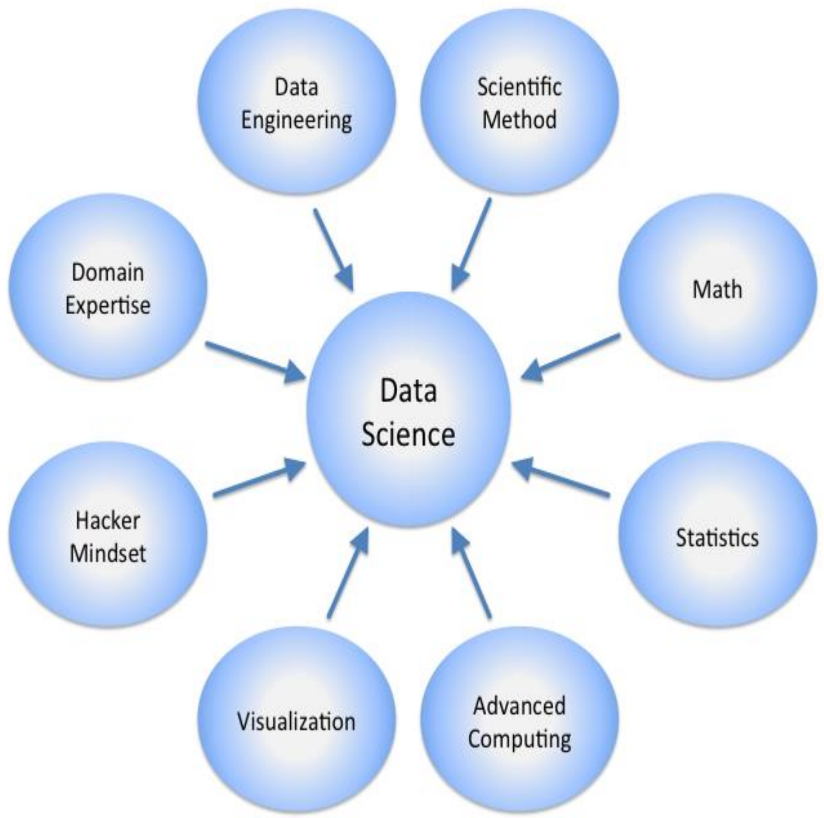
PYTHON PROGRAMMING 4

Anasua Sarkar
Computer Science & Engineering Department
Jadavpur University
anasua.sarkar@jadavpuruniversity.in

Data Science – A Definition

Data Science is the science which uses computer science, statistics and machine learning, visualization and human-computer interactions to collect, clean, integrate, analyze, visualize, interact with data to create data products.





Python : Introduction

- Most recent popular (scripting/extension) language
  - although origin ~1991
- heritage: teaching language (ABC)
  - Tcl: shell
  - perl: string (regex) processing
- object-oriented
- It includes modules for creating graphical user interfaces, connecting to relational databases, generating pseudorandom numbers, arithmetic with arbitrary-precision decimals, manipulating regular expressions, and unit testing.
- Large organizations that use Python include Wikipedia, Google, Yahoo!, CERN, NASA, Facebook, Amazo n, Instagram and Spotify. The social news networking site Reddit is written entirely in Python.

What's Python?

- Python is a general-purpose, interpreted high-level programming language.
- Its syntax is clear and emphasize readability.
- Python has a large and comprehensive standard library.
- Python supports multiple programming paradigms, primarily but not limited to object-oriented, imperative and, to a lesser extent, functional programming styles.
- It features a fully dynamic type system and automatic memory management

@Yang Li

Python structure

- modules: Python source files
  - import, top-level via from, reload
- statements
  - control flow
  - create objects
  - indentation matters – instead of {}
- objects
  - everything is an object
  - automatically reclaimed when no longer needed

Comprehensions

- https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Comprehensions.html
- https://realpython.com/list-comprehension-python/

Collections

- https://docs.python.org/3/library/collections.html

Lambda forms

- anonymous functions
- may not work in older versions

```
def make_incrementor(n):
    return lambda x: x + n

f = make_incrementor(42)
f(0)
f(1)
```

