

# The Python Language (Part 5)

Copyright © 2025 by  
Robert M. Dondero, Ph.D.  
Princeton University

# Objectives

- We will cover:
  - A subset of Python...
  - That is appropriate for COS 333...
  - Through example programs

# Agenda

- **Prelim: character encodings**
- Files
- The “with” statement
- Arrays
- Associative arrays

# Character Encodings

- ***Unicode***
  - Maps each character to a number
- ***Character encoding***
  - Maps each such number to the byte(s) which represent it

# Character Encodings

Common character encodings:

Encoding	Fixed/ Variable Width	Bytes
<b><i>ASCII</i></b>	Fixed	1 (7 bits)
<b><i>Latin-1</i></b>	Fixed	1
<b><i>UCS-2</i></b>	Fixed	2
<b><i>UTF-32</i></b>	Fixed	4
<b><i>UTF-8</i></b>	Variable	1, 2, 3, or 4

# Character Encodings

Character	Unicode	ASCII	Latin-1	UCS-2	UTF-32	UTF-8
<b>space</b>	<b>0020</b>	20	20	0020	00000020	20
<b>!</b>	<b>0021</b>	21	21	0021	00000021	21
<b>0</b>	<b>0030</b>	30	30	0030	00000030	30
<b>A</b>	<b>0041</b>	41	41	0041	00000041	41
<b>a</b>	<b>0061</b>	61	61	0061	00000061	61
<b>a with grave</b>	<b>00e0</b>		e0	00e0	000000e0	e0
<b>Greek small pi</b>	<b>03c0</b>			03c0	000003c0	cf80
<b>Double prime</b>	<b>2033</b>			2033	00002033	e280b3
<b>Aegean number 2000</b>	<b>10123</b>				00010123	f09084a3

# Aside: Python str Class

- Python `str` class
  - An object of class `str` is a sequence of 0 or more characters
  - Internal encoding: Latin-1 | UCS-2 | UTF-32

Example	Python Literal	Internally
abc	'abc'	61 62 63
abπc	'ab\u03c0c'	00 61 00 62 03 c0 00 63

# Aside: Python bytes Class

- Python `bytes` class
  - An object of class `bytes` is a sequence of 0 or more **bytes**
  - Internal encoding: **None**

Example	Python Literal	Internally
61 62 63	<code>b'\x61\x62\x63'</code>	61 62 63
61 62 63	<code>b'abc'</code>	61 62 63



# Agenda

- Prelim: character encodings
- **Files**
- The “with” statement
- Arrays
- Associative arrays

# Files

- See **copybytes.py**

```
$ cat file1
to be
or not to be
that is
the question
$ python copybytes.py file1 file2
$ cat file2
to be
or not to be
that is
the question
$
```

# Files

- See **copystings.py**

```
$ cat file1
to be
or not to be
that is
the question
$ python copystings.py file1 file2
$ cat file2
to be
or not to be
that is
the question
$
```

# Agenda

- Prelim: character encodings
- Files
- **The “with” statement**
- Arrays
- Associative arrays

# The “with” Statement

- Consider copystrings.py
- **Problem:**

```
in_file = open(...)
...
...
...
in_file.close()
```

Exception thrown =>  
File never explicitly  
closed

# The “with” Statement

- **Solution 1:**

- `try...finally` statement

```
in_file = open(...)
```

```
try:
```

```
    ...
```

```
    ...
```

```
    ...
```

```
finally:
```

```
    in_file.close()
```

If this is entered...

