Python

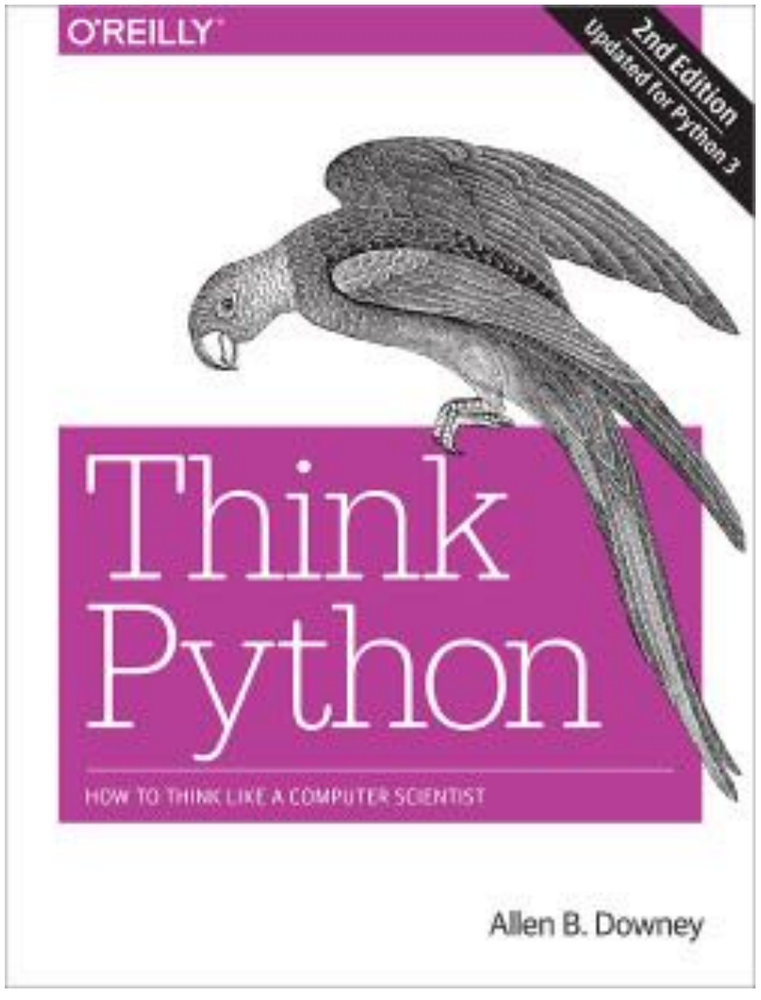
- Invented ~1990 by Guido von Rossum
- Now one of the most widely used programming languages in the world
- Python3 (2008) is mildly backwards-incompatible version
  - Now used for new developments
  - But lots of Python2 code remains
- Pythonic = "in proper Python style"
- Pythonista = "Python fan"

