LECTURES WEEK 1

AGENDA

week	subject	book	week	subject	book
1	Python features	1	3	any & all	19
	running Python	1		range, zip & enumerate	12
	dynamic binding	2		higher-order functions	16
	Python statements	1		classes and OOP	15..18
	printing stuff	2,3		exceptions	14
	Python types	1		assert	16
	numbers	1		file access	14
	strings	8		working with CSV and JSON	-
	control statements	7		coding style	-
	lists	10			
2	tuples	12	4	case: word histogram	13
	dictionaries	11		recursion	5
	sets	19		case: solving Numbrix	-
	functions	6		PySerial	-
	scope/visibility	11		tkinter GUI-toolkit	-
	comprehension	19		web-programming	
	modules and packages	14			

COURSE OUTLINE

- Python's core data model

- implementing complex algorithms

7	8		4			1	2	
6				7	5			9
			6		1		7	8
		7		4		2	6	
		1		5		9	3	
9		4		6				5
	7		3				1	2
1	2				7	4		
	4	9	2		6			7

AGENDA

- **Python features**
- running Python
- dynamic binding
- Python statements
- printing stuff
- Python types
- numbers
- strings
- control statements
- lists


Goog

PYTHON FEATURES

- a general purpose, interpreted, high-level programming language
  - esp. system programming, internet scripting and scientific programming
  - open source
  - maybe the most versatile and capable all-rounder
  - see www.python.org/about/success and wiki "List_of_Python_software"

PYTHON FEATURES

- the Python "ecosystem" is very large
  - a very large standard library
  - over 115 thousand packages in PyPI
  - many web & GUI frameworks
  - many frameworks for "scientific programming" aka "data science"

PYTHON FEATURES

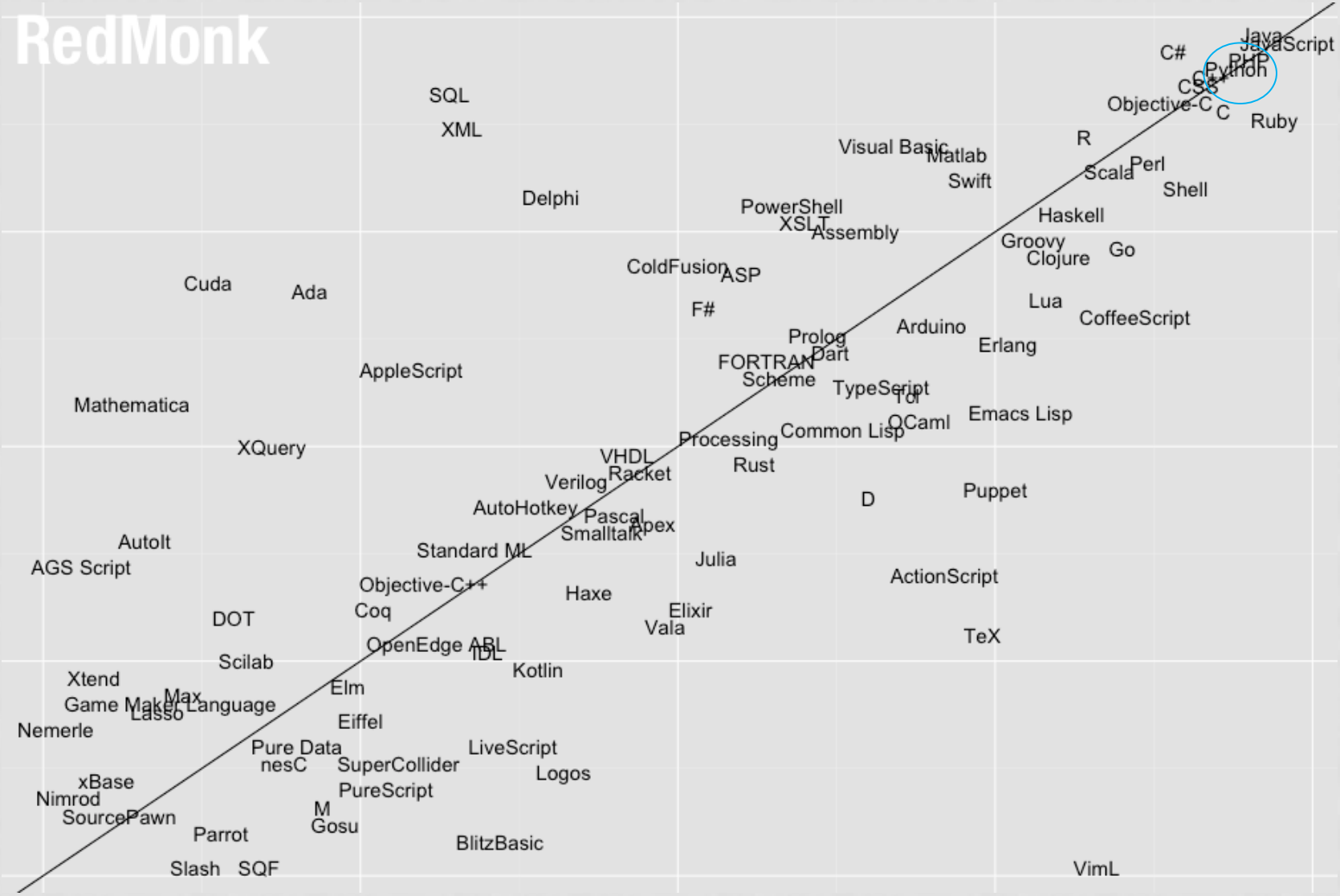
- created by Guido van Rossum in the early '90's at CWI (Amsterdam)
  - "a descendant of ABC that would appeal to Unix/C hackers"
- CPython: the reference implementation of Python
  - free and open-source, maintained by the Python Software Foundation
  - written in portable ISO C, compiles and runs on virtually every major platform currently in use

PYTHON FEATURES

- uses dynamic typing
- multi-paradigm : object-oriented, imperative, functional, procedural, reflective
- automatic memory management (garbage collection)
- current release version is 3.6
  - version 3.x is **not** compatible with 2.x
  - we'll use the latest version 3.6.2

PYTHON FEATURES

- "pythonic": conforms with Python's minimalist philosophy and emphasis on readability
  - intended to be easy to learn
  - but it has grown at least as complex as other languages
- development time vs. run-time :
  - Python gives you a rapid development cycle
  - Python's main downside is performance
  - you can mix Python with libraries coded in languages such as C or C++