Functional programming tools

- `filter(`*`function, sequence`*`)`
  - filter does the work of a list-comprehension if

```
def f(x): return x%2 != 0 and x%3 == 0
filter(f, range(2,25))
```

```
def is_even(x):
    """True if x is even, False if x is odd"""
    return x % 2 == 0
```

```
x_evens = [x for x in xs if is_even(x)]    # [2, 4]
x_evens = filter(is_even, xs)              # same as above
list_evener = partial(filter, is_even)     # *function* that filters a list
x_evens = list_evener(xs)                  # again [2, 4]
```

- `map(`*`function, sequence)`*
  - call function for each item
  - return list of return values

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

products = map(multiply, [1, 2], [4, 5]) # [1 * 4, 2 * 5] = [4, 10]
```

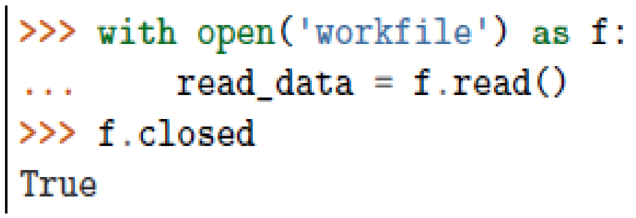
- `reduce(`*`function, sequence`*`)`
  - return a single value
  - call binary function on the first two items
  - then that result with the third and so on
  - iterate

```
x_product = reduce(multiply, xs)            # = 1 * 2 * 3 * 4 = 24
list_product = partial(reduce, multiply)    # *function* that reduces a list
x_product = list_product(xs)                # again = 24
```

File I/O

- open() returns a *file object*, and is most commonly used with two arguments: open(filename, mode).

```
f = file("foo", "r")
line = f.readline()
print line,
f.close()
# Can use sys.stdin as input;
# Can use sys.stdout as output.
```



```
>>> with open('workfile') as f:
...     read_data = f.read()
>>> f.closed
True
```

- *mode* can be 'r' when the file will only be read, 'w' for only writing (an existing file with the same name will be erased), and 'a' opens the file for appending; any data written to the file is automatically added to the end. 'r+' opens the file for both reading and writing.
- Normally, files are opened in *text mode*. 'b' appended to the mode opens the file in *binary mode.*

Files

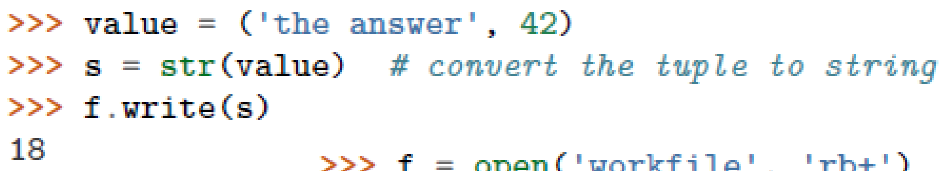
```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

```
list(f)
```