Then this certainly  
will be executed “on  
the way out”

# The “with” Statement

- See **copystingsfinally.py**

```
$ cat file1
to be
or not to be
that is
the question
$ python copystingsfinally.py file1 file2
$ cat file2
to be
or not to be
that is
the question
$
```

# The “with” Statement

- **Solution 2:**
  - `with` statement

```
with open(...) as in_file:
```

```
... ] If this is entered, then  
... ] in_file.close() certainly will  
... ] be executed on the way out
```



# The “with” Statement

- See **[copystingswith.py](#)**

```
$ cat file1
to be
or not to be
that is
the question
$ python copystingswith.py file1 file2
$ cat file2
to be
or not to be
that is
the question
$
```

# Agenda

- Prelim: character encodings
- Files
- The “with” statement
- **Arrays**
- Associative arrays

# Arrays

- Generic term: **array**
- Python term: *list*
  - A dynamically expanding (doubling) array of object references

# Arrays

- List fundamentals:

```
$ python
>>> a = ['Ruth', 'Gehrig', 'Mantle']
>>> a[1]
'Gehrig'
>>> a.append('Jeter')
>>> a
['Ruth', 'Gehrig', 'Mantle', 'Jeter']
>>> 'Gehrig' in a
True
>>> 'Berra' in a
False
>>> quit()
$
```

# Arrays

- Python term: *tuple*
  - An immutable list

# Arrays

- Tuple fundamentals:

```
$ python
>>> t = ('Ruth', 'Gehrig', 'Mantle')
>>> t[1]
'Gehrig'
>>> t.append('Jeter')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no
attribute 'append'
>>> 'Gehrig' in t
True
>>> 'Berra' in t
False
>>> quit()
$
```

# Arrays

- See [linesort.py](#)

```
$ cat file1
to be
or not to be
that is
the question
$ python linesort.py < file1
or not to be
that is
the question
to be
$
```

```
$ cat pride8.txt
...
$ python linesort.py < pride8.txt
...
$
```

Elapsed wall clock time: **1.0 sec**

# Arrays

- See **linesorttim.py**

```
$ cat file1
to be
or not to be
that is
the question
$ python linesorttim.py < file1
or not to be
that is
the question
to be
$
```

```
$ cat pride8.txt
...
$ python linesorttim.py < pride8.txt
...
$
```

Elapsed wall clock time: **1.5 sec**



# Agenda

- Prelim: character encodings
- Files
- The “with” statement
- Arrays
- **Associative arrays**

# Associative Arrays

- Generic term: **associative array**
- Python term: *dict*
  - An associative array implemented as a hash table

# Associative Arrays

## Dict fundamentals:

```
$ python
>>> aa = {'Ruth':3, 'Gehrig':4, 'Mantle':7}
>>> aa['Gehrig']
4
>>> aa.get('Gehrig')
4
>>> aa['Jeter']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Jeter'
>>> aa.get('Jeter')
>>> aa.get('Jeter', 2)
2
>>> aa['Jeter'] = 2
>>> aa
{'Ruth': 3, 'Gehrig': 4, 'Mantle': 7, 'Jeter': 2}
>>> 'Gehrig' in aa
True
>>> 'Berra' in aa
False
>>> quit()
$
```

# Associative Arrays

- See **concord.py**

```
$ cat file1
to be
or not to be
that is
the question
$ python concord.py < file1
to: 2
be: 2
or: 1
not: 1
that: 1
is: 1
the: 1
question: 1
$
```

```
$ cat pride8.txt
...
$ python concord.py < pride8.txt
chapter: 488
i: 16632
it: 12304
is: 6872
a: 15672
truth: 216
universally: 24
acknowledged: 160
...
whittingham: 8
co: 8
took: 8
chancery: 8
$
```

# Associative Arrays

- See [concord.py](#) (cont.)

Arg to re.compile()	Meaning
' [a-z] '	Used to find a lower-case letter.
' [a-z] + '	Used to find a sequence of 1 or more lower-case letters.
r ' [a-z] + '	A raw string literal. Backslash in string literal has no special meaning. Here unnecessary.