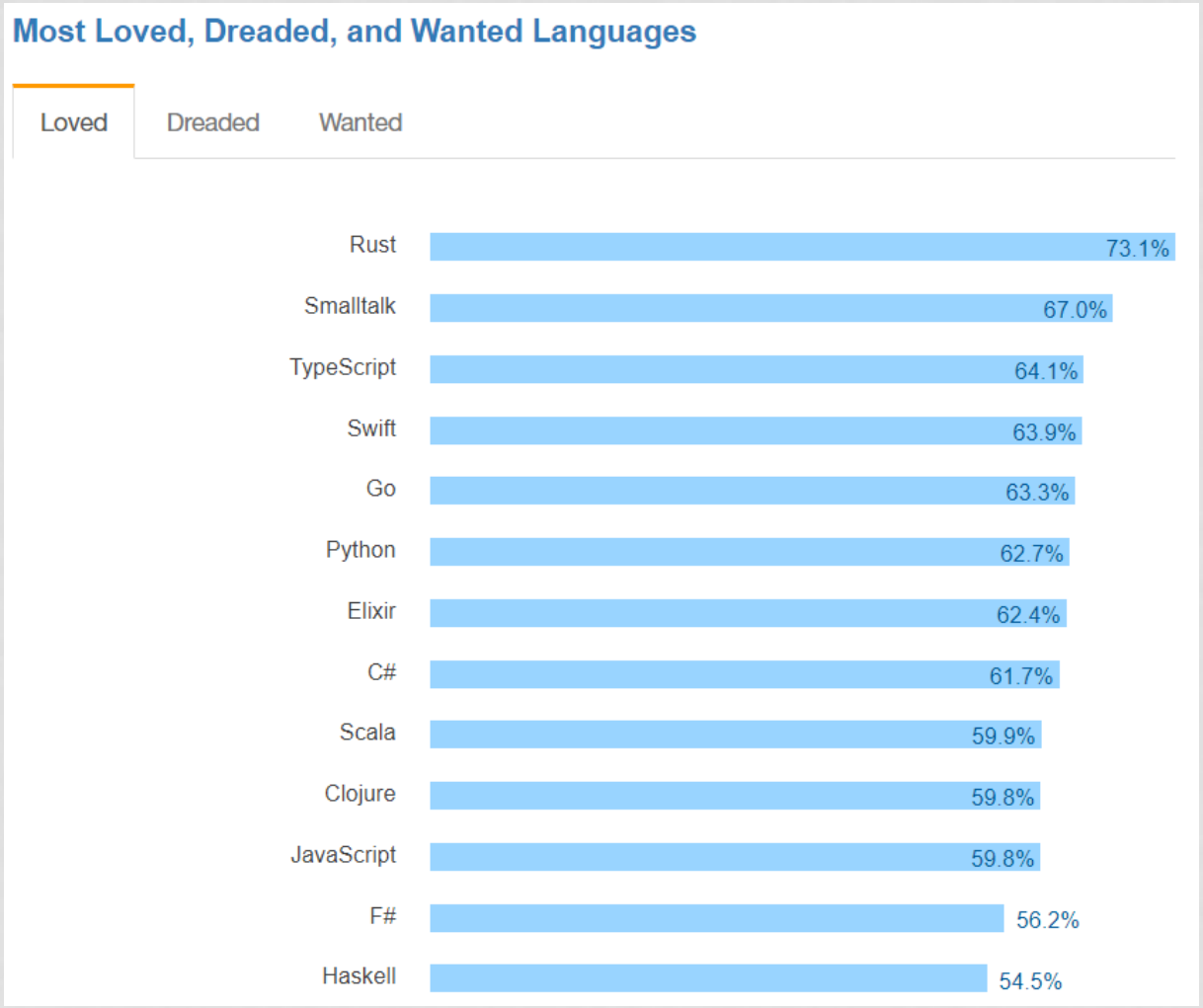


v = StackOverflow
h = Github

TIOBE INDEX

Sep 2017	Sep 2016	Change	Programming Language
1	1		Java
2	2		C
3	3		C++
4	4		C#
5	5		Python
6	7	^	PHP
7	6	v	JavaScript
8	9	^	Visual Basic .NET
9	10	^	Perl
10	12	^	Ruby
11	18	^^	R
12	11	v	Delphi/Object Pascal
13	13		Swift
14	17	^	Visual Basic
15	8	vv	Assembly language
16	15	v	MATLAB

STACKOVERFLOW SURVEY 2017



COMPILER

- translates source code to machine code, but does not execute the source code
- translation at compile time takes a lot of time, but execution at run time is fast
- does some memory management e.g. register allocation and code optimization

INTERPRETER

- translates source code into an immediate language ('bytecode') and immediately execute this
- interpreting a program is much slower than executing native machine code
  - interprets (=executes) one line at a time from the source file
  - interpreter must analyze each statement in the program each time it is executed and then perform the desired action - ~100 times slower
- development generally much quicker
  - edit-interpret-debug vs. edit-compile-run-debug

A SIDE-BY-SIDE COMPARISON OF COMPILED LANGUAGES AND INTERPRETED LANGUAGES

Upwork™

A look at how compilers and interpreters work, and how their differences affect memory, runtime speed, and computer workload.

	A COMPILER	AN INTERPRETER
Input	... takes an entire program as its input.	... takes a single line of code, or instruction, as its input.
Output	... generates intermediate object code.	... does not generate any intermediate object code.
Speed	... executes faster.	... executes slower.
Memory	... requires more memory in order to create object code.	... requires less memory (doesn't create object code).
Workload	... doesn't need to compile every single time, just once.	... has to convert high-level languages to low-level programs at execution.
Errors	... displays errors once the entire program is checked.	... displays errors when each instruction is run.

PYTHON VS JAVA

- static & dynamic typing
  - Java : all variable must be explicitly declared
  - Python : you never declare anything; an assignment statement binds a name to an object
- container objects
  - Java : collections can only hold objects, not primitives such as int
    - ~~ArrayList<int> intList = new ArrayList<int>();~~
  - Python : container objects (e.g. lists and dictionaries) can hold objects of any type

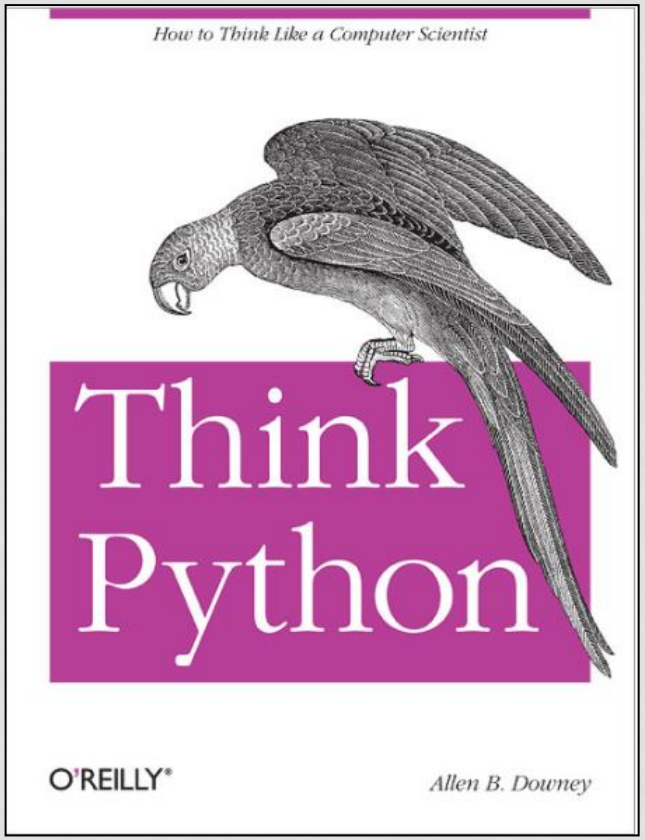
```python
# most_common_words.py
from collections import Counter

num_words = 10
with open('astronaut.txt', 'r') as f:
    counter = Counter(word.lower()
                      for line in f
                      for word in line.strip().split()
                      if word)

for word, count in counter.most_common(num_words):
    print(count, word)
```

