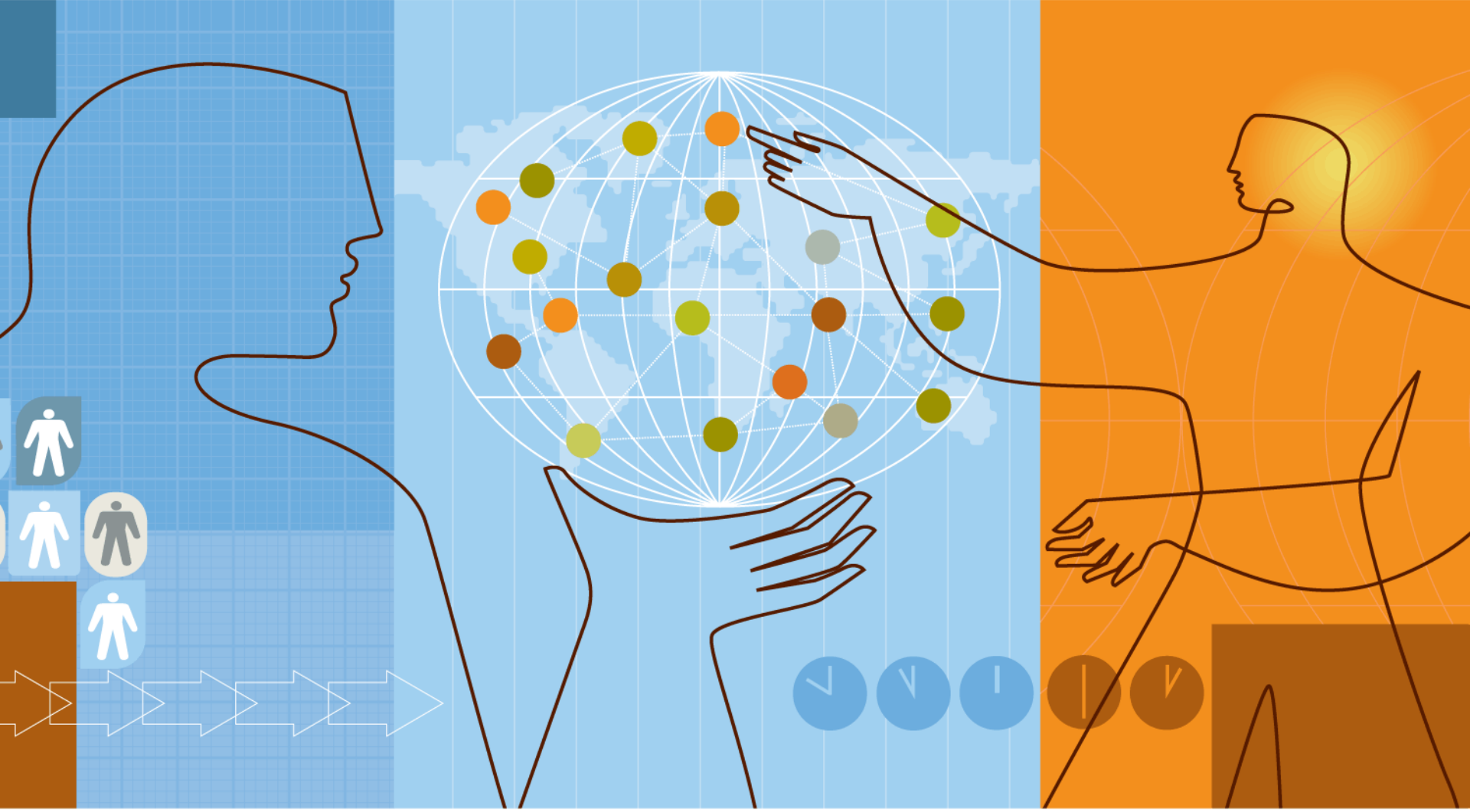


ARICENT™

Python Scripting Basis

Content

- Introduction to Python
- Variables and built in types
- Conditional Statements
- Functions
- Files
- Modules

"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Introduction – Why Python?

- Python is object-oriented
  - Structure supports such concepts as polymorphism, operation overloading, and multiple inheritance
- It's free (open source)
  - Downloading and installing Python is free and easy
  - Source code is easily accessible
  - Free doesn't mean unsupported! Online Python community is huge
- It's portable
  - Python runs virtually every major platform used today
  - As long as you have a compatible Python interpreter installed, Python programs will run in exactly the same manner, irrespective of platform
- It's powerful
  - Dynamic typing
  - Built-in types and tools
  - Library utilities
  - Third party utilities (e.g. Numeric, NumPy, SciPy)
  - Automatic memory management

Introduction – Why Python?

- It's mixable
  - Python can be linked to components written in other languages easily
    - Linking to fast, compiled code is useful to computationally intensive problems
    - Python is good for code steering and for merging multiple programs in otherwise conflicting languages
  - Python/C integration is quite common
- It's easy to use
  - Rapid turnaround: no intermediate compile and link steps as in C or C++
  - Python programs are compiled automatically to an intermediate form called *bytecode*, which the interpreter then reads
  - This gives Python the development speed of an interpreter without the performance loss inherent in purely interpreted languages
- It's easy to learn
  - Structure and syntax are pretty intuitive and easy to grasp

Running Python

- In addition to being a programming language, Python is also an interpreter. The interpreter reads other Python programs and commands, and executes them. Note that Python programs are compiled automatically before being scanned into the interpreter. The fact that this process is hidden makes Python faster than a pure interpreter.

- How to call up a Python interpreter will vary a bit depending on your platform, but in a system with a terminal interface, all you need to do is type “python” (without the quotation marks) into your command line.

- Example:
  - # From here on, the $ sign denotes the start of a terminal command line, and the # sign denotes a comment. Note: the # sign denotes a comment in Python. Python ignores anything written to the right of a # sign on a given line
  - $ python # Type python into your terminal's command line
  - >>> # After a short message, the >>> symbol will appear. This signals
  - # the start of a Python interpreter's command line

Running Python

- Once you're inside the Python interpreter, type in commands at will.

- Examples:
  - >>> print 'Hello world'
    - Hello world
  - # Relevant output is displayed on subsequent lines without the >>> symbol
    - >>> x = [0,1,2]
  - # Quantities stored in memory are not displayed by default
    - >>> x
  - # If a quantity is stored in memory, typing its name will display it
    - [0,1,2]
  - >>> 2+3
    - 5
  - >>> # Type ctrl-D to exit the interpreter
    - $

Running Python

- Python scripts can be written in text files with the suffix .py. The scripts can be read into the interpreter in several ways:

- Examples:
  - $ python script.py
    - # This will simply execute the script and return to the terminal afterwards
  - $ python -i script.py
    - # The -i flag keeps the interpreter open after the script is finished running
  - $ python
    - >>> execfile('script.py')
    - # The execfile command reads in scripts and executes them immediately, as though they had been typed into the interpreter directly
  - $ python
    - >>> import script # DO NOT add the .py suffix. Script is a *module* here
    - # The import command runs the script, displays any unstored outputs, and creates a lower level (or context) within the program. More on contexts later.