# Associative Arrays

- See [concord.py](#) (cont.)

```
$ python concord.py < pride8.txt | sort -n -k 2 -r |  
head -10  
the: 34880  
to: 33456  
of: 29040  
and: 28736  
her: 17840  
i: 16632  
a: 15672  
in: 15040  
was: 14784  
she: 13704  
$
```

Elapsed wall clock time: 0.77 sec

# For More Information

For more information:

```
$ python
>>> help(str)
>>> help(bytes)
>>> help(tuple)
>>> help(list)
>>> help(set)
>>> help(dict)
>>> quit()
$
```

# Python Commentary

**Date:** Sun, 1 Jan 2006 13:18:08 -0800  
**From:** Rob 'Commander' Pike <r@google.com>  
**To:** Brian Kernighan <bwk@CS.Princeton.EDU>

python is a very easy language. i think it's actually a good choice for some things. awk is perfect for a line or two, python for a page or two. both break down badly when used on larger examples, although python users utterly refuse to admit its weaknesses for large-scale programming, both in syntax and efficiency.

-rob



# Summary

- We have covered these aspects of Python:
  - Files
  - Arrays
  - Associative arrays

# Summary

- We have covered:
  - Subset of Python...
  - That is appropriate for COS 333...
  - Through example programs
- See also:
  - **Appendix 1:** Iterable Classes
  - **Appendix 2:** Variadic Functions/Methods
  - **Appendix 3:** Duck Typing
  - **Appendix 4:** Regular Expressions
  - **Appendix 5:** The Python Debugger

# Appendix 1: Iterable Classes

# Iterable Classes

- Python iterable classes

Class	Description of Object of that Class
<code>str</code>	An immutable sequence of characters
<code>bytes</code>	An immutable sequence of bytes
<code>list</code>	A sequence of object references
<code>tuple</code>	An immutable sequence of object references
<code>set</code>	A collection of object references that contains no duplicate references
<code>dict</code>	An associative array of object references implemented as a hash table
<code>file</code>	A persistent sequence of bytes

# Iterable Classes

## Creating iterable objects

### **str**

```
str1 = 'hi'  
str2 = "hi"  
str3 = r'hi' # raw string
```

### **bytes**

```
bytes1 = b'hi'  
bytes2 = b"hi"  
bytes3 = rb'hi'
```

### **list**

```
list1 = [obj1, obj2, ...]
```

### **tuple**

```
tuple1 = (obj1, obj2, ...)  
tuple2 = (obj1, ) ← hack
```

# Iterable Classes

## Creating iterable objects (cont.)

### **set**

```
set1 = {obj1, obj2, ...}  
# tests for object ref equality
```

### **dict**

```
dict1 = {keyobj1:valueobj1, keyobj2:valueobj2, ...}
```

### **file**

```
fileobj = open('filename', mode='somemode')  
# somemode: r, rb, w, wb, ...
```

# Iterable Classes

## Some str Methods

<code>str1 = str2.__add__(str3)</code>	<code># str1 = str2 + str3</code>
<code>bool1 = str1.__eq__(str2)</code>	<code># bool1 = str1 == str2</code>
<code>bool1 = str1.__ne__(str2)</code>	<code># bool1 = str1 != str2</code>
<code>bool1 = str1.__lt__(str2)</code>	<code># bool1 = str1 &lt; str2</code>
<code>bool1 = str1.__gt__(str2)</code>	<code># bool1 = str1 &gt; str2</code>
<code>bool1 = str1.__le__(str2)</code>	<code># bool1 = str1 &lt;= str2</code>
<code>bool1 = str1.__ge__(str2)</code>	<code># bool1 = str1 &gt;= str2</code>
<code>int1 = str1.__len__()</code>	<code># int1 = len(str1)</code>
<code>str1 = str2.__getitem__(int1)</code>	<code># str1 = str2[int1]</code>
<code>bool1 = str1.__contains__(str2)</code>	<code># bool1 = str2 in str1</code>