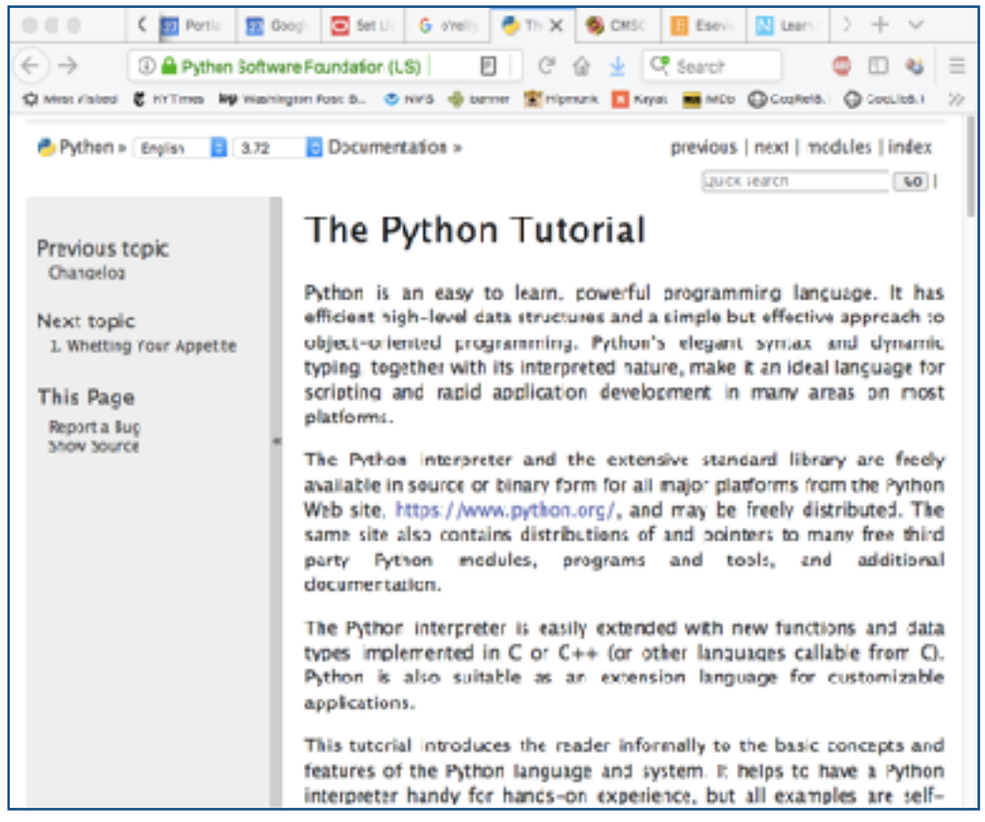
Python2: `print x`
Python3: `print(x)`

Python3: all strings use unicode

Free Python Resources

- There are dozens of websites and books
- Be careful to choose resources that cover Python3





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

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

Python: The Read-Eval-Print-Loop

be sure to specify python3

```
$ python3
Python 3.7.2 (default, Dec 27 2018, 07:35:45)
[Clang 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or
"license" for more information.
>>> 2+2
4
>>> x = 15 + 6
>>> y = x * 2
>>> y
42
>>> print('x + y =', x+y)
x + y = 63
>>> ^D
$
```

evaluate an expression

bind a variable to a value

use the value of a variable

display a value

print values of one or more expressions

Python: Batch execution of programs

comment

file containing python statements

```
$ cat example.py
# same statements as we typed in before
2+2
x = 15+6
y = x * 2
y
print('x + y =', x+y)
$ python3 example.py
x + y = 63
$
```

top-level expressions are evaluated invisibly

reads and executes file

print() results go to stdout

Python Values and Types

- Every value is an object of some class
  - Conceptually it lives in a box and is handled by reference
  - No distinction between objects and primitive values
- Every value has a type. Built-in types include
  - int (in Python3, these are unbounded integers)
    - boolean (subtype of int with values True and False)
  - float (64-bit double-precision)
  - string
  - list, tuple, dictionary, set, etc.
- You can define new classes, but many programs don't need to

Implicit vs. Explicit Declarations

Also: Ruby, Perl

- In Python, variables are **implicitly declared**
  - First use of a variable declares it; its type can change

```
x = 37
y = x + 5
x = 'abc'
```

no declaration needed; created when assigned to

x and y now exist and contain integers

x now contains a string

- Contrast Java/C/C++ use of explicit variable declarations
  - Variables must be named and typed before they are used

```
int x,y;
x = 37;
y = x + 5;
x = "abc";
```

declaration

use

use

static type error

Python Strings

- Strings are sequences of (unicode) characters
- They are immutable; once created, a string never changes
- String literals are surrounded by single or double quotes and can include escapes

```
>>> print('I am a string')
I am a string
>>> print("I'm also a string")
I'm also a string
>>> print('I have an embedded\nnewline')
I have an embedded
newline
>>>
```

Strings: Creation

- The str() function converts most any type to a string:

```
>>> str(3.14)
'3.14'
>>> str(True)
'True'
>>> str(1e7)
‘10000000.0'
>>> str()
''
```

- The print() function implicitly calls str() on non-string arguments

Strings: Immutable sequence concatenation

```
>>> s = 'abc'
>>> t = s
>>> t
'abc'
>>> u = s + 'def'
>>> u
'abcdef'
>>> s += 'def'
>>> s
'abcdef'
>>> t
'abc'
>>> t + 3.14
TypeError: can only concatenate str (not "float") to str
>>> t + str(3.14)
'abc3.14'
>>> 'ab' * 3
'ababab'
```

string concatenation: creates a new string

because strings are immutable this means the same thing as **`s = s + 'def'`**

assigning to s does not change t

no automatic conversions!