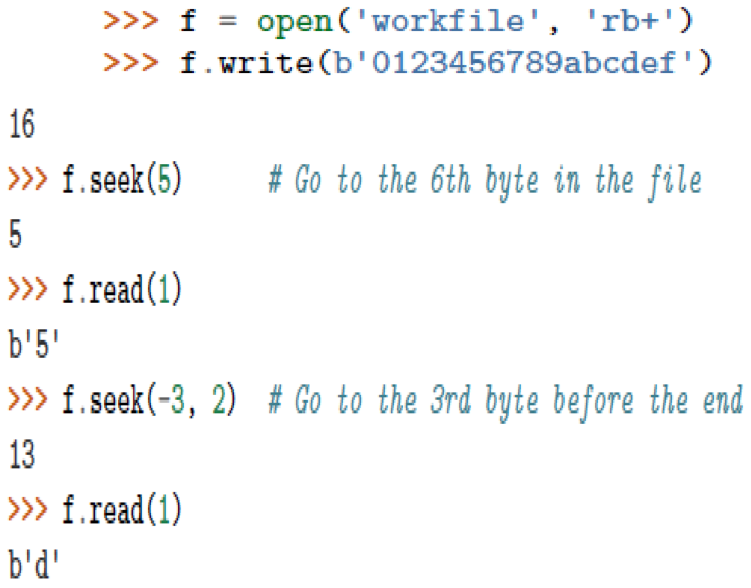
```
f.readlines()
```

- Creating file object
  - Syntax: file_object = open(filename, mode)
    - `Input = open(“d:\inventory.dat”, “r”)`
    - `Output = open(“d:\report.dat”, “w”)`
- Manual close
  - Syntax: close(file_object)
    - `close(input)`
- Reading an entire file
  - Syntax: string = file_object.read()
    - `content = input.read()`
- To read a file’s contents, call f.read(size), which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode).
  - Syntax: list_of_strings = file_object.readlines()
    - `lines = input.readline()`
    - It is only omitted on the last line of the file if the file doesn’t end in a newline.

Files



```
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to string
>>> f.write(s)
18
```



```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

- Reading one line at time
  - Syntax: list_of_strings = file_object.readline()
    - `line = input.readline()`
- Writing a string
  - Syntax: file_object.write(string)
- `output.write("Price is %(total)d" % vars())`
- Writing a list of strings
  - Syntax: file_object.writelines(list_of_string)
    - `output.write(price_list)`
- This is very simple!
  - Compare it with `java.io`
- f.tell() returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.
- To change the file object's position, use f.seek(offset, from_what).

Saving structured data with json

- JSON (JavaScript Object Notation)
- The standard module called json can take Python data hierarchies, and convert them to string representations; this process is called *serializing*.
- Reconstructing the data from the string representation is called *deserializing*.
- Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

- Another variant of the dumps() function, called dump(), simply serializes the object to a *text file*. So if f is a *text file* object opened for writing,

```
json.dump(x, f)
```

- To decode the object again, if f is a *text file* object which has been opened for reading

```
x = json.load(f)
```

Files: Input

input = open(‘data’, ‘r’)	Open the file for input
S = input.read()	Read whole file into one String
S = input.read(N)	Reads N bytes (N >= 1)
L = input.readlines()	Returns a list of line strings

Files: Output

output = open(‘data’, ‘w’)	Open the file for writing
output.write(S)	Writes the string S to file
output.writelines(L)	Writes each of the strings in list L to file
output.close()	Manual close

`open()` and `file()`

- These are identical:

```
f = open(filename, "r")
f = file(filename, "r")
```

- The `open()` version is older
- The `file()` version is the recommended way to open a file now
  - uses object constructor syntax (next lecture)

Collections

- https://docs.python.org/3/library/collections.html

Control flow: if

```
x = int(raw_input("Please enter #:"))
if x < 0:
  x = 0
  print 'Negative changed to zero'
elif x == 0:
  print 'Zero'
elif x == 1:
  print 'Single'
else:
  print 'More'
```

- no case statement

Control flow: for

```
a = ['cat', 'window', 'defenestrate']
for x in a:
  print x, len(x)
```

- no arithmetic progression, but
  - `range(10)` → `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`

  -
    ```
    for i in range(len(a)):
        print i, a[i]
    ```
- do not modify the sequence being iterated over

Loops: break, continue, else

- `break` and `continue` like C
- `else` after loop exhaustion

```
for n in range(2,10):
  for x in range(2,n):
    if n % x == 0:
      print n, 'equals', x, '*', n/x
      break
  else:
    # loop fell through without finding a factor
    print n, 'is prime'
```

Control flow (6)

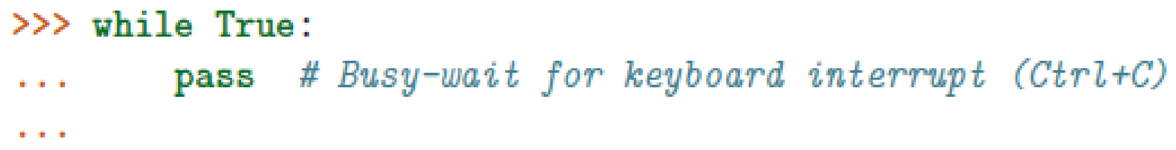
- Common **while** loop idiom:

```
f = open(filename, "r")
while True:
    line = f.readline()
    if not line:
        break
    # do something with line
```