# Iterable Classes

## Some str Methods (cont.)

```
bool1 = str1.startswith(str2)
```

```
bool1 = str1.endswith(str2)
```

```
bool1 = str1.isspace()
```

```
bool1 = str1.isalnum()
```

```
bool1 = str1.isalpha()
```

```
bool1 = str1.isdecimal()
```

```
bool1 = str1.isdigit()
```

```
bool1 = str1.islower()
```

```
bool1 = str1.isnumeric()
```

```
bool1 = str1.isupper()
```

```
str1 = str2.upper()
```

```
str1 = str2.lower()
```



# Iterable Classes

## Some str Methods (cont.)

```
list1 = str1.split(str2)
```

```
str1 = str2.replace(str3, str4)
```

```
str1 = str2.strip()
```

```
str1 = str2.lstrip()
```

```
str1 = str2.rstrip()
```

```
int1 = str1.find(str2)
```

```
int1 = str1.rfind(str2)
```

```
str1 = str2.join(list1) (See note)
```

```
bytes1 = str1.encode(encoding)
```

### Note:

```
'/'.join(['hello', 'there', 'world']) =>  
'hello/there/world'
```

# Iterable Classes

## Some list Methods

<code>list1 = list2.__add__(list3)</code>	<code># list1 = list2 + list3</code>
<code>bool1 = list1.__contains__(obj1)</code>	<code># <b>bool1 = obj1 in list1</b></code>
<code>list1.__delitem__(int1)</code>	<code># del(list1[int1])</code>
<code>obj1 = list1.__getitem__(int1)</code>	<code># <b>obj1 = list1[int1]</b></code>
<code>list1.__iadd__(list2)</code>	<code># list1 += list2</code>
<code>int1 = list1.__len__()</code>	<code># <b>int1 = len(list1)</b></code>
<code>list1.__setitem__(int1, obj1)</code>	<code># <b>list1[int1] = obj1</b></code>

# Iterable Classes

## Some list Methods (cont.)

```
list1.append(obj1)
```

```
list1.clear()
```

```
list1 = list2.copy()
```

```
int1 = list1.index(obj1)
```

```
list1.insert(obj1, int1)
```

```
obj1 = list1.pop()
```

```
list1.remove(obj1)
```

```
list1.reverse()
```

```
list1.sort()
```

# Iterable Classes

## Some dict Methods

```
bool1 = dict1.__contains__(obj1) # bool1 = obj1 in dict1
```

```
dict1.__delitem__(obj1) # del(dict1[obj1])
```

```
obj1 = dict1.__getitem__(obj2) # obj1 = dict1[obj2]
```

```
int1 = dict1.__len__() # int1 = len(dict1)
```

```
dict1.__setitem__(obj1, obj2) # dict1[obj1] = obj2
```

```
dict1.clear()
```

```
dict1 = dict2.copy()
```

```
list1 = dict1.keys()
```

```
list1 = dict1.items()
```

```
list1 = dict1.values()
```

# Iterable Classes

## Some file Methods

```
file1.close()
```

```
file1.flush()
```

```
str1 = file1.read()
```

```
bool1 = file1.readable()
```

```
str1 = file1.readline()
```

```
list1 = file1.readlines()
```

```
fool1 = file1.writable()
```

```
file1.write(str1)
```

```
file1.writelines(list1)
```

# Appendix 2: Variadic Functions/Methods

# Variadic Functions/Methods

- **Variadic** function/method:
  - A function/method that can be called with a variable number of arguments
  - **Example:** `printf()` in C
    - `printf("hello");`
    - `printf("The answer is %d", 5);`
    - `printf("The answers are %d and %d", 5, 10);`

# Variadic Functions/Methods

- **Question**
  - How to define variadic functions/methods in Python?