Running Python

- Suppose the file script.py contains the following lines:
  - print 'Hello world'
  - x = [0,1,2]
- Let's run this script in each of the ways described on the last slide:

- Examples:
  - $ python script.py
    - Hello world
  - $
  - # The script is executed and the interpreter is immediately closed. x is lost.
  - $ python -i script.py
  - Hello world
  - >>> x
  - [0,1,2]
  - >>>
  - # “Hello world” is printed, x is stored and can be called later, and the interpreter is left open

Running Python

- Examples: (continued from previous slide)
  - $ python
    - >>> execfile('script.py')
    - Hello world
    - >>> x
    - [0,1,2]
    - >>>
    - # For our current purposes, this is identical to calling the script from the terminal with the command python -i script.py
  - $ python
    - >>> import script
    - Hello world
    - >>> x
  - Traceback (most recent call last):
    - File "<stdin>", line 1, in ?
    - NameError: name 'x' is not defined
    - >>>
    - # When script.py is loaded in this way, x is not defined on the top level

"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Running Python

- Examples: (continued from previous slide)
  - # to make use of x, we need to let Python know which module it came from, i.e. give Python its context
  - >>> script.x
  - [0,1,2]
  - >>>
  - # Pretend that script.py contains multiple stored quantities. To promote x (and only x) to the top level context, type the following:
    - $ python
    - >>> from script import x
    - Hello world
    - >>> x
    - [0,1,2]
    - >>>
  - # To promote all quantities in script.py to the top level context, type
  - from script import * into the interpreter. Of course, if that's what you want, you might as well type python -i script.py into the terminal.

Variables and Types

```
>>> a = 'Hello world!'    # this is an assignment statement
>>> print a
'Hello world!'
>>> type(a)               # expression: outputs the value in interactive mode
<type 'str'>
```

- Variables are created when they are assigned
- No declaration required
- The variable name is case sensitive: ‘val’ is not the same as ‘Val’
- The type of the variable is determined by Python
- A variable can be reassigned to whatever, whenever

```
>>> n = 12
>>> print n
12
>>> type(n)
<type 'int'>

>>> n = 12.0
>>> type(n)
<type 'float'>
```

```
>>> n = 'apa'
>>> print n
'apa'
>>> type(n)
<type 'str'>
```


“The contents here are for Aricent internal training purposes only and do not carry any commercial value”

Numbers

- Integers: 12 0 -12987 0123 0X1A2
  - Type ‘int’
  - Can’t be larger than 2**31
  - Octal literals begin with 0 (0981 illegal!)
  - Hex literals begin with 0X, contain 0-9 and A-F
- Floating point: 12.03 1E1 -1.54E-21
  - Type ‘float’
  - Same precision and magnitude as C double
- Long integers: 10294L
  - Type ‘long’
  - Any magnitude
  - Python usually handles conversions from int to long
- Complex numbers: 1+3J
  - Type ‘complex’

Numeric Expressions

- The usual numeric expression operators: +, -, /, *, **, %, //
- Precedence and parentheses work as expected

```
>>> 12+5
17
>>> 12+5*2
22
>>> (12+5)*2
34
```

```
>>> 4 + 5.5
9.5
>>> 1 + 3.0**2
10.0
>>> 1+2j + 3-4j
(4-2j)
```

```
>>> a=12+5
>>> print a
17
>>> b = 12.4 + a              # 'a' converted to float automatically
>>> b                         # uses function 'repr'
29.399999999999999
>>> print b                   # uses function 'str'
29.4
```

Boolean Expressions

- ‘True’ and ‘ False’ are predefined values; actually integers 1 and 0
- Value 0 is considered False, all other values True
- The usual Boolean expression operators: not, and, or

```
>>> True or False
True
>>> not ((True and False) or True)
False
>>> True * 12
12
>>> 0 and 1
0
```

- Comparison operators produce Boolean values
- The usual suspects: <, <=, >, >=, ==, !=

```
>>> 12<13
True
>>> 12>13
False
>>> 12<=12
True
>>> 12!=13
True
```

Strings

```
>>> a = 'Hello world!'
>>> b = "Hello world!"
>>> a == b
True
```

```
>>> a = "Per's lecture"
>>> print a
Per's lecture
```

- Single quotes or double quotes can be used for string literals
- Produces exactly the same value
- Special characters in string literals: \n newline, \t tab, others
- Triple quotes useful for large chunks of text in program code

```
>>> a = "One line.\nAnother line."
>>> print a
One line.
Another line.
```

```
>>> b = """One line,
another line."""
>>> print b
One line,
another line.
```


"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

String Conversions

```
>>> a = "58"
>>> type(a)
<type 'str'>
>>> b=int(a)
>>> b
58
>>> type(b)
<type 'int'>
```

```
>>> f = float('1.2e-3')
>>> f                  # uses 'repr'
0.0011999999999999999
>>> print f            # uses 'str'
0.0012
>>> eval('23-12')
11
```

- Convert data types using functions ‘str’, ‘int’, ‘float’
- ‘repr’ is a variant of ‘str’
  - intended for strict, code-like representation of values
  - ‘str’ usually gives nicer-looking representation
- Function ‘eval’ interprets a string as a Python expression

```
>>> c = int('blah')            # what happens when something illegal is done?
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in -toplevel-
    c = int('blah')
ValueError: invalid literal for int(): blah
```

String Operations

```
>>> a = "Part 1"
>>> b = "and part 2"
>>> a + ' ' + b          # concatenation, adding strings
'Part 1 and part 2'
>>> s = a * 2            # repeat and concatenate string
>>> print s
Part 1Part 1
```

```
>>> s[0]                 # index: one single character, offset 0 (zero)
'P'
>>> s[0:4]               # slice: part of string
'Part'
>>> s[5:]                # leave out one boundary: to the end
'1Part 1'
>>> >>> s[6:-1]          # negative index counts from the end
'Part '
```

```
>>> len(s)               # function 'len' to get length of string
12
>>> 'p' in s             # membership test
False
>>> 'P' in s
True
>>> 'Part' in s          # also works for substrings (new feature)
True
```


"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

String Methods

- Strings have a set of built-in methods
- No method ever changes the original string!
- Several methods produce new strings

```
>>> s = 'a string, with stuff'
>>> s.count('st')                    # how many substrings?
2
>>> s.find('stu')                    # give location of substring, if any
15
>>> three = '3'
>>> three.isdigit()                  # only digit characters in string?
True
```

```
>>> supper = s.upper()               # convert to upper case
>>> supper
'A STRING, WITH STUFF'
>>> s.rjust(30)                      # right justify by adding blanks
'          a string, with stuff'
>>> "newlines\n\n\n".strip()         # a string literal also has methods!
'newlines'
```

```
>>> s.replace('stuff', 'characters') # replace substring (all occurrences)
'a string, with characters'
>>> s.replace('s', 'X', 1)           # replace only once
'a Xtring, with stuff'
```


"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Changing Strings

```
>>> cheer="Happy Holidays!"; cheer
'Happy Holidays!'
>>> cheer[6]='B' #Cannot do
Traceback(most recent call last):
File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> cheer[6:11]
'Holid'
>>> cheer[6:11]="Birth" #Still cannot do
Traceback(most recent call last):
File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
>>> cheer=cheer[0:6] + "Birthday" + cheer[-1]
>>> cheer
'Happy Birthday!'
```

NALANDA

"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

ARICENT™

List

- Ordered collection of objects; array
- Heterogenous; may contain mix of objects of any type

```
>>> r = [1, 2.0, 3, 5]          # list literal; different types of values
>>> r
[1, 2.0, 3, 5]
>>> type(r)
<type 'list'>
```

```
>>> r[1]                        # access by index; offset 0 (zero)
2.0
>>> r[-1]                       # negative index counts from end
5
```

```
>>> r[1:3]                      # a slice out of a list; gives another list
[2.0, 3]
```

```
>>> w = r + [10, 19]            # concatenate lists; gives another list
>>> w
[1, 2.0, 3, 5, 10, 19]
>>> r                           # original list unchanged; w and r are different
[1, 2.0, 3, 5]
```

```
>>> t = [0.0] * 10              # create an initial vector using repetition
>>> t
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

List Operations

- Lists are mutable; can be changed in-place
- Lists are dynamic; size may be changed

```
>>> r = [1, 2.0, 3, 5]
>>> r[3] = 'word'              # replace an item by index
>>> r
[1, 2.0, 3, 'word']
```