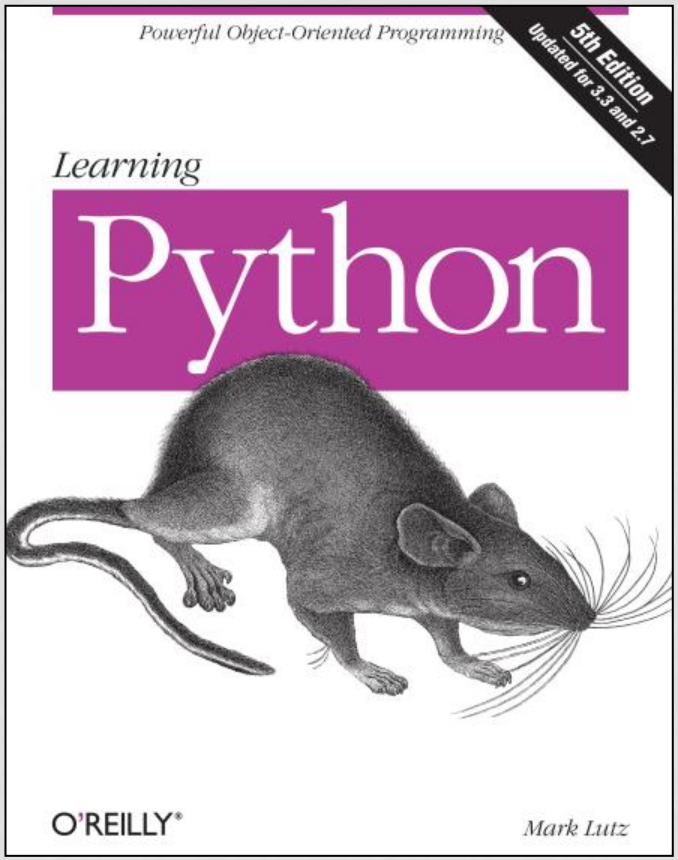
```
> python most_common_words.py
38 the
36 and
32 a
30 to
28 in
23 o
23 i
22 of
21 t
18 n
```

WE'LL USE



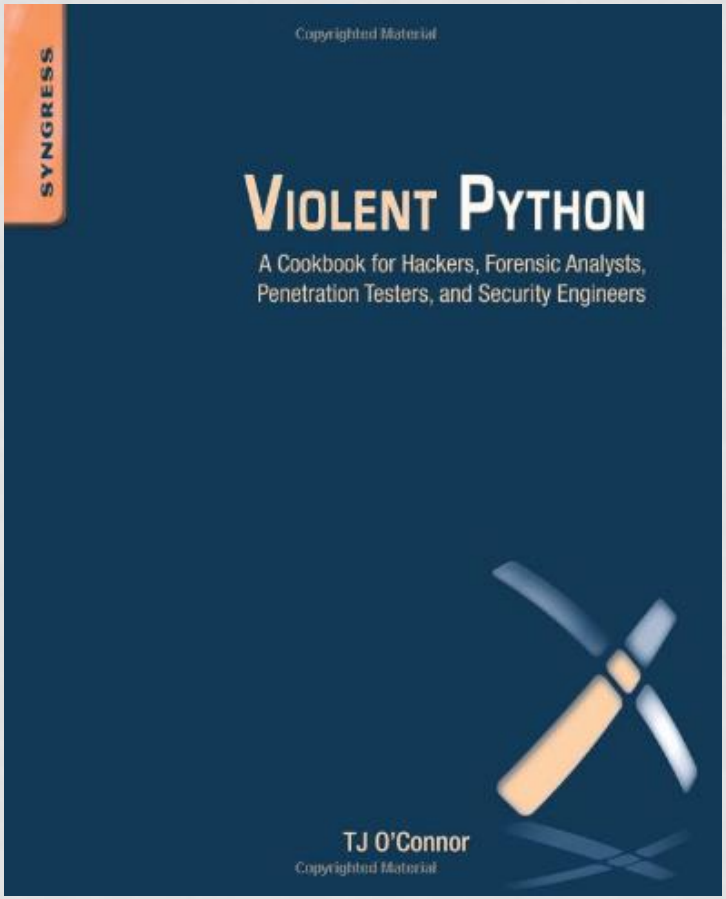
free download:
http://greenteapress.com/wp/think-python-2e/

ANOTHER GOOD BOOK



and many many more, for example
http://shop.oreilly.com/category/browse-subjects/programming/python.do

FORENSIC INVESTIGATION



RESOURCES

- and many, many resources on the web
- The Python Tutorial :
  - https://docs.python.org/3/tutorial/
- Googles Python is another
  - https://developers.google.com/edu/python
- and the Python documentation is here
  - https://docs.python.org/3
- and of course
  - StackOverflow

AGENDA

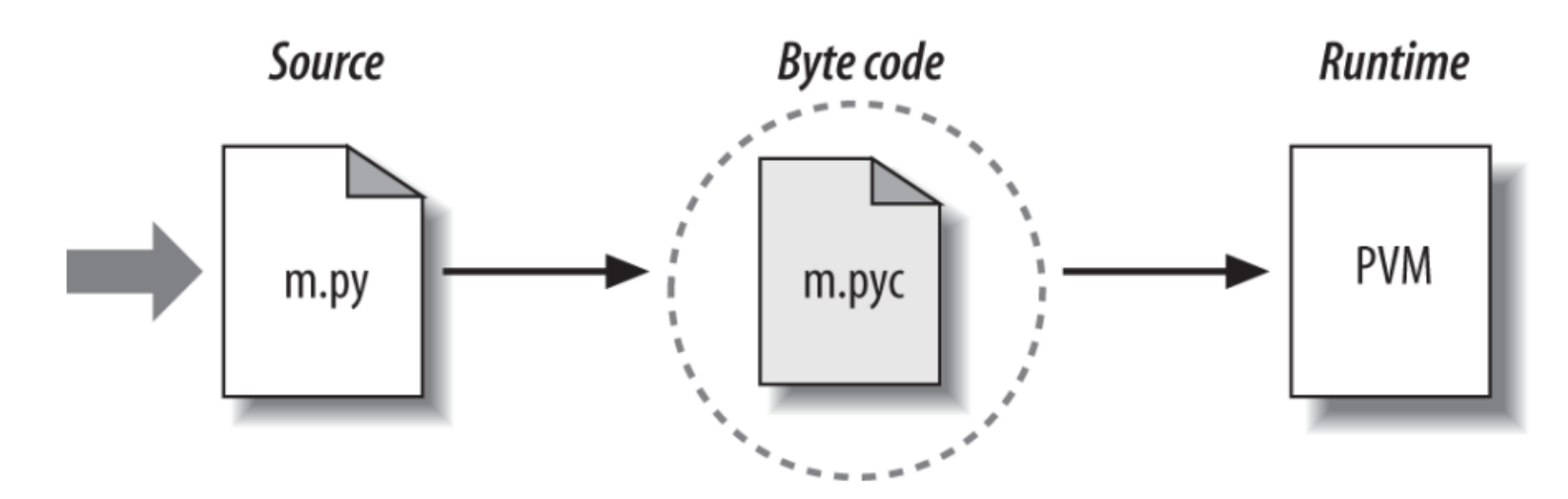
- Python features
- **running Python**
- dynamic binding
- Python statements
- printing stuff
- Python types
- numbers
- strings
- control statements
- lists