Do nothing

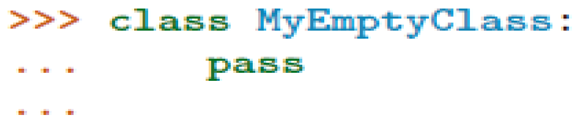
- pass does nothing
- syntactic filler

```
while 1:
        pass
```



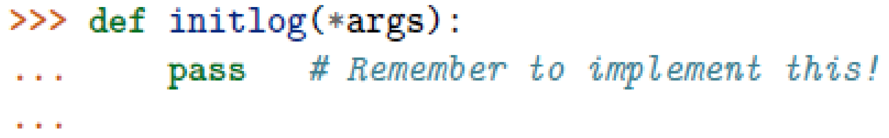
```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

- This is commonly used for creating minimal classes



```
>>> class MyEmptyClass:
...     pass
...
```

- Another place pass can be used is as a place-holder for a function or conditional body.



```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

Looping Techniques

- When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

- When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

- To loop over two or more sequences at the same time, the entries can be paired with the zip() function.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

More looping techniques

- To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the reversed() function.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

- To loop over a sequence in sorted order, use the sorted() function which returns a new sorted list while leaving the source unaltered.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

Defining functions

-
  ```
  def foo(x):
      y = 10 * x + 2
      return y
  ```
- Def keyword must be followed by the function name and the parenthesized list of formal parameters.
- All variables are local unless specified as `global`
- Arguments passed by value
- Arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object).
- When a function calls another function, a new local symbol table is created for that call.
- The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*.

Executing functions

```
def foo(x):
    y = 10 * x + 2
    return y
print foo(10)   # 102
```

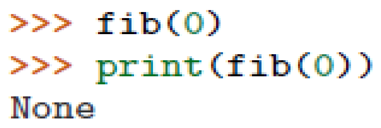
- The *execution* of a function introduces a new symbol table used for the local variables of the function.
- More precisely, all variable assignments in a function store the value in the local symbol table;
- whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names.
- Thus, global variables cannot be directly assigned a value within a function (unless named in a global statement), although they may be referenced.

Defining functions

```
def fib(n):
  """Print a Fibonacci series up to n."""
  a, b = 0, 1
  while b < n:
    print b,
    a, b = b, a+b

>>> fib(2000)
```

- First line is *docstring*
- first look for variables in local, then global
- need global to assign global variables



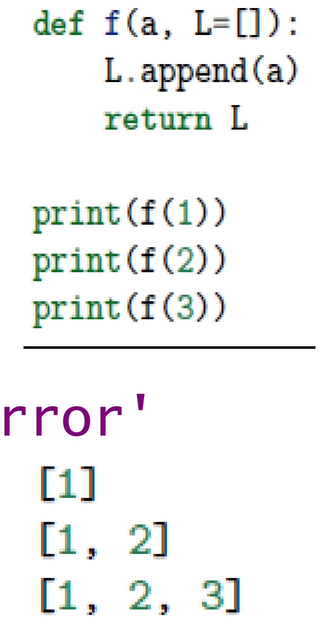
```
>>> fib(0)
>>> print(fib(0))
None
```

- In fact, even functions without a return statement do return a value, albeit a rather boring one. This value is called None (it’s a built-in name).

Functions: default argument values

```
def  ask_ok(prompt,  retries=4,  complaint='Yes  or  no,
 please!'):
  while 1:
    ok = raw_input(prompt)
    if ok in ('y', 'ye', 'yes'): return 1
    if ok in ('n', 'no'): return 0
    retries = retries - 1
    if retries < 0: raise IOError, 'refusenik error'
    print complaint

>>> ask_ok('Really?')
```



```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