```
>>> r[0] = [9, 8]              # lists can be nested
>>> r
[[9, 8], 2.0, 3, 'word']
```

```
>>> r[0:3] = [1, 2, 5, 6]     # change a slice of list; may change list length
>>> r
[1, 2, 5, 6, 'word']
>>> r[1:3] = []               # remove items by setting slice to empty list
>>> r
[1, 6, 'word']
```

```
>>> len(r)                    # length of list; number of items
3
```

```
>>> 6 in r                    # membership test
True
>>> r.index(6)                # search for position; bombs if item not in list
1
```


"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

List Methods

- Lists have a set of built-in methods
- Some methods change the list in-place

```
>>> r = [1, 2.0, 3, 5]
>>> r.append('thing')              # add a single item to the end
>>> r
[1, 2.0, 3, 5, 'thing']
>>> r.append(['another', 'list'])  # list treated as a single item
>>> r
[1, 2.0, 3, 5, 'thing', ['another', 'list']]
```

```
>>> r = [1, 2.0, 3, 5]
>>> r.extend(['item', 'another'])  # list items appended one by one
>>> r
[1, 2.0, 3, 5, 'item', 'another']
```

```
>>> k = r.pop()                    # remove last item from list and return
>>> k
'another'
>>> r
[1, 2.0, 3, 5, 'item']
```

- Methods 'append' and 'pop' can be used to implement a stack

List Methods

- Use the built-in 'sort' method: efficient
- The list is sorted in-place; a new list is not produced!

```
>>> r = [2, 5, -1, 0, 20]
>>> r.sort()
>>> r
[-1, 0, 2, 5, 20]
```

```
>>> w = ['apa', '1', '2', '1234']
>>> w.sort()                          # strings: lexical sort using ASCII order
>>> w
['1', '1234', '2', 'apa']
```

```
>>> w.reverse()                        # how to flip a list; in-place!
>>> w
['apa', '2', '1234', '1']
```

```
>>> v = w[:]                          # first create a copy of the list
>>> v.reverse()                       # then reverse the copy
>>> v                                 # use same technique for sort
['1', '1234', '2', 'apa']
>>> w
['apa', '2', '1234', '1']
```

Converting Lists between strings

```
>>> s = 'biovitrum'                                  # create a string
>>> w = list(s)                                      # convert into a list of
chars
>>> w
['b', 'i', 'o', 'v', 'i', 't', 'r', 'u', 'm']
>>> w.reverse()
>>> w
['m', 'u', 'r', 't', 'i', 'v', 'o', 'i', 'b']
>>> r = ''.join(w)                                   # join using empty string
>>> r
'murtivoib'
>>> d = '-'.join(w)                                  # join using dash char
>>> d
'm-u-r-t-i-v-o-i-b'
```

```
>>> s = 'a few words'
>>> w = s.split()              # splits at white-space (blank, newline)
>>> w
['a', 'few', 'words']
```

- 'split' is useful for simple parsing
- Otherwise use regular expression module 're'; later

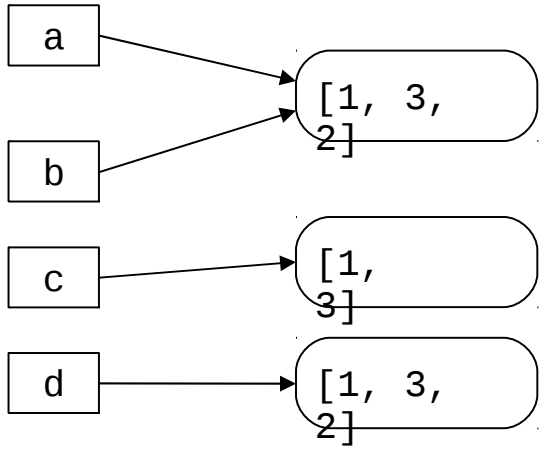
```
>>> ' | '.join(w)             # use any string with method 'join'
'a | few | words'
```


"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Objects, names and References

- All values are objects
- A variable is a name referencing an object
- An object may have several names referencing it
- Important when modifying objects in-place!
- You may have to make proper copies to get the effect you want
- For immutable objects (numbers, strings), this is never a problem

```
>>> a = [1, 3, 2]
>>> b = a
>>> c = b[0:2]
>>> d = b[:]
```




```
>>> b.sort()    # 'a' is affected!
>>> a
[1, 2, 3]
```

Dictionary

- An unordered collection of key/value pairs
- Each key maps to a value
- Also called "mapping", "hash table" or "lookup table"

```
>>> h = {'key': 12, 'nyckel': 'word'}
>>> h['key']                                  # access by key
12
>>> h.has_key('nyckel')
True
```

```
>>> h['Per'] = 'Kraulis'                      # adding a key/value
>>> h
{'nyckel': 'word', 'Per': 'Kraulis', 'key': 12}  # the output order is random
>>> h['Per'] = 'Johansson'                    # replaces the value
>>> h
{'nyckel': 'word', 'Per': 'Johansson', 'key': 12}
```

- The key is
  - Usually an integer or a string
  - Should (must!) be an immutable object
  - May be any object that is 'hashable' (more later)
  - Any key occurs at most once in a dictionary!
- The value may be any object
  - Values may occur many times

Dictionaries

- Python has a built in dictionary type called dict which you can use to create dictionaries with arbitrary definitions for character strings.
  - spanish = dict()
    spanish['hello'] = 'hola'
    spanish['yes'] = 'si'
    spanish['one'] = 'uno'
    spanish['two'] = 'dos'
    spanish['three'] = 'tres'
    spanish['red'] = 'rojo'
    spanish['black'] = 'negro'
    spanish['green'] = 'verde'
    spanish['blue'] = 'azul'

    print spanish['two']
    print spanish['red']

Forgetting Things : del

- Use command 'del' to get rid of stuff
- Command! Not function!
- Actually removes variables (names), not objects

```
>>> a = 'thing'                                    # define a variable
>>> a
'thing'
>>> del a                                          # forget about the variable
>>> a
Traceback (most recent call last):
  File "<pyshell#182>", line 1, in -toplevel-
    a
NameError: name 'a' is not defined
```

```
>>> h = {'key': 12, 'nyckel': 'word'}
>>> del h['key']                              # remove the key and its value
>>> h
{'nyckel': 'word'}
```

```
>>> r = [1, 3, 2]
>>> del r[1]                                  # another way of removing list
items
>>> r
[1, 2]
```

Dictionary Methods

```
>>> h = {'key': 12, 'nyckel': 'word'}
>>> 'Per' in h                          # test if key in dictionary
False
>>> h['Per']
Traceback (most recent call last):
  File "<pyshell#192>", line 1, in -toplevel-
    h['Per']
KeyError: 'Per'
```

```
>>> h.get('Per', 'unknown')            # return value, or default if not found
'unknown'
>>> h.get('key', 'unknown')
12
```

```
>>> h.keys()                    # all keys in a list; unordered
['nyckel', 'key']
>>> h.values()                  # all values in a list; unordered
['word', 12]
```

```
>>> len(h)                      # number of keys in dictionary
2
```


"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Tuple

- Same as list, except immutable
- Once created, can't be changed
- Some functions return tuples

```
>>> t = (1, 3, 2)
>>> t[1]                          # access by index; offset 0 (zero)
3
```

```
>>> (a, b, c) = t                 # tuple assignment (unpacking)
>>> a
1
>>> b
3
>>> a, b, c                       # actually a tuple expression!
(1, 3, 2)
```

```
>>> a, b = b, a                   # neat trick to swap values
>>> a, b
(3, 1)
```

```
>>> r = list(t)                   # convert tuple to a list
>>> r
[1, 3, 2]
>>> tuple(r)                      # convert list to a tuple
(1, 3, 2)
```

String Formatting

- String formatting operator '%'
- Usually the best way to create new strings
- C-like formatting: Slightly tricky, but powerful
- Many string formatting codes
  - %s: string (uses function 'str')
  - %r: string (uses function 'repr')
  - %f, %e, %g: float

```
>>> w = "Number %i won!" % 12     # string formatting operator %
>>> w
'Number 12 won!'
```

- Tuples are used as operands in string formatting when >1 items
- The length of the tuple must match the number of format codes in the string
- Lists won't do!