RUNNING PYTHON

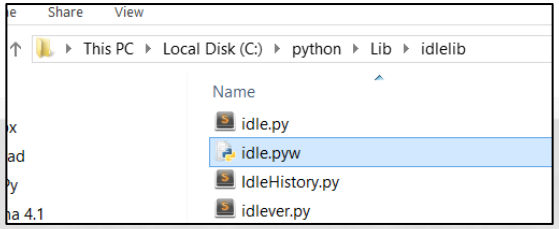
- download installer from www.python.org
  - we'll use latest version 3.6
  - probably already installed on Linux and Mac
- running python :
  - $ python hello.py
- interactive prompt
  - $ python
  - >>> run python commands interactively
- Windows :
  - check $path environment var in "advanced system settings"

RUNNING PYTHON

- interpreter converts source code into byte code
- byte code is stored in .pyc-files (in `__pycache__`)
- PVM = Python VM

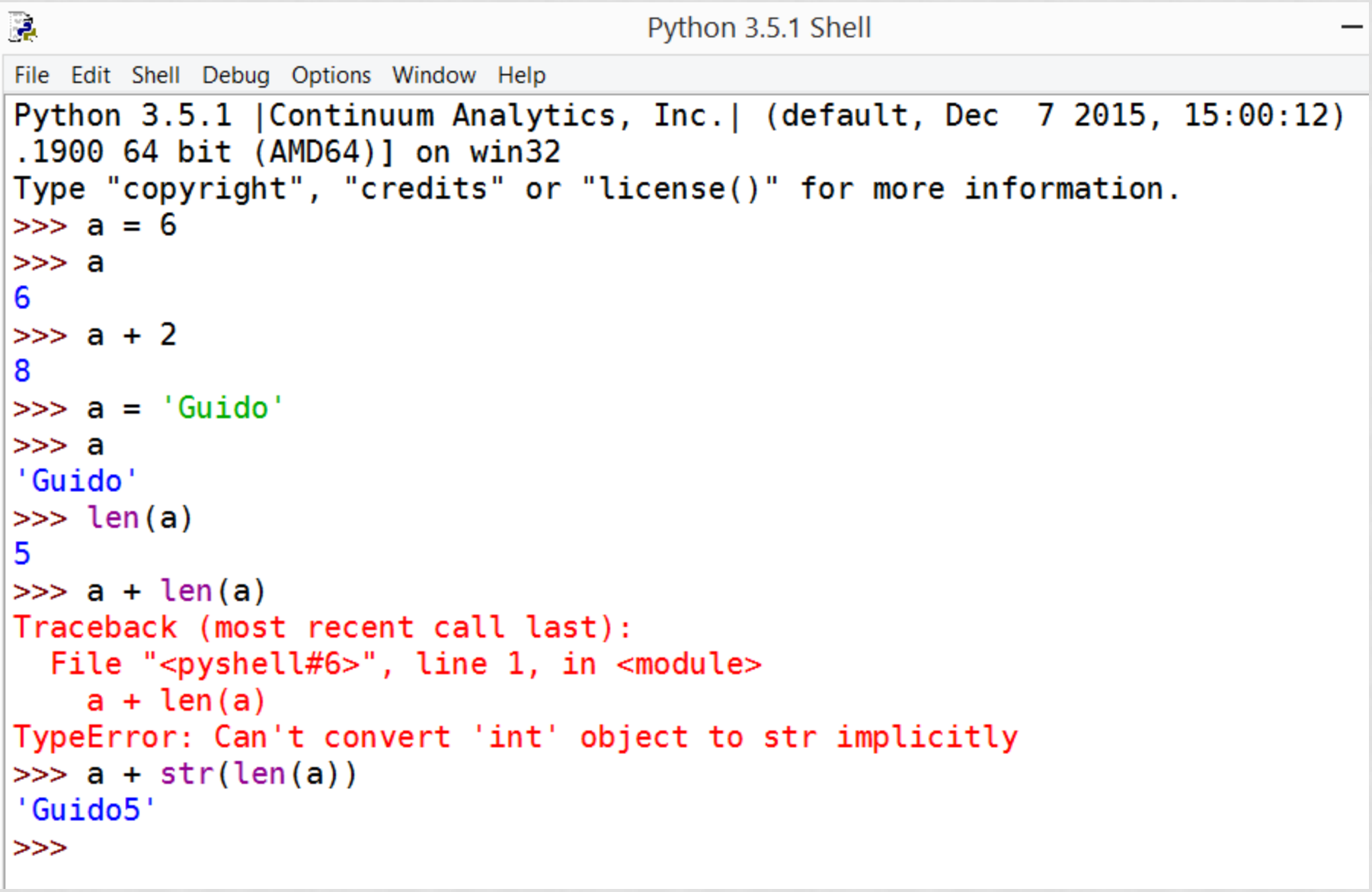


IDLE



- IDLE for interactive Python
- invoke idle.pyw somewhere under \Lib\idlelib (Windows)
- previous & next command with alt-p, alt-n

- other options :
- your favorite editor and a Windows terminal
  - e.g. Sublime Text + ConEmu
- PyCharm (has a nice debugger with GUI)


Python 3.5.1 Shell
File Edit Shell Debug Options Window Help
Python 3.5.1 |Continuum Analytics, Inc.| (default, Dec 7 2015, 15:00:12)
.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> a = 6
>>> a
6
>>> a + 2
8
>>> a = 'Guido'
>>> a
'Guido'
>>> len(a)
5
>>> a + len(a)
Traceback (most recent call last):
File "<pyshell#6>", line 1, in <module>
a + len(a)
TypeError: Can't convert 'int' object to str implicitly
>>> a + str(len(a))
'Guido5'
>>>