```
[1]
[1, 2]
[1, 2, 3]
```

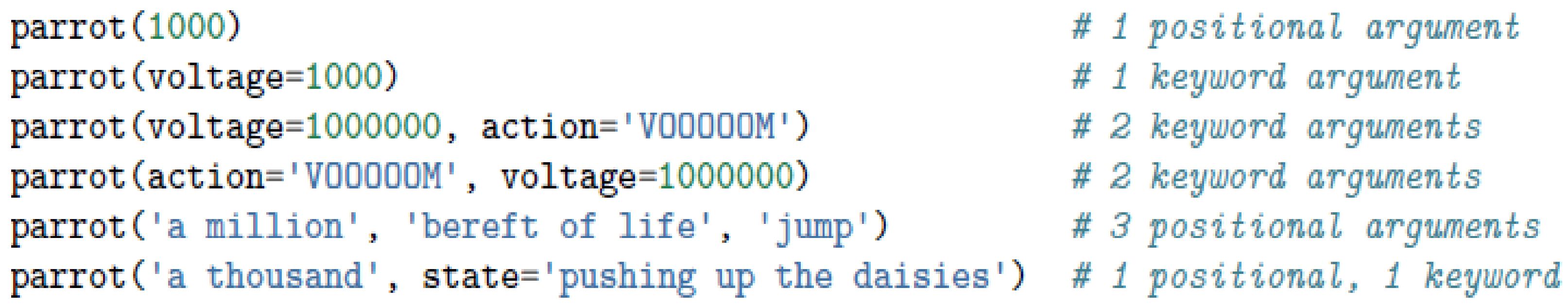
- giving only the mandatory argument: ask_ok('Do you really want to quit?')
- giving one of the optional arguments: ask_ok('OK to overwrite the file?', 2)
- or even giving all arguments: ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')

Keyword arguments

- last arguments can be given as keywords

```
def parrot(voltage, state='a stiff', action='voom',
 type='Norwegian blue'):
  print "-- This parrot wouldn't", action,
  print "if you put", voltage, "Volts through it."
  print "Lovely plumage, the ", type
  print "-- It's", state, "!"

parrot(1000)
parrot(action='VOOOM', voltage=100000)
```



```
parrot(1000)                                      # 1 positional argument
parrot(voltage=1000)                              # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')         # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)         # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')     # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

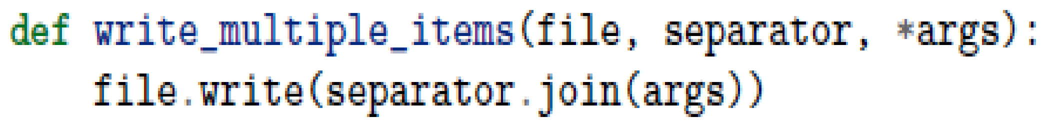
```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

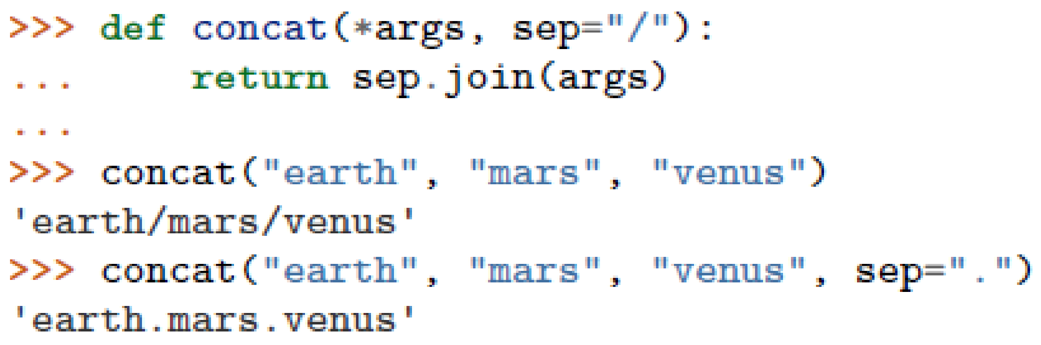
```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Arbitrary Number of Arguments

- These arguments will be wrapped up in a tuple. Before the variable number of arguments, zero or more normal arguments may occur.



```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```



```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

Unpacking Argument Lists

- The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments.
- If they are not available separately, write the function call with the *-operator to unpack the arguments out of a list or tuple.

```
>>> list(range(3, 6))            # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))            # call with arguments unpacked from a list
[3, 4, 5]
```

- In the same fashion, dictionaries can deliver keyword arguments with the **-operator.