repetition: creates a new string

Quiz: Strings

What is the result of evaluating this Python expression?

```
>>> 2 * ('sha ' + 'na ' * 3) + 'na'
```

A
```
TypeError: can only concatenate str (not "int") to str
```

B
```
'sha na na na sha na na na na'
```

C
```
‘sha na na na na'
```

D
```
‘2 sha na 3 na’
```

Strings: Examining a sequence

- Individual characters can be fetched by position index

```
>>> s = 'abc'
>>> s[0]
'a'
>>> s[2]
'c'
>>> s[3]
IndexError: string index out of range
>>> s[-1]
'c'
```

indices start from 0

indices must be within range

negative indices count from the end

There is no distinct character type.
A character is just a single-element string.

Slices: specifying ranges in a sequence

```
>>> s = 'abcdefghijklmnopqrstuvwxyz'
>>> s[1:3]
'bc'
>>> s[24:38]
'yz'
>>> s[:4]
'abcd'
>>> s[20:]
'uvwxyz'
>>> s[-2:]
'yz'
>>> s[0:9:2]
'acegi'
>>> s[5:0:-1]
'fedcb'
>>> s[::-1]
'zyxwvutsrqponmlkjihgfedcba'
```

slice is inclusive on the start, exclusive on the end

happily ignores positions out of range

missing start index defaults to 0

missing end index defaults to length

we can provide a a "step"

negative steps are legal

and they reverse the defaults

More string operators

```
>>> s = 'abcabc'
>>> len(s)
6
>>> 'cab' in s
True
>>> 'bac' in s
False
>>> 'd' not in s
True
>>> s.find('bc')
1
>>> s.rfind('bc')
4
>>> s.count('ab')
2
```

strangely, a prefix function, not a method call

position of first occurrence of substring

position of last occurrence of substring

number of times substring appears

And there are **many** more…

Quiz : Slices

Assume the following definition

```
>>> s = 'abcdefgh'
```

Which of the following expressions produces a **different** result than the other three?