# Variadic Functions/Methods

- **Answer 1**
  - Default parameter values
    - Already described

```
def sub(minuend, subtrahend=0):  
    return minuend - subtrahend  
  
def main():  
    difference = sub(5, 2)  
    difference = sub(5)  
    ...  
if __name__ == '__main__':  
    main()
```

# Variadic Functions/Methods

- **Answer 2**
  - `*args` and `**kwargs...`

# Variadic Functions/Methods

- See [variadic.py](#)

```
$ python variadic.py  
a  
b  
c  
d  
key1 e  
key2 f  
$
```

# Variadic Functions/Methods

- See [variadic.py](#) (cont.)

Parameter	Referenced Object
<code>i</code>	<code>'a'</code>
<code>j</code>	<code>'b'</code>
<code>args</code>	<code>['c', 'd']</code>
<code>kwargs</code>	<code>{'key1': 'e', 'key2': 'f'}</code>

# Appendix 3: Duck Typing

# Duck Typing

- Observation:
  - Python uses *duck typing*

“When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”

-- James Whitcomb Riley

# Duck Typing

- Example: Recall **euclid.py**
  - `i` and `j` parameters of `gcd()` can reference objects of any class, as long as they can be:
    - Operands of `==`
    - Arguments to `abs()`
    - Operands of `!=`
    - Operands of `%`

# Duck Typing

Language	Object references have types?	Objects have types?	Language classification
C	yes	no	<b>weakly</b> typed
Java	yes	yes	<b>strongly</b> typed
Python	no	yes	<b>dynamically (duck)</b> typed



# Duck Typing

- Is duck typing good or bad (vs. strong typing) ?
- See **euclidstrong.py**
  - Which is better, euclid.py or euclidstrong.py?

# Duck Typing

- **Style 1:** Don't validate parameter types
  - Validating parameter types is constraining and slow
  - **So euclid.py is better**
- **Style 2:** Validate parameter types
  - Validating parameter types is safe
  - **So euclidstrong.py is better**
- We'll use Style 1

# Duck Typing

- Commentary
  - **Small** projects:
    - Maybe need not validate parameter types
  - **Large** projects:
    - Maybe should validate parameter types

# Duck Typing

- Commentary
  - But if you feel the need to validate parameter types, then why are you using Python???

# Appendix 4: Regular Expressions

# Regular Expressions

- Used widely
  - Java (string manipulation)
  - Python (string manipulation)
  - Unix `grep` command (file searching)
  - Bash shell (filename wildcards)
  - SQL `like` clauses (querying databases)
    - See upcoming *Database Programming* lectures
  - ...

# Regular Expressions

**RE**

thing  
^thing  
thing\$  
^thing\$  
^  
^\$  
.  
thing.\$  
thing\.\$  
\\thing\\  
[tT]hing  
thing[0-9]  
thing[^0-9]  
thing[0-9][^0-9]  
thing1.\*thing2  
^thing1.\*thing2\$

**Matches**

thing anywhere in string  
thing at beginning of string  
thing at end of string  
string that contains only thing  
any string, even empty  
empty string  
non-empty, i.e. the first char in string  
thing plus any char at end of string  
thing. at end of string  
\\thing\\ anywhere in string  
thing or Thing anywhere in string  
thing followed by one digit  
thing followed by a non-digit  
thing followed by digit, then non-digit  
thing1 then any (or no) text then thing2  
thing1 at beginning and thing2 at end

# Regular Expressions

- What do these match?
  - `a.*e.*i.*o.*u`
    - Try with `grep` command and `/usr/share/dict/words` file
  - `^[^aeiou]*a[^aeiou]*e[^aeiou]*i[^aeiou]*o[^aeiou]*u[^aeiou]*$`
    - Try with `grep` command and `/usr/share/dict/words` file

Thanks to Prof. Brian Kernighan



# Regular Expressions

- Implementations vary
  - See *Mastering Regular Expressions* (Jeffrey Friedl) book
    - ~500 pages!
- In Python...