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

Lambda forms

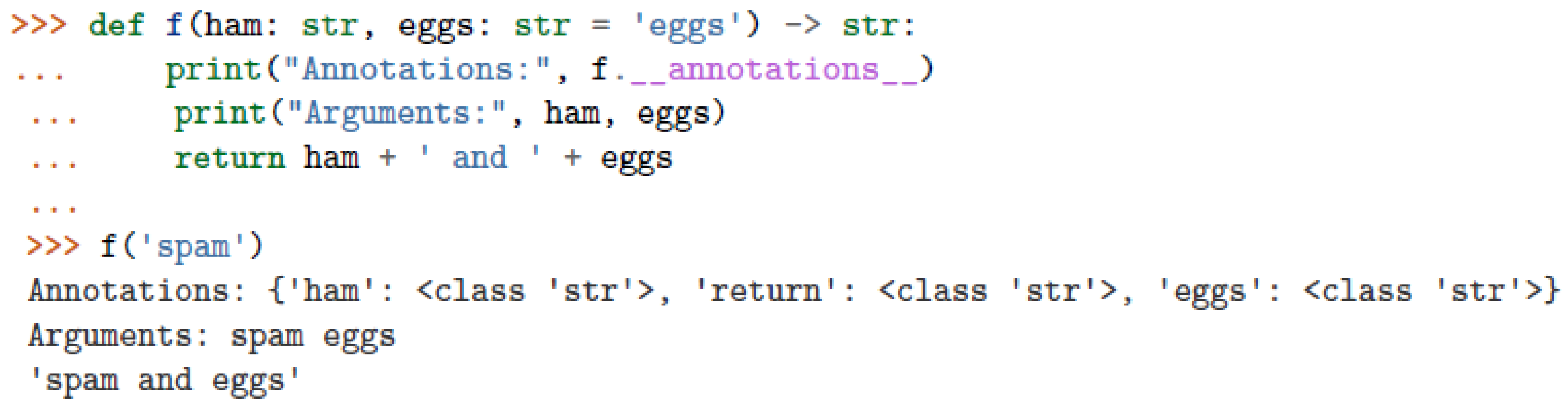
```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

- anonymous functions
- may not work in older versions
- Small anonymous functions can be created with the lambda keyword. This function returns the sum of its two arguments: lambda a, b: a+b. Lambda functions can be used wherever function objects are required.
- They are syntactically restricted to a single expression.
- Like nested function definitions, lambda functions can reference variables from the containing scope.
- Another use is to pass a small function as an argument.

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

Function Annotations

- Function annotations are completely optional metadata information about the types used by user-defined functions.



```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

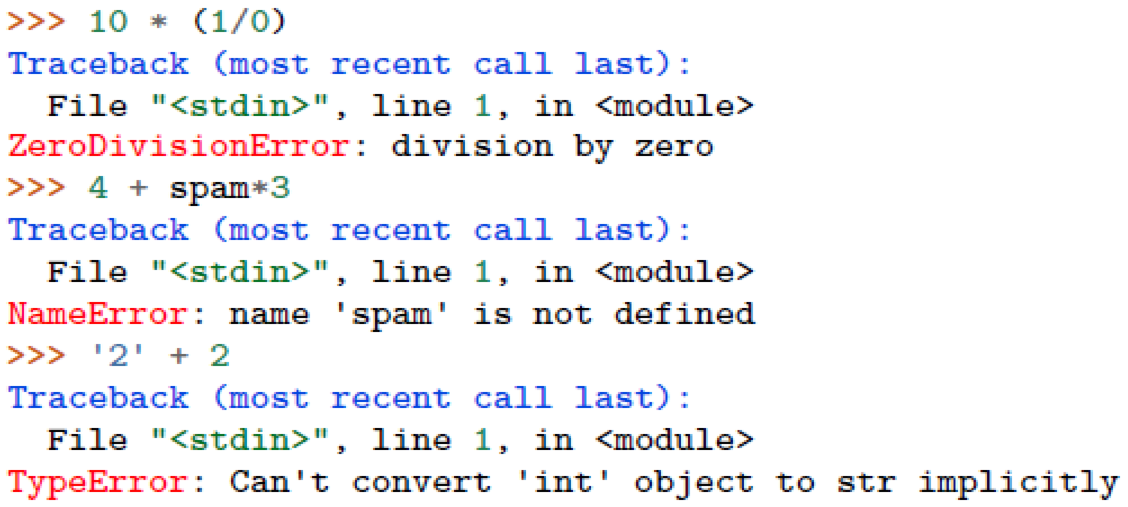
Command-line arguments

```
import sys
print len(sys.argv) # NOT argc
# Print all arguments:
print sys.argv
# Print all arguments but the program
# or module name:
print sys.argv[1:]  # "array slice"
```