```
>>> c = 'Python'
>>> n = 11
>>> "This is a %s course with %i students." % (c, n)
'This is a Python course with 11 students.'
```

Dictionary Methods

```
>>> g = h.copy()               # a separate copy of the dictionary
>>> del h['key']
>>> h
{'nyckel': 'word'}
>>> g
{'nyckel': 'word', 'key': 12}
```

```
>>> h['Per'] = 'Johansson'
>>> h
{'nyckel': 'word', 'Per': 'Johansson'}
>>> h.update(g)                          # add or update all key/value from g
>>> h
{'nyckel': 'word', 'key': 12, 'Per': 'Johansson'}
```


"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Getting data from user

- *input* function can be used to get input from *stdin*.
  - person = input('Enter your name: ')
    print 'Hello', person
  - x, y = input("Enter two comma separated numbers: ")
    print 'The sum of %s and %s is %s.' % (x, y, x+y)

Indexing and Slicing

- Indexing and slicing
  - Python starts indexing at 0. A string s will have indexes running from 0 to len(s)-1 (where len(s) is the length of s) in integer quantities.
  - s[i] fetches the ith element in s

```
>>> s = 'string'
>>> s[1] # note that Python considers 't' the first element
't'      # of our string s
```

  - s[i:j] fetches elements i (inclusive) through j (not inclusive)

```
>>> s[1:4]
'tri'
```

  - s[:j] fetches all elements up to, but not including j

```
>>> s[:3]
'str'
```

  - s[i:] fetches all elements from i onward (inclusive)

```
>>> s[2:]
'ring'
```

Indexing and Slicing

- Indexing and slicing, contd.
  - s[i:j:k] extracts every kth element starting with index i (inlcusive) and ending with index j (not inclusive)
    ```
    >>> s[0:5:2]
    'srn'
    ```
  - Python also supports negative indexes. For example, s[-1] means extract the first element of s from the end (same as s[len(s)-1])
    ```
    >>> s[-1]
    'g'
    >>> s[-2]
    'n'
    ```

Mutable Vs Immutable Types

- Mutable types (dictionaries, lists, arrays) can have individual items reassigned in place, while immutable types (numbers, strings, tuples) cannot.
  - >>> L = [0,2,3]
  - >>> L[0] = 1
  - >>> L
  - [1,2,3]
  - >>> s = 'string'
  - >>> s[3] = 'o'
  - Traceback (most recent call last):
  - File "<stdin>", line 1, in ?
  - TypeError: object does not support item assignment
- However, there is another important difference between mutable and immutable types; they handle name assignments differently. If you assign a name to an immutable item, then set a second name equal to the first, changing the value of the first name will not change that of the second. However, for mutable items, changing the value of the first name will change that of the second.
- An example to illustrate this difference follows on the next slide.

Mutable Vs Immutable Types

- Immutable and mutable types handle name assignments differently
  - >>> a = 2
  - >>> b = a # a and b are both numbers, and are thus immutable
  - >>> a = 3
  - >>> b
  - 2
- Even though we set b equal to a, changing the value of a does not change the value of b. However, for mutable types, this property does not hold.
  - >>> La = [0,1,2]
  - >>> Lb = La # La and Lb are both lists, and are thus mutable
  - >>> La = [1,2,3]
  - >>> Lb
  - [1,2,3]
- Setting Lb equal to La means that changing the value of La changes that of Lb. To circumvent this property, we would make use of the function copy.copy().
  - >>> La = [0,1,2]
  - >>> Lb = copy.copy(La)
- Now, changing the value of La will not change the value of Lb.

Types - Associated Functions

- *type* function returns the type of the object.
  - Example
    - >>> type(7)
    - <type ‘int’>
    - >>> type(1.5)
    - <type ‘float’>
    - >>> type(‘Hello’)
    - <type ‘str’>
    - type([1, 2, 3])
    - <type ‘list’>
- *len* function returns the length of sequence of string and list
  - Example
    - >>> len([2, 4, 6])
    - 3
    - len('abcd')
    - 4

Types – Associated Functions

- *max* functions returns maximum from a list
  - >>> max(5, 11, 2)
- Conversion between int and str
  - >>> str(123)
  - >>> int('123')

Basic Statements : if statement block structure

- If statements have the following basic structure:
  - # inside the interpreter # inside a script
  - >>> if condition: if condition:
  - ... action action
  - ...
- >>>
- Subsequent indented lines are assumed to be part of the if statement. The same is true for most other types of python statements. A statement typed into an interpreter ends once an empty line is entered, and a statement in a script ends once an unindented line appears. The same is true for defining functions.
- If statements can be combined with else if (elif) and else statements as follows:
  - if condition1: # if condition1 is true, execute action1
  - action1
  - elif condition2: # if condition1 is not true, but condition2 is, execute
  - action2 # action2
  - else: # if neither condition1 nor condition2 is true, execute
  - action3 # action3

Basic Statements : if statement block structure

- Conditions in if statements may be combined using and & or statements
  - if condition1 and condition2:
  - action1
  - # if both condition1 and condition2 are true, execute action1
    - if condition1 or condition2:
    - action2
  - # if either condition1 or condition2 is true, execute action2
- Conditions may be expressed using the following operations:
  - <, <=, >, >=, ==, !=, in
- Somewhat unrealistic example:
  - >>> x = 2; y = 3; L = [0,1,2]
  - >>> if (1<x<=3 and 4>y>=2) or (1==1 or 0!=1) or 1 in L:
  - ... print 'Hello world'
  - ...
  - Hello world
  - >>>

Basic Statements : if statement block structure

```
person = 'Luke'

if person == 'Per':
    status = 'Pythonist'
elif person == 'Luke':
    status = 'Jedi knight'
else:
    status = 'unknown'

print person, status
```


"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Dictionary often better than if… elif…

- Particularly with many hardcoded choices (elif's)…

```
"file t3.py"

status_map = {'Luke': 'Jedi Knight',
              'Per': 'Pythonist'}

person = 'Luke'

print person, status_map.get(person, 'unknown')
```

- More compact, and more efficient
- This pattern is very useful

"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Built-in types and their representations

int	0	False
	-1	True
	124	True
float	0.0	False
str	""	False
	"False"	True !
dict	{ }	False
	{'key': 'val'}	True
list	[ ]	False
	[False]	True !

- All built-in types can be used directly in 'if' statements
- Zero-valued numbers are False
- All other numbers are True
- Empty containers (str, list, dict) are False
- All other container values are True
- Use function 'bool' to get explicit value

Basic Statements : while

- While statements have the following basic structure:
  - # inside the interpreter # inside a script
  - >>> while condition: while condition:
  - ... action action
  - ...
- >>>
- As long as the condition is true, the while statement will execute the action
  - Example:
    - >>> x = 1
    - >>> while x < 4: # as long as x < 4...
    - ... print x**2 # print the square of x
    - ... x = x+1 # increment x by +1
    - ...
    - 1 # only the squares of 1, 2, and 3 are printed, because
    - 4 # once x = 4, the condition is false
    - 9
    - >>>

Basic Statements : while

- Pitfall to avoid:
  - While statements are intended to be used with changing conditions. If the condition in a while statement does not change, the program will be stuck in an infinite loop until the user hits ctrl-C.
  - Example:
    - >>> x = 1
    - >>> while x == 1:
    - ... print 'Hello world'
    - ...
  - Since x does not change, Python will continue to print “Hello world” until interrupted

Basic Statements : while

- Repetition of a block of statements
- Loop until test becomes false, or 'break'