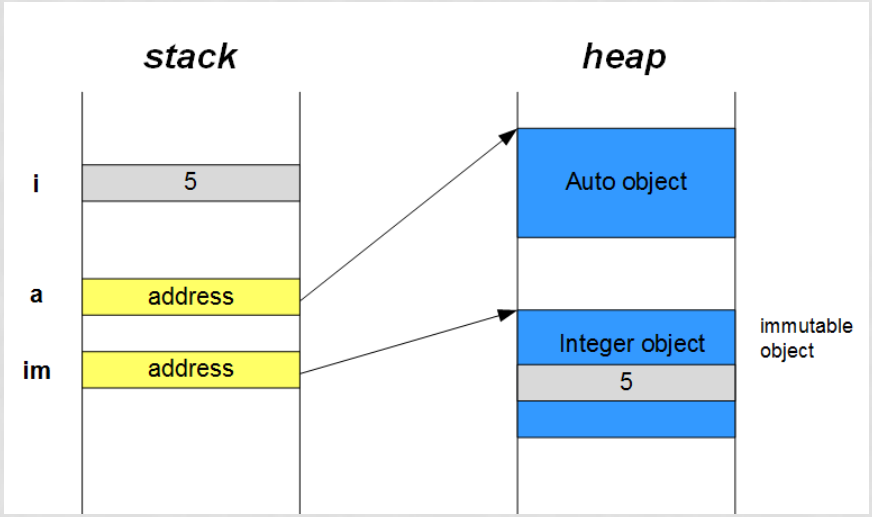
AGENDA

- Python features
- running Python
- **dynamic binding**
- Python statements
- printing stuff
- Python types
- numbers
- strings
- control statements
- lists

STATIC TYPING IN JAVA

- C, C#, C++, Java : variables have a type, and declaring a variable means reserving memory for that type

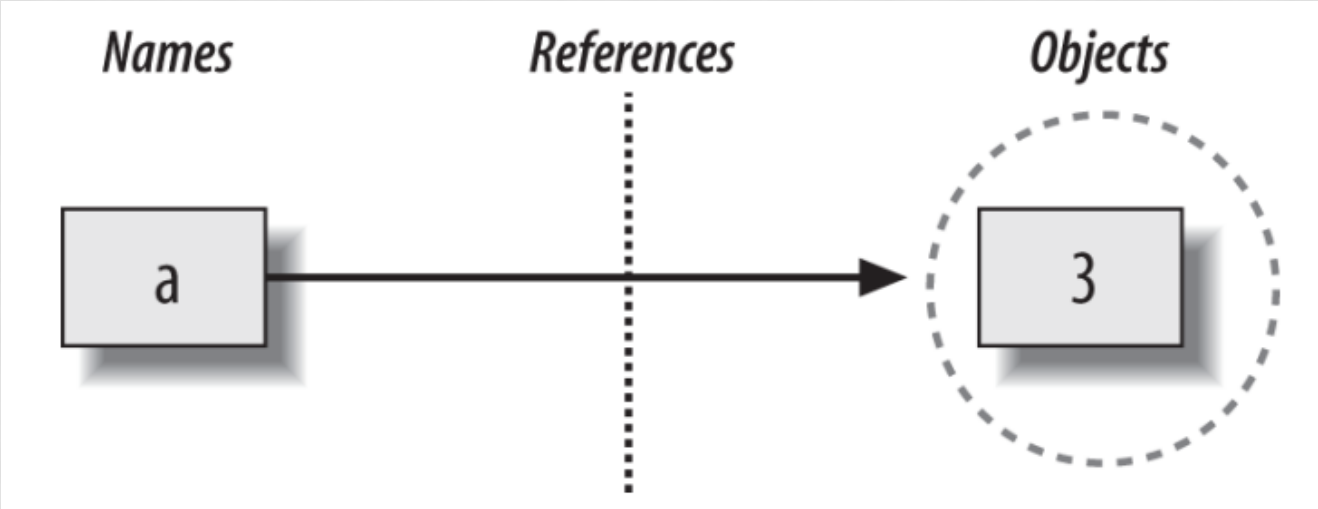
```
int i = 5;
Integer im = new Integer(5);
Auto a = new Auto();
```



DYNAMIC TYPING

- variables are just names and can reference any type of object
- an assignment statement binds a name to an object
  - variables must be assigned before they can be referenced

DYNAMIC TYPING




```
>>> a = 3          # It's an integer
>>> a = 'spam'     # Now it's a string
>>> a = 1.23       # Now it's a floating point
```

AGENDA

- Python features
- running Python
- dynamic binding
- **Python statements**
- printing stuff
- Python types
- numbers
- strings
- control statements
- lists

STATEMENTS

- end-of-line is end of statement (no ; required)
- end of indentation is end of block (no {} required)
  - this makes your code more readable
  - use 4 spaces for indentation (PEP8: 1 tab = 4 spaces)
- all compound statements have a header line terminated in a colon
- parentheses are optional

```
if x > y:
    x = 1
    y = 2
```

instead of

```
if (x > y) {
    x = 1;
    y = 2;
}
```

STATEMENTS

- statements may span multiple lines if you're using a pair of brackets : (), {}, or []

```
L = ["Good",
     "Bad",
     "Ugly"]                                    # Open pairs may span lines
```

```
if (A == 1 and
    B == 2 and
    C == 3):
        print('spam' * 3)
```

STATEMENTS

Operation	Interpretation
`spam = 'Spam'`	Basic form
`spam, ham = 'yum', 'YUM'`	Tuple assignment (positional)
`[spam, ham] = ['yum', 'YUM']`	List assignment (positional)
`a, b, c, d = 'spam'`	Sequence assignment, generalized
`a, *b = 'spam'`	Extended sequence unpacking (Python 3.X)
`spam = ham = 'lunch'`	Multiple-target assignment
`spams += 42`	Augmented assignment (equivalent to `spams = spams + 42`)

AGENDA

- Python features
- running Python
- dynamic binding
- Python statements
- **printing stuff**
- Python types
- numbers
- strings
- control statements
- lists

PRINTING STUFF

- old C-style:

```
>>> print('%s: %-.4f, %05d' % ('Result', 3.14159, 42))
Result: 3.1416, 00042
```

- modern Python:

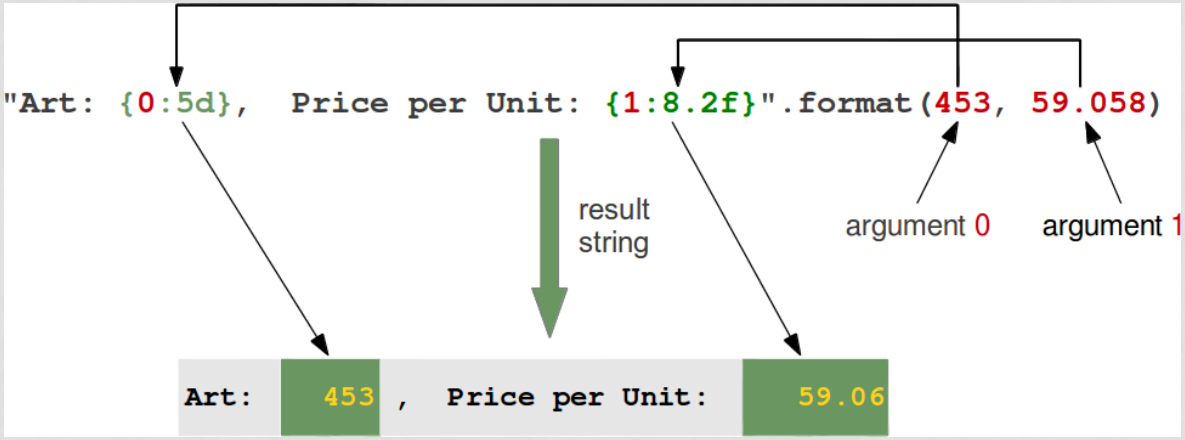
```
>>> 'That is {0} {1} bird!'.format(1, 'dead')
'That is 1 dead bird!'
>>> print('That is {0} {1} bird!'.format(1, 'dead'))
That is 1 dead bird!
>>>
```