Exceptions

- Try/except/raise

```
>>> while 1:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops! That was not valid.  Try again"
...
```



```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Handling exceptions

```
while 1:
  try:
    x = int(raw_input("Please enter a number: "))
    break
  except ValueError:
    print "Not a valid number"
```

- First, execute `try` clause
- if no exception, skip `except` clause
- if exception, skip rest of `try` clause and use except clause
- if no matching exception, attempt outer `try` statement

Handling exceptions

- try.py

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

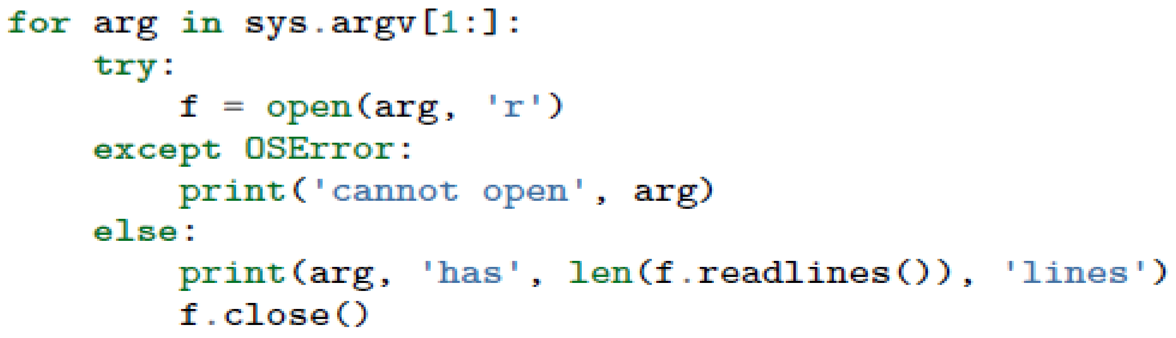
- e.g., as `python try.py *.py`

Exceptions

- syntax (parsing) errors