```
r = []
n = 0
last = 20

while n <= last:            # any expression interpretable as Boolean
    r.append(str(n))
    n += 3

print ', '.join(r)
```


"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Basic Statements : for

- for statements have the following basic structure:
  - for item i in set s:
    - action on item i
  - # item and set are not statements here; they are merely intended to clarify the relationships between i and s
  - Example:
    - >>> for i in range(1,7):
    - ... print i, i**2, i**3, i**4
    - ...
    - 1 1 1 1
    - 2 4 8 16
    - 3 9 27 81
    - 4 16 64 256
    - 5 25 125 625
    - 6 36 216 1296
    - >>>

"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Basic Statements : for

- The item i is often used to refer to an index in a list, tuple, or array
  - Example:
    - >>> L = [0,1,2,3] # or, equivalently, range(4)
    - >>> for i in range(len(L)):
    - ... L[i] = L[i]**2
    - ...
    - >>> L
    - [0,1,4,9]
    - >>>
  - Of course, we could accomplish this particular task more compactly using arrays:
    - >>> L = arange(4)
    - >>> L = L**2
    - >>> L
    - [0,1,4,9,]

Basic Statements : for

- Repetition of a block of statements
- Iterate through a sequence (list, tuple, string, iterator)

```
s = 0
for i in [0, 1, 2, 3, 4, 5, 6, 7, 8]:     # walk through list, assign to i
    s = s + i
    if s > 10:
        break                             # quit 'for' loop, jump to after it

print "i=%i, s=%i" % (i, s)
```

```
r = []
for c in 'this is a string with blanks':  # walks through string, char by
char
    if c == ' ': continue                 # skip rest of block, continue loop
    r.append(c)

print ''.join(r)
```


"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Built-in functions 'range' and 'xrange'

- Built-in functions 'range' and 'xrange' useful with 'for'
- 'range' creates a list
- Warning: may use lots of memory; inefficient!

```
>>> range(9)                        # start=0, step=1 by default
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> range(1, 12, 3)                 # explicit start, end, step
[1, 4, 7, 10]
>>> range(10**9)                    # MemoryError!
```

- 'xrange' creates an iterator, which works like a list
- Very memory-efficient!

```
s = 0
for i in xrange(100000):
    if i % 19 == 0:                 # remainder: evenly divisible with 19?
        s = s + i

print s
```

Optional 'else' block in loops

- 'else' block executed if no 'break' encountered
- May often replace success/failure flags
- Valid in 'for' and 'while' loops

```
r = [1, 3, 10, 98, -2, 48]

for i in r:
    if i < 0:
        print 'input contains negative value!'
        break                   # this skips out of loop, including 'else'
    else:
        pass                    # do-nothing statement
else:                           # do if loop ended normally
    print 'input is OK'
```

- 'pass' statement does absolutely nothing
- May be useful as placeholder for unwritten code
- A few cases where required (more later)

"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Error handling: “try” and “except”

- Run-time error normally causes execution to bomb
- The error message gives type of error
- Use a ’try’, ’except’ blocks to catch and handle errors

```
numbers = []
not_numbers = []

for s in ['12', '-4.1', '1.0e2', 'e3']:
    try:
        n = float(s)
        numbers.append(s)
    except ValueError, msg:
        not_numbers.append(str(msg))

print 'numbers:', numbers
print 'not numbers:', not_numbers
```

```
numbers: ['12', '-4.1', '1.0e2']
not numbers: ['invalid literal for float(): e3']
```

How to split up long lines

- Sometimes a source code line needs splitting up
- Indentation rule means we do not have free-format!

```
"illegal syntax example"

if a_complicated_expression and
   another_complicated_expression:
    print 'this is illegal syntax; it will not work'
```

- Alt 1: Use continuation character '\' as the very last

```
"valid syntax example 1"

if a_complicated_expression and \
    another_complicated_expression:
    print 'this is valid syntax'
```

- Alt 2: Enclose the expression in parenthesis
  - Lines within parenthesis can be broken up
  - Also true for [ ] and { }

```
"valid syntax example 2"

if (a_complicated_expression and
    another_complicated_expression):
    print 'this is valid syntax'
```

Functions

- Usually, function definitions have the following basic structure:
  - def func(args):
  - return values
- Regardless of the arguments, (including the case of no arguments) a function call must end with parentheses.
- Functions may be simple one-to-one mappings
  - >>> def f1(x):
  - ... return x*(x-1)
  - ...
  - >>> f1(3)
  - 6
- They may contain multiple input and/or output variables
  - >>> def f2(x,y):
  - ... return x+y,x-y
  - ...
  - >>> f2(3,2)
  - (5,1)

Functions

- Functions don't need to contain arguments at all:
  - >>> def f3():
  - ... print 'Hello world'
  - ...
  - >>> f3()
  - Hello world
- The user can set arguments to default values in function definitions:
  - >>> def f4(x,a=1):
  - ... return a*x**2
  - ...
  - >>>
- If this function is called with only one argument, the default value of 1 is assumed for the second argument
  - >>> f4(2)
  - 4
- However, the user is free to change the second argument from its default value
  - >>> f4(2,a=2) # f4(2,2) would also work
  - 8

Functions

- Use the 'def' statement
- Function body follows; indented!
- This is a statement like others
  - Can be placed basically anywhere
  - Required: Define the function before calling it

```
def all_items_positive(r):                          # function definition
    "Check that all values in list r are positive."  # documentation string
    for i in r:
        if i <= 0: return False                     # return a result value
    return True

sequences = [[1, 5, 6, -0.01], [0.01, 1, 2]]

for seq in sequences:
    if not all_items_positive(seq):                 # call the function
        print 'invalid: ', seq
```

Functions

- Anything calculated inside a function but not specified as an output quantity (either with return or global) will be deleted once the function stops running
  - >>> def f5(x,y):
  - ... a = x+y
  - ... b = x-y
  - ... return a**2,b**2
  - ...
  - >>> f5(3,2)
  - (25,1)
- If we try to call a or b, we get an error message:
  - >>> a
  - Traceback (most recent call last):
  - File "<stdin>", line 1, in ?
  - NameError: name 'a' is not defined
- This brings us to scoping issues, which will be addressed in the next section.

Scope Rules

- Python employs the following scoping hierarchy, in decreasing order of breadth:
  - Built-in (Python)
    - Predefined names (len, open, execfile, etc.) and types
  - Global (module)
    - Names assigned at the top level of a module, or directly in the interpreter
    - Names declared global in a function
  - Local (function)
    - Names assigned inside a function definition or loop
  - Example:
    - >>> a = 2 # a is assigned in the interpreter, so it's global
    - >>> def f(x): # x is in the function's argument list, so it's local
    - ... y = x+a # y is only assigned inside the function, so it's local
    - ... return y # using the sa
    - ...
    - >>>

"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Scope Rules

- If a module file is read into the interpreter via execfile, any quantities defined in the top level of the module file will be promoted to the top level of the program
- As an example: return to our friend from the beginning of the presentation, script.py:
  - print 'Hello world'
  - x = [0,1,2]
  - >>> execfile('script.py')
  - Hello world
  - >>> x
  - [0,1,2]
- If we had imported script.py instead, the list x would not be defined on the top level. To call x, we would need to explicitly tell Python its scope, or context.
  - >>> import script
  - Hello world
  - >>> script.x
  - [0,1,2]
- If we had tried to call x without a context flag, an error message would have appeared

Scope Rules

- Modules may well contain submodules. Say we have a file named module.py which, in its definition, imports a submodule named submodule, which in turn contains some quantity named x.
  - >>> import module
- If we load the module this way, we would type the following to call x:
  - >>> module.submodule.x
- We can also import the submodule without importing other quantities defined in module.py:
  - >>> from module import submodule
- In this case, we would type the following to call x:
  - >>> submodule.x
- We would also call x this way if we had read in module.py with execfile()

Scope Rules

- You can use the same names in different scopes
- Examples:
  - >>> a = 2
  - >>> def f5(x,y)
  - ... a = x+y # this a has no knowledge of the global a, and vice-versa
  - ... b = x-y
  - ... return a**2,b**2
  - ...
  - >>> a
  - 2
- The local a is deleted as soon as the function stops running
  - >>> x = 5
  - >>> import script # same script as before
  - Hello world
  - >>> x
  - 5
  - >>> script.x # script.x and x are defined in different scopes, and
  - [0,1,2] # are thus different

Scope Rules

- Changing a global name used in a function definition changes the function
- Example:
  - >>> a = 2
  - >>> def f(x):
  - ... return x+a # this function is, effectively, f(x) = x+2
  - ...
  - >>> f(4)
  - 6
  - >>> a = 1
  - >>> f(4) # since we set a=1, f(x) = x+1 now
  - 5
- Unlike some other languages, Python function arguments are not modified by default:
  - >>> x = 4
  - >>> f(x)
  - 5
  - >>> x
  - 4

Function Features

- The value of an argument is not checked for type
  - Often very useful; overloading without effort
  - Of course, the function may still bomb if invalid value is given

- The documentation string is not required (more later)
  - But strongly encouraged!
  - Make a habit of writing one (before writing the function code)

- A user-defined function has exactly the same status as a built-in function, or a function from another module

Function arguments: fixed

- Fixed number of arguments
- Associated by order