A
```
s[2:]
```

B
```
s[2:len(s)]
```

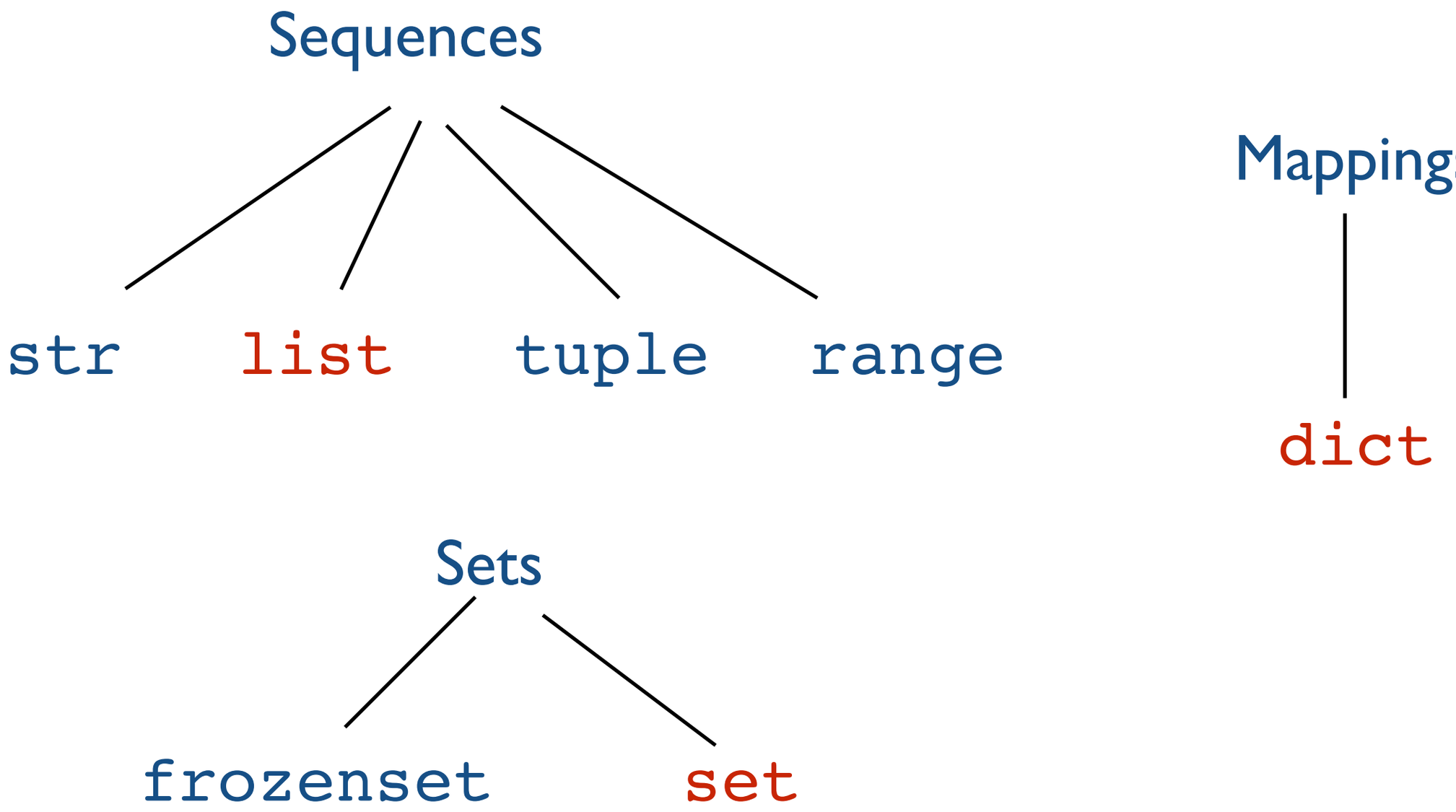
C
```
s[s.find('cd'):7]
```

D
```
'cdefgh'
```

Other Python high-level data types

(mutable)

(immutable)



Lists: a mutable sequence type

- Contains a sequence of arbitrary elements
  - Can be heterogenous — not all elements of the same type
- Literals: empty list `[]`, singleton list `[3]`, arbitrary list `[3,'abc',47,[True,8]]`, etc.
- Supports many of the same operations as strings, including indexing `[]`, slices `[:]`, membership testing with `in`, `len()`, `index()`, etc. And more:

```
>>> s = 'comma,separated,words'
>>> s.split(',')
['comma', 'separated', 'words']
>>> ','.join(['comma', 'separated', 'words'])
'comma,separated,words'
```

character to split at

bizarrely, this is method on separating string , not list

Lists: a **mutable** sequence type

```
>>> a = [1,2,3]
>>> a[1] = 4
>>> a
[1, 4, 3]
>>> a[1:2] = [5,6]
>>> a
[1, 5, 6, 3]
>>> a[2:3] = []
>>> a
[1, 5, 3]
>>> a.append(7)
>>> a
[1, 5, 3, 7]
>>> b = a
>>> a += [8,9]
>>> a
[1, 5, 3, 7, 8, 9]
>>> b
[1, 5, 3, 7, 8, 9]
```

update a single item

replace a slice with more elements

replace a slice with fewer elements

add a single element at the end

extend one list with another

because lists are mutable this
is **not** the same thing as a = a + [8,9]

Tuples: an immutable sequence type

- Essentially like lists, but immutable
- Literals: empty tuple `()`, singleton tuple `(1,)`, and arbitrary tuple `(1,True,4.5,[23,4])`, etc.
- Supports all the non-mutating operations as lists
- The surrounding `()`'s can be dropped in some contexts
- Useful for multiple return values e.g. `divmod(7,2) = (3,1)`
- and for parallel multi-assignment, e.g. `a,b = b+1,a+2`
- All sequence types are implemented using extensible arrays

Dictionaries

- Mutable collection of key -> value mappings
- Literals: empty dict `{}`, arbitrary dict, e.g. `{'a':1,'b':3}`
- Can construct with `dict(s)`, where s is any sequence of 2-element sequences, e.g. `d=dict([('a',1),['b',2]])`
- Keys can be of any type that is immutable "all the way down"
- Read and write mapping using index notation, e.g.,
  `d['a'] = d['b'] + 6`
- Can test membership with `in`, e.g. `'b' in d` yields `True`
- Remove entries with `del`, e.g. `del d['b']`
- Implemented using efficient hash tables

Sets and Frozensets

- Mutable and immutable versions of a collection with unique elements
- Literals look like **`{1,2,3,4}`** --- but for an empty set, must use **`set()`**
- Check membership with **`in`**
- There are operators to compute set union, intersection, difference, subset checking, etc.
- Unfrozen sets can add and remove elements with **`add(),remove()`**
- Implemented as a dict with empty **`None`** values for all keys

Quiz: Fancy types

Consider the value of the following expression

```
[{('bar', 20), ('foo', 10)}, {(1, 2, 3)}]
```

Which of the following describes the type of this value?