# Regular Expressions

<b>RE</b>	<b>Matches</b>
X	the character X, except for metacharacters
\X	the character X, where X is a metacharacter
.	any character except \n (use DOTALL as argument to compile() to match \n too)
^	start of string
\$	end of string
XY	X followed by Y
X*	zero or more cases of X (X*? is the same, but non-greedy)
X+	one or more cases of X (X+? is the same, but non-greedy)
X?	zero or one case of X (X?? is the same, but non-greedy)
[...]	any one of ...
[^...]	any character other than ...
[X-Y]	any character in the range X through Y
X Y	X or Y
(...)	..., and indicates a group

Precedence: \* + ? higher than concatenation, which is higher than |

# Regular Expressions

## **RE**   **Matches**

```
\t  tab
\v  vertical tab
\n  newline
\r  return
\f  form feed
\a  alert
\e  escape
\\  backslash
\A  empty string at start of given string
\b  empty string, but only at start or end of a word
\B  empty string, but not at start or end of a word
\d  a digit
\D  a non-digit
\s  a white space character, that is, [\t\n\r\f\v]
\S  a non-white space character
\w  an alphanumeric character, that is, [a-zA-Z0-9_]
\W  a non-alphanumeric character
\Z  the empty string at the end of the given string
```

# Regular Expressions

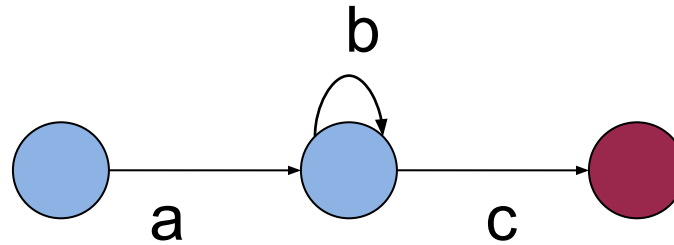
- What kinds of strings do these regular expressions match?
  - `[-+]?([0-9]+\.[0-9]*|\.[0-9]+)([Ee][-+]?[0-9]+)?`
  - `/\.*.*?\*/` (use with `DOTALL`)
    - Why the question mark?
    - Why `DOTALL`?
- **Commentary: Regular expressions are write-only!!!**

# Regular Expressions

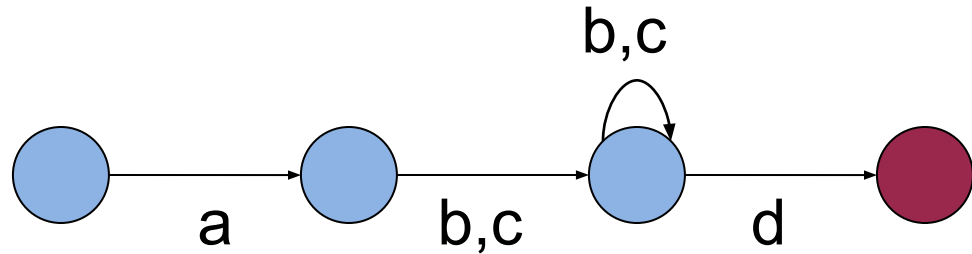
- Some theory:
  - Regular expressions have the same power as *deterministic finite state automata* (DFAs)
  - A regular expression defines a *regular language*
  - A DFA also defines a regular language

# Regular Expressions

$ab^*c$



$a[bc]^+d$



# Appendix 5: The Python Debugger

# The Python Debugger

- `pdb` debugger is bundled with Python
- To use `pdb`:

```
$ python -m pdb somefile.py
```



# The Python Debugger

## Some `pdb` Commands

- `help`
- `break functionOrMethod`
- `break filename:linenum`
- `run`
- `list`
- `next`
- `step`
- `continue`
- `print expr`
- `where`
- `quit`

Note similarities with `gdb`

# The Python Debugger

- Common commands have abbreviations:  
h, b, r, l, n, s, c, p, w, q
- Blank line means repeat the same command
- Beware: Cannot easily read from stdin