```
def fixed_args(a, c, b):                                # note!: order of args
    "Format arguments into a string and return.“        # doc string
    return "a=%s, b=%s, c=%s" % (a, b, c)               # '%s' converts to string

print fixed_args('stuff', 1.2, [2, 1])
```


“The contents here are for Aricent internal training purposes only and do not carry any commercial value”

Function arguments: variable

- List of any number of arguments
- Useful when unknown number of arguments needed
- The argument values collected into a tuple
  - Called 'args', by convention
  - The '*' is the magical part

```
def unspec_args(a, *args):                    # name 'args' is a convention
    return "a=%s, others=%s" % (a, args)

print unspec_args('bla', 'qwe', 23, False)
```

```
a=bla, others=('qwe', 23, False)
```


"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Function Arguments – Default Values

- Arguments may have default values
- When argument not given in a call, default value is used
- If no default value, and not given when called: bombs
- Use explicit names to override argument order

```
def default_args(a, b='bar', c=13):
    return "a=%s, b=%s, c=%s" % (a, b, c)

print default_args('apa')                    # uses all default values
print default_args('s', b='py')              # overrides one default value
print default_args(c=-26, a='apa')           # override argument order
```

```
a=apa, b=bar, c=13
a=s, b=py, c=13
a=apa, b=bar, c=-26
```


"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Function arguments: Explicit type checking

- Use the 'assert' statement
- Checks that its Boolean expression is True, else bombs
- Can be used for sanity checks anywhere in code
- Optional explanatory message (or data)

```
def fixed_args(a, c, b):
    assert type(a) == type(1), "'a' must be an integer"
    return "a=%s, b=%s, c=%s" % (a, b, c)

print fixed_args('a', 1.2, [2, 1])
```

```
Traceback (most recent call last):
  File "C:\Python tests\t15.py", line 8, in -toplevel-
    print fixed_args('a', 1.2, [2, 1])
  File "C:\Python tests\t15.py", line 5, in fixed_args
    assert type(a) == type(1), "'a' must be an integer"
AssertionError: 'a' must be an integer
```

Functions: Local Variables

- Arguments become local variables
  - Immutable values are copied, in effect
  - Mutable values may still be changed: be careful
- Variables created within 'def' block are local
  - Forgotten on return

```
def test_local(a, r):
    print 'local original ', a, r
    a = 12
    r[1] = 999
    print 'local changed  ', a, r

a = -5
r = [0, 1, 2]
print 'global original', a, r
test_local(a, r)
print 'global changed ', a, r
```

```
global original -5 [0, 1, 2]
local original  -5 [0, 1, 2]
local changed   12 [0, 999, 2]
global changed  -5 [0, 999, 2]
```

Functions without “return” : Value “None”

- A function does not have to use the 'return' statement
- If not, then same as a 'procedure' in other languages
- Actually returns a value anyway: 'None'
- A 'return' without value is OK: returns 'None'
- 'None' is a special value meaning 'nothing'
  - Useful in many contexts
  - Particularly in object-oriented programming (more later)

```
def concat_strings(a, b):
    str_type = type('')                                  # save a type value!
    if type(a) == str_type and type(b) == str_type:
        return a + ' ' + b

print 'strings:', concat_strings('first', 'second')
print 'integers:', concat_strings(1, 2)
```

```
strings: first second
integers: None
```

Math module: Functions and Constants

- A peek at modules
- Math functions available in a separate module

```
from math import *              # import everything from module 'math'

print e, pi
print cos(radians(180.0))
print log(10.0)
print exp(-1.0)
```

```
2.71828182846 3.14159265359
-1.0
2.30258509299
0.367879441171
```


"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Functions are objects; names are references

- A function is just another kind of object
- Nothing magical about their names; can be changed

```
from math import *

def print_calc(f):
    print "log(%s)=%s, exp(%s)=%s" % (f, log(f), f, exp(f))

print_calc(1.0)
log, exp = exp, log       # evil code! swap the objects the names refer to
print_calc(1.0)
```

```
log(1.0)=0.0, exp(1.0)=2.71828182846
log(1.0)=2.71828182846, exp(1.0)=0.0
```

- A function can be passed as any argument to another function
- A function can assigned to a variable

Built-in function 'map'

- Built-in function that works on a list
- 'map' takes a function and a list
  - The function must take only one argument, and return one value
  - The function is applied to each value of the list
  - The resulting values are returned in a list

```
>>> from math import *
>>> r = [0, 1, 2, 3, 4, 5, 6]
>>> map(cos, r)
[1.0, 0.54030230586813977, -0.41614683654714241, -0.98999249660044542,
 -0.65364362086361194, 0.28366218546322625, 0.96017028665036597]
```


"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Built-in function 'reduce'

- Built-in function that works on a list
- 'reduce' takes a function and a list
- It boils down the list into one value using the function
  - The function must take only two arguments, and return one value
  - 'reduce' applies the function to the first two values in the list
  - The function is then applied to the result and the next value in the list
  - And so on, until all values in the list have been used

```
>>> r = [0, 1, 2, 3, 4, 5, 6]
>>> def sum(x, y): return x+y

>>> reduce(sum, r)                  # (((((1+2)+3)+4)+5)+6)
21
```

Built-in function 'filter'

- Built-in function that works on a list
- 'filter' takes a function and a list
- It uses the function to decide which values to put into the resulting list
  - Each value in the list is given to the function
  - If the function return True, then the value is put into the resulting list
  - If the function returns False, then the value is skipped

```
>>> r = [0, 1, 2, 3, 4, 5, 6]
>>> def large(x): return x>3

>>> filter(large, r)
[4, 5, 6]
```

Files : reading

- A file object is created by the built-in function 'open'
- The file object has a set of methods
- The 'read' methods get data sequentially from the file
  - 'read': Get the entire file (or N bytes) and return as a single string
  - 'readline': Read a line (up to and including newline)
  - 'readlines': Read all lines and return as a list of strings