USING FORMAT SPECIFIERS

```
1 print('First argument: {0}, second one: {1}'.format(47,11))
2 print('First argument: {}, second one: {}'.format(47,11))
3 print('First argument: {:.2f}, second one: {:.2f}'.format(1.234, 5.678))
4 print('Various precions: {0:6.2f} or {0:6.3f}'.format(1.4148))
5 print('{0:<20s} {1:6.2f}'.format('Spam & Eggs:', 6.99))
6 print('{0:>20s} {1:6.2f}'.format('Spam & Eggs:', 6.99))
7 print('{0:>20} {1:6.2f}'.format('Spam & Ham:', 7.99))
8 print('Total amount is : {0:>08d}'.format(59832))
9 print('Interest is : {0:>10.4f}'.format(5.4))
```

```
First argument: 47, second one: 11
First argument: 47, second one: 11
First argument: 1.23, second one: 5.68
Various precions:   1.41 or  1.415
Spam & Eggs:            6.99
        Spam & Eggs:    6.99
         Spam & Ham:    7.99
Total amount is : 00059832
Interest is :     5.4000
```

USING FORMAT SPECIFIERS



The general form of a *standard format specifier* is:

```
format_spec ::=  [[fill]align][sign][#][0][width][,][.precision][type]
fill        ::=  <any character>
align       ::=  "<" | ">" | "=" | "^"
sign        ::=  "+" | "-" | " "
width       ::=  integer
precision   ::=  integer
type        ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%"
```

https://docs.python.org/3/library/string.html#format-string-syntax

SEPARATORS

- print() displays a blank line
- end defines the final character
- sep defines a separator

For example, the following code

```
1  print("AAA", end = ' ')
2  print("BBB", end = '')
3  print("CCC", end = '***')
4  print("DDD", end = '***')
```

displays

```
AAA BBBCCC***DDD***
```

```
>>> x = 'ham'
>>> y = 999
>>> z = ['eggs']
>>> print(x, y, z, sep=', ') #custom separator
ham, 999, ['eggs']
>>>
```

AGENDA

- Python features
- running Python
- dynamic binding
- Python statements
- printing stuff
- **Python types**
- numbers
- strings
- control statements
- lists

PYTHON'S BUILT-IN TYPES

Object type	Example literals/creation
Numbers	`1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()`
Strings	`'spam', "Bob's", b'a\x01c', u'sp\xc4m'`
Lists	`[1, [2, 'three'], 4.5], list(range(10))`
Dictionaries	`{'food': 'spam', 'taste': 'yum'}, dict(hours=10)`
Tuples	`(1, 'spam', 4, 'U'), tuple('spam'), namedtuple`
Files	`open('eggs.txt'), open(r'C:\ham.bin', 'wb')`
Sets	`set('abc'), {'a', 'b', 'c'}`
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes
Implementation-related types	Compiled code, stack tracebacks

MUTABLE VS. IMMUTABLE

- numbers, strings and tuples are immutable
  - every string assignment results in a new string
  - every number assignment results in a new number
- lists, dictionaries and sets are mutable
  - they can be changed in place after they are created

QUIZ: TRUE OR FALSE?

a)
```
s = 'abc'
s[0] = 'x'
s == 'xbc'
```

b)
```
s = 'abc'
t = 'abc'
s == t
s is t
```

c)
```
L = [1,2,3]
L[0] = 7
L == [7,2,3]
```

d)
```
L = [1,2,3]
M = [1,2,3]
M == L
M is L
```

```
>>> s = 'abc'
>>> s[0]
'a'
>>> s[0] = x
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    s[0] = x
NameError: name 'x' is not defined
>>> id(s)
5930528
>>> s = 'z' + s
>>> id(s)
44691264
>>> s
'zabc'
>>> L = [1,2,3]
>>> id(L)
49384344
>>> L[0] = 7
>>> id(L)
49384344
>>> L
[7, 2, 3]
>>> x = 5
>>> id(x)
1409959632
>>> x = 7
>>> id(x)
1409959664
```

AGENDA

- Python features
- running Python
- dynamic binding
- Python statements
- printing stuff
- Python types
- **numbers**
- strings
- control statements
- lists

NUMERIC TYPES

Literal	Interpretation
`1234, -24, 0, 99999999999999`	Integers (unlimited size)
`1.23, 1., 3.14e-10, 4E210, 4.0e+210`	Floating-point numbers
`0o177, 0x9ff, 0b101010`	Octal, hex, and binary literals in 3.X
~~`0177, 0o177, 0x9ff, 0b101010`~~	Octal, octal, hex, and binary literals in 2.X
`3+4j, 3.0+4.0j, 3J`	Complex number literals
`set('spam'), {1, 2, 3, 4}`	Sets: 2.X and 3.X construction forms
`Decimal('1.0'), Fraction(1, 3)`	Decimal and fraction extension types
`bool(X), True, False`	Boolean type and constants

TRUE OR FALSE

- booleans: True and False
- zero numbers, empty objects, and object None are evaluated as False
  - None
  - 0 and 0.0
  - [ ] an empty list
  - ( ) an empty tuple
  - { } an empty dict
  - ' ' an empty string
  - set() an empty set
- any nonzero number or nonempty object is True

```
>>> False == 0
True
>>> True == 1
True
>>> if 7: print(True)

True
>>> if []: print(True)

>>> if [1]: print(True)

True
```

NUMBERS

- no ++ operator

```
>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)
(2.5, 2.5, -2.5, -2.5)
>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2) # floor division
(2, 2.0, -3.0, -3)
```

```
>>> import random
>>> random.random()
0.35320656434077446
>>> random.choice(['Life of Brian', 'Holy Grail', 'Meaning of Life'])
'Life of Brian'
```

```
>>> import math
>>> math.pi, math.e # common constants
(3.141592653589793, 2.718281828459045)
>>> math.sqrt(144), math.sqrt(2)
(12.0, 1.4142135623730951)
>>> pow(2, 4), 2 ** 4, 2.0 ** 4.0
(16, 16, 16.0)
>>> min(3, 1, 2, 4), max(3, 1, 2, 4)
(1, 4)
>>>
```

AGENDA

- Python features
- running Python
- dynamic binding
- Python statements
- printing stuff
- Python types
- numbers
- **strings**
- control statements
- lists

Operation	Interpretation
`S = ''`	Empty string
`S = "spam's"`	Double quotes, same as single
`S = 's\np\ta\x00m'`	Escape sequences
`S = """...multiline..."""`	Triple-quoted block strings
`S = r'\temp\spam'`	Raw strings (no escapes)
`B = b'sp\xc4m'`	Byte strings
`U = u'sp\u00c4m'`	Unicode strings
`S1 + S2`	Concatenate, repeat
`S * 3`	
`S[i]`	Index, slice, length
`S[i:j]`	
`len(S)`	
`"a %s parrot" % kind`	String formatting expression
`"a {0} parrot".format(kind)`	String formatting method in 2.6, 2.7, and 3.X
`S.find('pa')`	String methods (see ahead for all 43): search,
`S.rstrip()`	remove whitespace,
`S.replace('pa', 'xx')`	replacement,
`S.split(',')`	split on delimiter,