A. List of sets of tuples.

B. List of dictionaries

C. Tuple of sets of lists

D. Illegal construction, so has no type

Python: Statement syntax

- Whitespace and indentation matter!
  - Code blocks are demarcated by indentation level
  - Continue statement over lines with trailing \

```
n = 6
if n % 2 == 0:
    print("divisible by 2")
    print("hence not prime")
elif n % 3 \
      == 0:
    print("divisible by 3")
else:
    print("maybe prime")
print("we're done")
```

test.py

```
$ python3 test.py
divisible by 2
hence not prime
we're done
$
```

Python: generalized **for** loop

```
for n in [1,4,3,2]:
    print(n,end=' ')
print()
for c in 'abcd':
    print(c,end=' ')
print()
for i in range(1,4):
    print(i,end=' ')
print()
d = {'a':1, 'b':2, 'c':3}
for k in d.keys():
    print(k,':',d[k],end=' ')
print()
f = open('test.py','r')
for c in f:
    print(len(c),end=' ')
print()
```

test1.py

```
$ python3 test1.py
1 4 3 2
a b c d
1 2 3
a : 1 b : 2 c : 3
6 15 27 28 13 11 27 6 24 20
$
```

any sequence or set here

creates a sequence;
arguments like slice

one of several set views
on dicts

for each line
in file

Function Definitions

```
def f(x,y):
    c = x+y
    print('in f')
    return c+2

def g(z,w=3):
    d = z+w
    print('in g:',d+2)

print(f(1,2))

print(g(1,2))

print(f(y=6,x=3))

print(g(z=6))
```

test3.py

```
$ python3 test3.py
in f
5
in g: 5
None
in f
11
in g: 11
None
$
```

functions that don't return anything evaluate to **None**

arguments may be specified by parameter name

default values may be provided by function definition

there are ways to define functions taking variable numbers of arguments

Quiz: Iterators and functions

Assuming n > 0, what does this program print?

```
def f(seq):
    i = 0
    for n in seq:
        i += 1
    return i

print (f({1,3,5}) + f(range(1,n)))
```

A. n

B. n + 2

C. n + 3

D. Illegal construction, so prints an error

Scoping without Declarations

- Python identifiers can be defined in
  - the global scope (e.g. top-level assignments, imports)
  - the scope of a function body (e.g. parameters, local vars)
- As in many languages, uses of an identifier **x** resolve to the nearest enclosing scope that binds **x**

- But since variables are not explicitly declared it is not obvious where binding should occur
- Unusual Python feature: a variable that is written to anywhere in a function is treated as local to that function
  - Unless a `global` declaration is used

Examples

```
a = 10
def g(b):
   c = a + b
   return c
print (g(1), a)
```

11 10

```
a = 10
def g(b):
   a = 20
   c = a + b
   return c
print (g(1), a)
```

21 10

```
a = 10
def g(b):
   global a
   a = 20
   c = a + b
   return c
print (g(1), a)
```

21 20

```
a = 10
def g(b):
   c = a + b
   a = 20
   return c
print (g(1),a)
```

UnboundLocalError: local variable 'a' referenced before assignment

Files are modules

- Each script file `foo.py` defines a **module** called `foo`
- To access a function or variable defined in another module, we can either:
  - import the module and use dot notation on name we want
  - import the specific names we want from the module

```
a = 10

def f(x):
    return x + a
```

mymod.py

```
import mymod
print (mymod.f(mymod.a))
```

client1.py

```
from mymod import f, a
print (f(a))
```

client2.py

Recap: Python as a scripting language

- Interactive use (e.g. Read-Eval-Print-Loop)
- Syntax encourages brevity
- Variables have scope but no declarations
- Dynamic typing
- Strong support for string manipulation and pattern matching
- Direct access to OS system facilities and external libraries
- Built-in support for high-level types
- Interpreted execution
  - Often "defined" by single official implementation