```
>>> f = open('test.txt')      # by default: read-only mode
>>> line = f.readline()       # read a single line
>>> line
'This is the first line.\n'
>>> lines = f.readlines()     # read all remaining lines
>>> lines
['This is the second.\n', 'And third.\n']
```

- Several modules define objects that are file-like, i.e. have methods that make them behave as files

Files: writing

- The 'write' method simply outputs the given string
- The string does not have to be ASCII; binary contents allowed

```
>>> w = open('output.txt', 'w')          # write mode (text by default)
>>> w.write('stuff')                     # does *not* add newline
automatically
>>> w.write('\n')
>>> w.write('more\n and even more\n')
>>> w.close()
```

```
stuff
more
 and even more
```


NALANDA

"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

ARICENT™

Files: read by 'for' loop

- Iteration using the 'for' loop over the file reads line by line
- The preferred way to do it

```
infile = open('test.txt')                # read-only mode
outfile = open('test_upper.txt', 'w')    # write mode; creates the file

for line in infile:
    outfile.write(line.upper())

infile.close()                    # not strictly required; done automatically
outfile.close()
```

- Note: Each line will contain the trailing newline '\n' character
- Use string method 'strip' or 'rstrip' to get rid of it

Files, old-style read strategies

- Previous versions of Python did not have the 'for line in file' feature
- Instead, the following alternatives were used:

```
for line in infile.readlines():     # reads entire file into list of lines
    do_something(line)
```

```
for line in infile.xreadlines():    # like xrange: more memory-efficient
    do_something(line)
```

```
line = infile.readline()
while line:                         # line will be empty only at end-of-file
    do_something(line)
    line = infile.readline()
```

- The last alternative works because 'readline' returns the line including the final newline '\n' character
- Only when end-of-file is reached will a completely empty line be returned, which has the Boolean value 'False'

Modules

- Given a nucleotide sequence, produce reverse complement
- Use available features

```
complement_map = {'c': 'g', 'g': 'c', 'a': 't', 't': 'a'}

seq = 'cgtaacggtcaggttatattt'

complist = map(complement_map.get, seq)
complist.reverse()
revseq = ''.join(complist)

print seq
print revseq
```

```
cgtaacggtcaggttatattt
aaatataacctgaccgttacg
```

Modules

- How to make the example code more reusable?
- Step 1: Make a function

```
complement_map = {'c': 'g', 'g': 'c', 'a': 't', 't': 'a'}

def reverse_complement(seq):
    complist = map(complement_map.get, seq)
    complist.reverse()
    return ''.join(complist)

seq = 'cgtaacggtcaggttatattt'
print seq
print reverse_complement(seq)
```

Modules

- How to make the code even more reusable?
- Step 2: Make a module out of it
- Is actually already a module!
- Let’s simply rename it to ’ntseq.py’

```
complement_map = {'c': 'g', 'g': 'c', 'a': 't', 't': 'a'}

def reverse_complement(seq):
    "Return the reverse complement of an NT sequence."
    complist = map(complement_map.get, seq)
    complist.reverse()
    return ''.join(complist)

seq = 'cgtaacggtcaggttatattt'
print seq
print reverse_complement(seq)
```


NALANDA
“The contents here are for Aricent internal training purposes only and do not carry any commercial value”
ARICENT™

How to use the module: ’import’ statement

- The ’import’ statement makes a module available
- The module name (not the file name) is imported: skip the ’.py’
- Access module features through the ’dot’ notation

```
import ntseq

seq = 'aaaccc'
print seq
print ntseq.reverse_complement(seq)
```

```
cgtaacggtcaggttatattt
aaatataacctgaccgttacg
aaaccc
gggttt
```

Standard Library Modules – Module “re”

Regular expressions: advanced string patterns

- Define a pattern
  - The pattern syntax is very much like Perl or grep
- Apply it to a string
- Process the results

```
import re

seq = "MAKEVFSKRTCACVFHKVHAQPNVGITR"

zinc_finger = re.compile('C.C..H..H')    # compile regular expression pattern
print zinc_finger.findall(seq)

two_charged = re.compile('[DERK][DERK]')
print two_charged.findall(seq)
```

```
['CACVFHKVH']
['KE', 'KR']
```

Some basic rules for regular expressions

- **Any ordinary character,** by itself, is an RE. Example: '"a"' is an RE that matches the character a in the candidate string. While trivial, it is critical to know that each ordinary character is a stand-alone RE.

- Some characters have special meanings. We can escape that special meaning by using a '\' in front of them. For example, '*' is a special character, but '*' escapes the special meaning and creates a single-character RE that matches the character '*'.

- Additionally, some ordinary characters can be made special with '\'. For instance '\d' is any digit, '\s' is any whitespace character. '\D' is any non-digit, '\S' is any non-whitespace character.

Some basic rules for regular expressions

- **The character '.'** is an RE that matches any single character. Example: '"x.z"' is an RE that matches the strings like xaz or xbz, but doesn't match strings like xabz.

- **The brackets, '"[...]"'**, create a RE that matches any character between the [ ]'s. Example: '"x[abc]z"' matches any of xaz, xbz or xcz. A range of characters can be specified using a '-', for example '"x[1-9]z"'. To include a '-', it must be first or last. '^' cannot be first. Multiple ranges are allowed, for example '"x[A-Za-z]z"' . Here's a common RE that matches a letter followed by a letter, digit or _: '"[A-Za-z][A-Za-z1-9_]"'.

- **The modified brackets, "[^...]"**, create a regular expression that matches any character except those between the [ ]'s. Example: '"a[^xyz]b"' matches strings like a9b and a$b, but don't match axb. As with '[ ]', a range can be specified and multiple ranges can be specified.

NALANDA

"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

ARICENT

Some basic rules for regular expressions

- **A regular expression can be formed from concatenating regular expressions**. Example: ‘"a.b"’ is three regular expressions, the first matches a, the second matches any character, the third matches b.

- **A regular expression can be a group of regular expressions, formed with ()’s**. Example: ‘"(ab)c"’ is a regular expression composed of two regular expressions: ‘"(ab)"’ (which, in turn, is composed of two RE’s) and ‘"c"’. ‘()’‘s also group RE’s for extraction purposes. The elements matched within ‘()’ are remembered by the regular expression processor and set aside in a Match object.

NALANDA

“The contents here are for Aricent internal training purposes only and do not carry any commercial value”

ARICENT™

Some basic rules for regular expressions

- **A regular expression can be repeated.** Several repeat constructs are available: '"x*"' repeats '"x"' zero or more times; '"x+"' repeats '"x"' 1 or more times; '"x?"' repeats '"x"' zero or once. Example: '"1(abc)*2"' matches '12' or '1abc2' or '1abcabc2', etc. The first match, against 12, is often surprising; but there are zero copies of abc between 1 and 2. The above ('*', '+' and '?') are called "greedy" repeats because they will match the longest string of repeats.
- There are versions which are not greedy: '*?', '+?' will match the shortest string of repeats .

- **The character "^" is an RE that only matches the beginning of the line**, "$" is an RE that only matches the end of the line. Example: "^$" matches a completely empty line.

- **The character "$" is an RE that only matches the end of the line**

NALANDA

"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

ARICENT™

Module “re”

- The most common first step is to compile the RE definition string to make an Pattern object.
- The resulting Pattern object can then be used to match or search candidate strings.
- A successful match returns a Match object with details of the matching substring.
- compile(expr)
  - Create a Pattern object from an RE string, expr. The Pattern is used for all subsequent searching or matching operations. A Pattern has several methods, including match() and search().

“The contents here are for Aricent internal training purposes only and do not carry any commercial value”

Module “re”

- The following methods are part of a Pattern object created by the re.compile() function.
  - match(string)
    - Match the candidate string against the compiled regular expression (an instance of Pattern). Matching means that the regular expression and the candidate string must match, starting at the beginning of the candidate string. A Match object is returned if there is match, otherwise None is returned.
  - search(string)
    - Search a candidate string for the compiled regular expression (an instance of Pattern). Searching means that the regular expression must be found somewhere in the candidate string. A Match object is returned if the pattern is found, otherwise None is returned.
  - group(number)
    - Retrieve the string that matched a particular ‘()’ grouping in the regular expression. Group zero is a tuple of everything that matched. Group 1 is the material that matched the first set of ‘()’‘s.

Module “re”