Operation	Interpretation
`S.isdigit()`	content test,
`S.lower()`	case conversion,
`S.endswith('spam')`	end test,
`'spam'.join(strlist)`	delimiter join,
`S.encode('latin-1')`	Unicode encoding,
`B.decode('utf8')`	Unicode decoding, etc.
`for x in S: print(x)`	Iteration, membership
`'spam' in S`	
`[c * 2 for c in S]`	
`map(ord, S)`	
`re.match('sp(.*)am', line)`	Pattern matching: library module

STRING EXAMPLES

- single- and double-quoted strings are the same

```
>>> myjob = 'hacker'
>>> for c in myjob: print(c, end=' ')

h a c k e r
>>> 'k' in myjob
True
>>> "k" in myjob
True
>>> 'z' in myjob
False
```

SPLIT

- gegeven string s = 'bob,hacker,40', hoe maak je hiervan ['bob', 'hacker', '40'] ?

```
>>> s
'bob,hacker,40'
>>> s.split(',')
['bob', 'hacker', '40']
```

JOIN

- gegeven L = ['x','y','z'] hoe krijg je de string 'xyz' ?

```
>>> "".join(['x','y','z'])
'xyz'
```

- gegeven L = ['a','b','c'] hoe krijg je de string 'a,b,c' ?

```
>>> ",".join(["a", "b", "c"])
'a,b,c'
```

INDEXING AND SLICING

[*start:end*]
Indexes refer to places the knife "cuts."

Defaults are beginning of sequence and end of sequence.

SEQUENCES

- strings, lists and tuples are all sequences
- all sequences support slicing: `L[start:stop]`
- elements of sequence have an order
- dictionaries are not sequences, elements have no order

INDEXING AND SLICING

```
>>> S = 'spam'
>>> S[0], S[-2]                    # Indexing from front or end
('s', 'a')
>>> S[1:3], S[1:], S[:-1]          # Slicing: extract a section
('pa', 'pam', 'spa')
```

```
>>> L = [1,2,3,4,5,6,7]
>>> L[3:5]
[4, 5]
>>> L[:3]
[1, 2, 3]
>>> L[5:]
[6, 7]
>>> L[:-2]
[1, 2, 3, 4, 5]
```

```
>>> L = [1,2,3]
>>> M = L
>>> M is L
True
>>> M = L[:]
>>> M is L
False
```

MORE SLICING

- some sequences also support "extended slicing" with a third "step" parameter: `L[start:stop:step]`
  - step < 0 : direction is from right to left

```
>>> s = 'bicycle'
>>> s[::3]
'bye'
>>> s[::-1]
'elcycib'
>>> s[::-2]
'eccb'
```

QUIZ

- Gegeven een willekeurige string s
- Welke van onderstaande is (altijd) gelijk aan s?

a) s[:]
b) s + s[0:-1+1]
c) s[0:]
d) s[:-1]
e) s[:3] + s[3:]

AGENDA

- Python features
- running Python
- dynamic binding
- Python statements
- printing stuff
- Python types
- numbers
- strings
- **control statements**
- lists

CONTROL STATEMENTS

if/elif/else	Selecting actions	`if "python" in text:` `  print(text)`
for/else	Iteration	`for x in mylist:` `  print(x)`
while/else	General loops	`while X > Y:` `  print('hello')`
pass	Empty placeholder	`while True:` `  pass`
break	Loop exit	`while True:` `  if exittest(): break`
continue	Loop continue	`while True:` `  if skiptest(): continue`

NO CASE STATEMENT

- no case statement, but elif is short for "else if"

```
>>> x = 'killer rabbit'
>>> if x == 'roger':
...     print("shave and a haircut")
... elif x == 'bugs':
...     print("what's up doc?")
... else:
...     print('Run away! Run away!')
...
Run away! Run away!
```

FOR

- a for loop can step across any kind of sequence object

```
for target in object:                # Assign object items to target
    statements
    if test: break                   # Exit loop now, skip else
    if test: continue                # Go to top of loop now
else:
    statements                       # If we didn't hit a 'break'
```

EXAMPLE FOR

```
1  sum = 0
2  for x in [1,2,3,4]:
3      sum = sum + x
4  print(sum)
5
6  sum = 0
7  for x in range(4): # 0..3
8      sum = sum + x
9  print(sum)
10
11 D = {'a': 1, 'b' : 2 ,'c' : 3}
12 for key in D:
13     print(key, "=>", D[key])
```

```
10
6
a => 1
b => 2
c => 3
```

WHILE

```
while test:
    statements
    if test: break          # Exit loop now, skip else if present
    if test: continue       # Go to top of loop now, to test
else:
    statements              # Run if we didn't hit a 'break'
```

- **break** : jumps out of the (enclosing) loop
- **continue** : jumps to the loop's condition again
- **while ... else** is like a normal loop exit

EXAMPLE WHILE

```
1  x = list(range(7))
2  while x:                    # while x not empty
3      if x[0] == 5:           # some random test
4          print('Match found')
5          break               # we're done, skip else
6      x = x[1:]
7  else:
8      print('Not found')  # only here if x == []
9
```

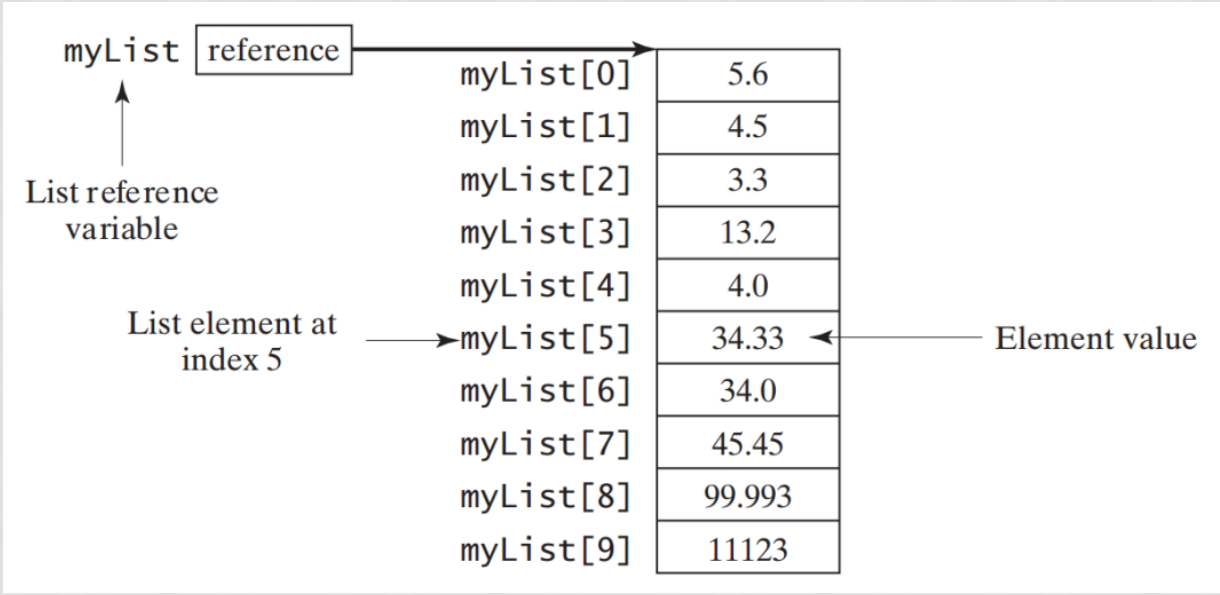
PRIME NUMBER?