```
while 1 print 'Hello World'
File "<stdin>", line 1
    while 1 print 'Hello World'
                ^
SyntaxError: invalid syntax
```

- exceptions
  - run-time errors
  - e.g., `ZeroDivisionError, NameError, TypeError`



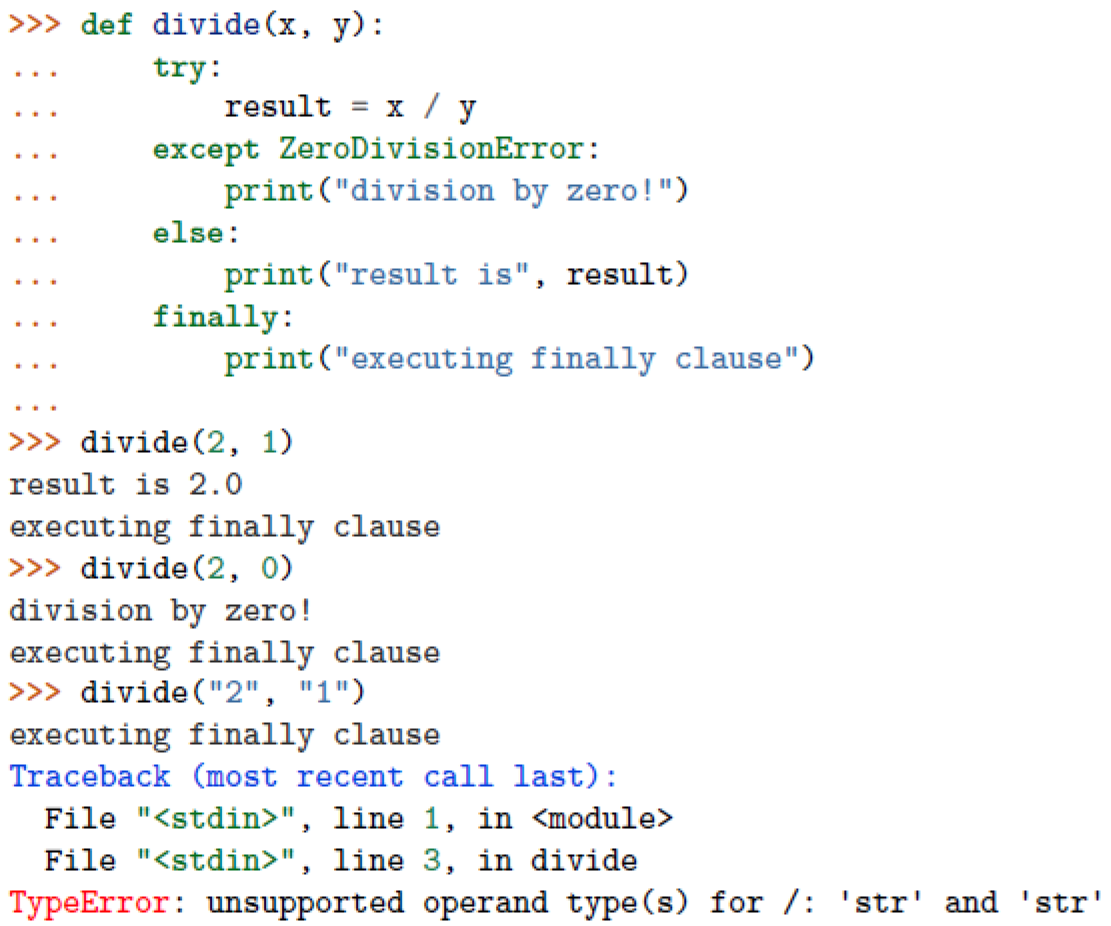
```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

Catching Exceptions

```
#python code a.py
x = 0
try:
    print 1/x
except ZeroDivisionError, message:
    print "Can’t divide by zero:"
    print message
>>>python a.py
Can't divide by zero:
integer division or modulo by zero
```

Try-Finally: Cleanup

```
f = open(file)
try:
    process_file(f)
finally:
    f.close()     # always executed
print "OK"        # executed on success only
```



```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Raising Exceptions

- raise IndexError
- raise IndexError("k out of range")
- raise IndexError, "k out of range”
- try:

```
try:
    something
except:         # catch everything
    print "Oops"
    raise       # reraise
```

Reading Assignment

- Guido van Rossum and Fred L. Drake, Jr. (ed.), *Python tutorial*, PythonLabs, 2001.
  - Read chapters 6 to 9
  - http://www.python.org/doc/current/tut/tut.html
  - Write some simple programs
- Glance at the RE HOWTO and reference
  - You will need REs in the next assignment
  - HOWTO
    - http://py-howto.sourceforge.net/regex/regex.html
  - Module reference
    - http://www.python.org/doc/current/lib/module-re.html

References

- Python Homepage
  - http://www.python.org
- Python Tutorial
  - http://docs.python.org/tutorial/
- Python Documentation
  - http://www.python.org/doc
- Python Library References
  - http://docs.python.org/release/2.5.2/lib/lib.html
- Python Add-on Packages:
  - http://pypi.python.org/pypi
- Data Science from Scratch – First principles with Python, Joel Grus, O’Reilly.
- Python Data Science Handbook – Essential Tools for Working with Data, Jake VanderPlas, O’Reilly.