```
>>> import re
>>> raw_input= "20:07:13.2"
>>> hms_pat = re.compile( r'(\d+):(\d+):(\d+\.?\d*)' )
>>> hms_match= hms_pat.match( raw_input )
>>> hms_match.group( 0, 1, 2, 3 )
('20:07:13.2', '20', '07', '13.2')
>>>
>>> h,m,s= map( float, hms_match.group(1,2,3) )
>>> h
20.0
>>> m
7.0
>>>
>>> s
13.199999999999999
>>> seconds= ((h*60)+m)*60+s
>>> seconds
72433.199999999997
```

Module ‘re’

- re.split(*pattern*, *string*[, *maxsplit=0*])
  - Split *string* by the occurrences of *pattern*. If capturing parentheses are used in *pattern*, then the text of all groups in the pattern are also returned as part of the resulting list. If *maxsplit* is nonzero, at most *maxsplit* splits occur, and the remainder of the string is returned as the final element of the list.
- Example
  - >>> re.split('\W+', 'Words, words, words.')
    - ['Words', 'words', 'words', '']
  - >>> re.split('(\W+)', 'Words, words, words.')
    - ['Words', ', ', 'words', ', ', 'words', '.', '']
  - >>> re.split('\W+', 'Words, words, words.', 1)
    - ['Words', 'words, words.']

Module “re”

- To get all matches from a string, call re.**findall**(regex, subject). This will return an array of all non-overlapping regex matches in the string. "Non-overlapping" means that the string is searched through from left to right, and the next match attempt starts beyond the previous match.

- More efficient than re.findall() is re.**finditer**(regex, subject). It returns an iterator that enables you to loop over the regex matches in the subject string: for m in re.finditer(regex, subject). The for-loop variable m is a Match object with the details of the current match.

- re.**sub**(regex, replacement, subject) performs a search-and-replace across subject, replacing all matches of regex in subject with replacement. The result is returned by the sub() function. The subject string you pass is not modified.

Module “sys”

Variables and functions for the Python interpreter

- sys.argv
  - List of command-line arguments; sys.argv[0] is script name
- sys.path
  - List of directory names, where modules are searched for
- sys.platform
  - String to identify type of computer system

```
>>> import sys
>>> sys.platform
'win32'
```

Module “sys”

- sys.stdout, sys.stdin, sys.stderr
  - Predefined file objects for input/output
  - 'print' stuff goes to 'sys.stdout'
  - May be set to other files
- sys.exit(n)
  - Force exit from Python execution
  - 'n' is an integer error code, normally 0

```
>>> import sys
>>> sys.stdout.write('the hard way')
the hard way
```

Module “os”

Portable interface to operating-system services

- os.getcwd()
  - Returns the current directory

```
>>> os.getcwd()
'M:\\My Documents\\Python course\\tests'
```

- os.environ
  - Dictionary containing the current environment variables

```
>>> for k, v in os.environ.items(): print k, v

TMP C:\DOCUME~1\se22312\LOCALS~1\Temp
COMPUTERNAME WS101778
USERDOMAIN BIOVITRUM
COMMONPROGRAMFILES C:\Program Files\Common Files
PROCESSOR_IDENTIFIER x86 Family 6 Model 9 Stepping 5, GenuineIntel
PROGRAMFILES C:\Program Files
PROCESSOR_REVISION 0905
HOME C:\emacs
...
```

Module “os”

- os.chdir(path)
  - Changes the current working directory to 'path'
- os.listdir(path)
  - Return a list of the contents of the directory 'path'

- os.mkdir(path)
  - Create the directory 'path'
- os.rmdir(path)
  - Remove the directory 'path'

- os.remove(path)
  - Remove the file named 'path'

Module “os”

- os.system(command)
  - Execute the shell command (string) in a subprocess
  - Return the error code as integer
- os.popen(command, mode='r')
  - Run the shell command (string)
  - Open a pipe to the command, return as a file object
  - Mode is either read, or write; not both
- os.popen2, os.popen3, os.popen4
  - Variants of os.popen, with different file objects
- os.getpid()
  - Return the process ID as integer

Module “os”

Portable path name handling

- os.path.abspath(path)
  - Returns the absolute path for the given relative 'path'

```
>>> d = os.path.abspath('.')
>>> d
'M:\\My Documents\\Python course\\tests'
```

- os.path.dirname(path)
  - Returns the directory name part of 'path'

```
>>> os.path.dirname(d)
'M:\\My Documents\\Python course'
```

Module “os”

- os.path.join(path, path, …)
  - Joint together the path parts intelligently into a valid path name

```
>>> d = os.path.join(os.getcwd(), 't1.py')
>>> d
'M:\\My Documents\\Python course\\tests\\t1.py'
```

- os.path.split(path)
  - Splits up the path into directory name and filename
  - Reverse of 'os.path.join'

```
>>> os.path.split(d)
('M:\\My Documents\\Python course\\tests', 't1.py')
```

- os.path.splitext(path)
  - Splits up the path into base filename and the extension (if any)

```
>>> >>> os.path.splitext(d)
('M:\\My Documents\\Python course\\tests\\t1', '.py')
```

Module “os”

- os.path.exists(path)
  - Does the 'path' exist? File, or directory

```
>>> d = os.path.join(os.getcwd(), 't1.py')
>>> os.path.exists(d)
True
```

- os.path.isfile(path)
  - Is 'path' the name of a file?
- os.path.isdir(path)
  - Is 'path' the name of a directory?

```
>>> os.path.isfile(d)
True
>>> os.path.isdir(d)
False
```

- os.path.walk(path, func, arg)
  - Used to process an entire directory tree
  - Visits each subdirectory
  - Calls the function 'func' with a list of the filenames in that directory

Module “datetime”

- # dates are easily constructed and formatted
  - >>> from datetime import date
  - >>> now = date.today()
  - >>> now
  - datetime.date(2003, 12, 2)
  - >>> now.strftime("%m-%d-%y or %d%b %Y is a %A on the %d day of %B")
  - ’12-02-03 or 02Dec 2003 is a Tuesday on the 02 day of December’
- # dates support calendar arithmetic
  - >>> birthday = date(1964, 7, 31)
  - >>> age = now - birthday
  - >>> age.days
  - 14368

Assignment

- Write a program which creates a back up of files
  - The files and directories to be backed up are specified in a list
  - The back up must be stored in main back up directory
  - The files are backed up to a zip file
  - The name of the zip archive is the current date and time

Queries

"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

Acknowledgements

- www.python.org
- http://docs.python.org/tutorial/
- http://www.tutorialspoint.com/python/python_reg_expressions.htm

NALANDA

"The contents here are for Aricent internal training purposes only and do not carry any commercial value"

ARICENT™

Disclaimer

"Aricent makes no representations or warranties with respect to contents of these slides and the same are being provided "as is". The content/materials in the slides are of a general nature and are not intended to address the specific circumstances of any particular individual or entity. The material may provide links to internet sites (for the convenience of users) over which Aricent has no control and for which Aricent assumes no responsibility for the availability or content of these external sites. While the attempt has been to acknowledge sources of materials wherever traceable to an individual or an institution; any materials not specifically acknowledged is purely unintentional"

Thank you