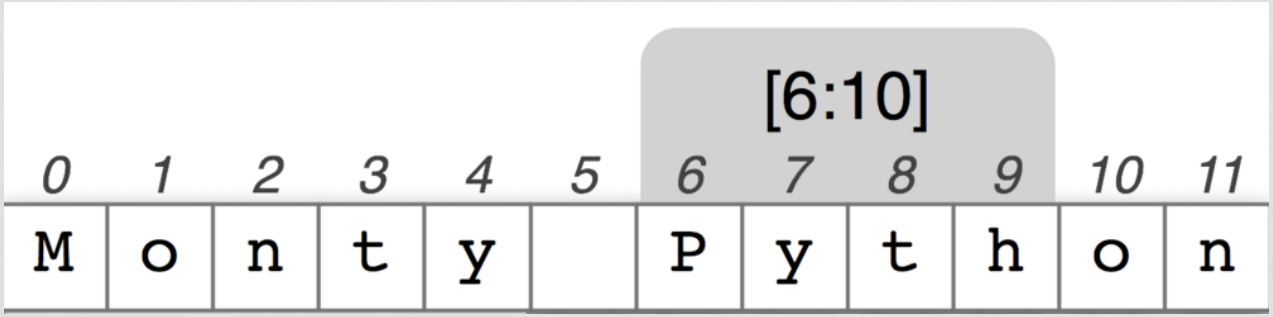
```
 1  reply = input("Please enter number: ")
 2  y = int(reply)
 3  x = y // 2                    # only have to test 1..y/2
 4  while x > 1:
 5      if y % x == 0:
 6          print(y, 'has factor', x)
 7          break                 # we're done, skip else
 8      x -= 1                    # test next one
 9  else:
10      print(y, 'is prime')
```

AGENDA

- Python features
- running Python
- dynamic binding
- Python statements
- printing stuff
- Python types
- numbers
- strings
- control statements
- **lists**

LISTS


myList
reference
List reference variable
myList[0]
myList[1]
myList[2]
myList[3]
myList[4]
myList[5]
myList[6]
myList[7]
myList[8]
myList[9]
5.6
4.5
3.3
13.2
4.0
34.33
34.0
45.45
99.993
11123
List element at index 5
Element value


[6:10]
0 1 2 3 4 5 6 7 8 9 10 11
M o n t y P y t h o n

```
>>> len([1, 2, 3])
3
>>> [1, 2, 3] + [4, 5, 6] # concatenation
[1, 2, 3, 4, 5, 6]
>>> 3 in [1, 2, 3] # membership
True
>>>
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[2]
'SPAM!'
>>> L[-2]
'Spam'
>>>
>>> for x in [1, 2, 3]:print(x, end=' ')

1 2 3
>>>
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> matrix[1]
[4, 5, 6]
>>> matrix[1][1]
5
>>> matrix[2][0]
7
```

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[1] = 'eggs'
>>> L
['spam', 'eggs', 'SPAM!']
>>> L = ['eat', 'more', 'SPAM!']
>>> L.append('please')
>>> L
['eat', 'more', 'SPAM!', 'please']
>>> L.sort()
>>> L
['SPAM!', 'eat', 'more', 'please']
>>> L.pop()
'please'
>>> L.count('more')
1
>>> L[1:3]
['eat', 'more']
>>>
```

```
>>> L = [0] + 10*[1]
>>> L
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>>
```

LIST OPERATIONS

Operation	Interpretation
`L = []`	An empty list
`L = [123, 'abc', 1.23, {}]`	Four items: indexes 0..3
`L = ['Bob', 40.0, ['dev', 'mgr']]`	Nested sublists
`L = list('spam')`	List of an iterable's items, list of successive integers
`L = list(range(-4, 4))`	
`L[i]`	Index, index of index, slice, length
`L[i][j]`	
`L[i:j]`	
`len(L)`	
`L1 + L2`	Concatenate, repeat

Operation	Interpretation
`L * 3`	
`for x in L: print(x)`	Iteration, membership
`3 in L`	
`L.append(4)`	Methods: growing
`L.extend([5,6,7])`	
`L.insert(i, X)`	
`L.index(X)`	Methods: searching
`L.count(X)`	
`L.sort()`	Methods: sorting, reversing,
`L.reverse()`	copying (3.3+), clearing (3.3+)
`L.copy()`	
`L.clear()`	
`L.pop(i)`	Methods, statements: shrinking
`L.remove(X)`	
`del L[i]`	
`del L[i:j]`	
`L[i:j] = []`	
`L[i] = 3`	Index assignment, slice assignment
`L[i:j] = [4,5,6]`	
`L = [x**2 for x in range(5)]`	List comprehensions and maps
`list(map(ord, 'spam'))`	

OPGAVE

- Schrijf een programma dat twee gesorteerde lijsten samenvoegt naar een nieuwe gesorteerde lijst.
- Aangezien de lijsten niet even lang hoeven te zijn zal je moeten kijken naar de lengte van beide lijsten.
- Je mag geen gebruik maken van een build-in sorteerfunctie zoals sort().

```python
# nb. functies zijn nog niet besproken in week 1
def merge(list1, list2):
    result = []

    idx1 = 0 # Current index in list1
    idx2 = 0 # Current index in list2

    while idx1 < len(list1) and idx2 < len(list2):
      if list1[idx1] < list2[idx2]:
          result.append(list1[idx1])
          idx1 += 1
      elif list1[idx1] > list2[idx2]:
          result.append(list2[idx2])
          idx2 += 1
      else:
          # same value in both lists
          result.append(list1[idx1])
          result.append(list2[idx2])
          idx1 += 1
          idx2 += 1

    while idx1 < len(list1):
        result.append(list1[idx1])
        idx1 += 1

    while idx2 < len(list2):
        result.append(list2[idx2])
        idx2 += 1

    return result
```

```
32  # do some checks
33  l1 = []
34  l2 = [2,4]
35  l3 = merge(l1, l2)
36  assert l3 == l2
37
38  l1 = [1,3]
39  l2 = []
40  l3 = merge(l1, l2)
41  assert l3 == l1
42
43  l1 = [0,0,1,1,2,3]
44  l2 = [1,4,5,6]
45  l3 = merge(l1, l2)
46  assert l3 == [0,0,1,1,1,2,3,4,5,6]
47
48  print("The merged list is: ", end = "")
49  for e in l3:
50      print(e, end = " ")
51  print()
```